2003

LINEAR AND NONLINEAR MODELING OF ASPERITY SCALE FRICTIONAL MELTING IN BRITTLE FAULT ZONES

Ravi V. S. Kanda
University of Kentucky, rvkand2@uky.edu

ABSTRACT OF THESIS

LINEAR AND NONLINEAR MODELING OF ASPERITY SCALE FRICTIONAL MELTING IN BRITTLE FAULT ZONES

Study of pseudotachylytes (PT) (frictional melts) can provide information on the physical and chemical conditions at the earthquake source. This study examines the influence of asperity-scale fault dynamics on asperity temperature distribution, and therefore, the potential for frictional melting to occur. Frictional melting occurs adiabatically, and is initiated between opposing asperity tips during fault slip. Our model considers 2-D heat conduction in elastic, isotropic, hemispherical asperities, with temperature dependent thermal properties. The only heat source is a *point heat flux pulse* at the asperity tip. The non-linear problem was solved using the δ-form of Newton-Kantorovich procedure coupled with the δ-form of Douglas-Gunn two level finite difference scheme, while the linear problem required only the latter method. Results for quartz and feldspar indicate that peak temperatures can reach melting point values for typical asperity sizes (1-100 mm), provided that contact (frictional) shear stress is sufficiently high. For any asperity size, the temperature distribution peak becomes insignificant by the time it reaches the asperity center. These results imply that much of asperity scale melting is highly localized, which may explain why most PT veins in the field are usually very thin. However, in some cases, successive asperity encounters may generate temperature increases large enough to trigger the massive melting inferred from typical PT exposures. Significant differences were observed between the results of the linear and nonlinear models.

KEYWORDS: Frictional Melting, Nonlinear Thermal Modeling, Pseudotachylytes, 2-D Heat Conduction, Douglas-Gunn Method

Ravi V. S. Kanda

February 2003

LINEAR AND NONLINEAR MODELING OF ASPERITY SCALE FRICTIONAL MELTING IN BRITTLE FAULT ZONES

By

Ravi V. S. Kanda

Dr. Kieran O'Hara
Director of Thesis

Dr. Alan Fryar
Director of Graduate Studies

February 2003

RULES FOR THE USE OF THESES

Unpublished theses submitted for the Master's degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgements.

Extensive copying or publication of the thesis in whole or in part also requires the consent of the Dean of the Graduate School of the University of Kentucky.

A library that borrows this thesis for use by its patrons is expected to secure the signature of each user.

<u>Name</u> <u>Date</u>

THESIS

Ravi V. S. Kanda

The Graduate School
University of Kentucky
2003

LINEAR AND NONLINEAR MODELING OF ASPERITY SCALE FRICTIONAL MELTING IN BRITTLE FAULT ZONES

THESIS

A thesis submitted in partial fulfillment of
the requirements for the degree of Master of Science in the
College of Arts and Sciences
at the University of Kentucky

By

Ravi V. S. Kanda

Lexington, Kentucky

Director: Dr. Kieran O'Hara, Associate Professor of Geological Sciences

Lexington, Kentucky

2003

MASTER'S THESIS RELEASE

I authorize the University of Kentucky
Libraries to reproduce this thesis in
whole or in part for purposes of research.

Signed: Ravi V. S. Kanda

Date: February 2003

To my parents and to my wife, Liz

ACKNOWLEDGEMENTS

The following thesis, while an individual work, benefited from the insights and direction of several people. First, my Thesis Chair, Prof. Kieran O'Hara, who exemplifies the high quality of scholarship to which I aspire. Without his financial and logistical support, this research could not have been performed. In addition, Prof. James McDonough provided me with the basic foundation for the numerical methods used here, through his very interesting and challenging courses. He also provided timely instructive comments and evaluation at every stage of the code development process, allowing me to complete this project on schedule. Next, I wish to thank the other members of my Thesis Committee: Dr. Shelley Kenner, and Prof. Alan Fryar. Each individual provided insights that guided and challenged my thinking, substantially improving the finished product.

In addition to the technical and financial assistance above, I received equally important assistance from my family. My wife, Elizabeth Springborn, provided a tremendous amount of support throughout the thesis process – providing the inspiration to work, providing technical support and assistance, critically reviewing parts of my research as well as some of the text included here, and enduring my long hours at work. My parents - who have instilled in me, from an early age, the notion that there are no bounds to what a person can achieve in life if he/she is practical, and makes a sincere attempt at making things happen – and whose moral support has not wavered from halfway around the world. Finally, I wish to thank Geological Sciences graduate students Thomas Becker and German Bayona, with who I have enjoyed having stimulating discussions about my research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION

1.1 What are pseudotachylytes?

Definition: The word pseudotachylyte refers to a rock having an appearance similar, but a origin distinct from, certain glassy basaltic rocks known as tachylytes. The term has come to refer to a particular assemblage of mesoscale and microscale characteristics associated with fault, shear or impact zones, that include: typically dark, aphanitic veins showing intrusive behavior, sharp boundaries, and included clasts and crystals of the host. The veins may contain glassy (amorphous) areas, microlites, spherulites, vesicles, amygdules, and embayed lithic fragments, newly grown high temperature minerals, and dendritic crystals; and show chilled margins and flow textures (at both the field and microscopic scale) (Magloughlin & Spray 1992, Spray 1992).

Inferred Origin: Pseudotachylytes have been interpreted as frictional melts produced during high strain rates. Spray (1995) argues that depending on shear velocity-stress-displacement relations prevailing during frictional slip, rocks produced in seismogenic zones (the brittle, upper 10-12 kilometers of the earth's crust) can be predominantly comminuted wall rock ("host-rock grounds") or fragment-melt mixes (pseudotachylytes). While melting contributes to much of the dark matrix mentioned above, comminution provides most of the clasts (macroscopic or microscopic). Also, Shimamoto and Nagahama (1992) have argued that particles below about 5μm are completely melted and are not typically observed in pseudotachylyte specimens. Indeed, particles at the lower end of the size distribution have a larger average surface area to volume ratio, making them highly susceptible to melting. Pseudotachylytes are thus products of both fracture and fusion, containing a mix of both fragments and melt (Spray 1995).

Formation Settings: Pseudotachylytes have been found to be very rare in nature. Where observed, pseudotachylytes have been found to form under a variety of situations:

- In Normal, thrust, and strike-slip fault zones (Curewitz and Karson 1999, Spray 1995, Magloughlin & Spray 1992, Swanson 1992, Scholz 1990, Sibson 1975, McKinzie and Brune 1972), and in connecting lateral ramps associated with them (O'Hara 1992). They have been interpreted to have formed at relatively shallow crustal depths (2-10 km below the surface), or mid-crustal depths (10-20 km). They have been associated with both brittle deformation within the "elasto-frictional" regime of the upper 10-12 km of the crust, and with ductile deformation within the "crystal-plastic" transition regime between 11-22 km of the crust.
- In meteorite impact structures (Spray 1997, Spray 1995, Magloughlin & Spray 1992), where they possibly form due to shock wave compression originating from a hypervelocity impact.
- In unconfined "superfaults" (Spray 1997).
- At the base of major landslides (Curewitz and Karson 1999, Masch et al. 1985, Erismann 1979, Scott & Drever 1953).

1.2 Why study pseudotachylytes?

Pseudotachylytes can be used to infer past behavior of fault zones. They have been traditionally interpreted as an indicator of high-velocity slip (> 10 m/s), and therefore, as a fossil remnant of paleoseismic events (Spray 1995, Sibson 1975, McKinzie and Brune 1972). Their presence may also be indicative of meteorite impact, in which case their distribution can help to determine the diameters of impact structures (Spray 1995). The focus of this thesis is on pseudotachylyte formation in fault zones. The goal is to improve our understanding of fault zones processes. The practical implications of studying frictional melts in fault zones are:

- Inferring the temperature and depth of formation of pseudotachylytes. Magloughlin and Spray (1992) argue that formation depth, in conjunction with lithology causes certain patterns in fault behavior. Formation depths have been inferred from (a) structures in pseudotachylyte veins, including shapes and sizes of clasts (Swanson 1992, Shimamoto & Nagahama 1992, Grocott 1981); (b) inferred melt temperatures based on chemical composition of re-crystallized minerals and pseudotachylyte matrix (Curewitz and Karson 1999, O'Hara 1992, Magloughlin 1992, Sibson 1975); (c) wear-melt ratios (O'Hara 2001); and sometimes even (d) local stratigraphy and erosion rates (Killick and Roering 1998). The latter information can be used to determine paleo-earthquake types, and tectonic settings. Ultimately, at the megascopic scale, this information can be used to support or reconstruct past tectonic events (like continental rifting or collision). An example of such an application is Curewitz and Karson's (1999) study, which further supports earlier evidence of the Early Tertiary rifting of Eastern Greenland from Scandinavia and Western Europe.
- Earthquake rupturing is now viewed as a key structural process that contributes to the cumulative evolution of fault zones (Swanson 1992). There is an association between pseudotachylyte generation and relatively long-lived, large displacement faulting and shearing (Magloughlin and Spray 1992). Quantification of temperatures attained by melts can help determine the overall energy budgets for, and stress levels causing, faulting and shearing. Grocott (1981) studied the fracture geometry associated with pseudotachylyte generation to understand the nature of fracturing during earthquake faulting. He argued that a study of pseudotachylyte-bearing fault structures can provide information that cannot be obtained through indirect seismic studies – for instance, fault behavior at the earthquake source. Swanson (1992) argued that the presence of pseudotachylyte along faults enables the distinction to be made between those seismic structures resulting directly from dynamic rupture propagation and aseismic structures that develop through plastic shearing, cataclastic flow or small-increment-cumulative-displacements.
- Last but not the least, to develop a theoretical model of frictional melting, as is attempted in this thesis, is to better understand the mechanistic (kinematic and dynamic), energetic, as well as material and lithologic constraints on fault motion. Melt volumes, wear-melt ratios, and clast size characteristics can be theoretically estimated from the total energy budget available for fault slip, and then compared to field, experimental, and chemical analysis data for calibration and/or revision.

1.3 Pseudotachylyte Constituents

Pseudotachylyte constituents have been studied extensively by earlier researchers. An enormous amount of data and information have been gathered from their geochemical and mineralogical analyses. Detailed structural observations have been carried out from the sub-microscopic scale [Scanning Electron Microprobe (SEM) and Transmission Electron Microscope (TEM)] to the field scale. A detailed overview is provided by Magloughlin and Spray (1992) and Sibson (1975), while specific regional analyses of pseudotachylyte matrix and clasts are provided in O'Hara (2001), O'Hara (1992), Curewitz and Karson (1999), Ray (1998), and Swanson (1992), amongst many others. As discussed in Section 1.1, the main constitutents of pseudotachylytes are a dark aphanatic matrix with embedded clasts.

Matrix: The pseudotachylyte matrix is typically dark (brown, black, sub-opaque to opaque), dense, and extremely fine-grained, but rarely contains optically recognizable glass (Sibson 1975). It is predominantly made up of recrystallized frictional melt, and makes up anywhere from 70-90% by volume (based on thin section analysis). The dark color of the matrix is sometimes due to the presence of felsic minerals (either re-crystallized or surviving from the host rock) like epidote, clorite and sericite, and commonly, magnetite. The matrix often displays either microlitic structures resulting from rapid chilling of a melt, or devitrification textures, both of which may be obliterated by recrystallization. Where some glass is seen, it is typically dark in color, and displays flow structures. Sometimes, the matrix contains dendritic crystal growths and/or stellate clusters of plagioclase microlites that have nucleated on porphyroclasts. Occasionally, microlites flow around porphyroclasts in a trachytic manner (microlites are aligned sub-parallel to melt flowlines), indicating that some crystallization had proceeded prior to melt solidification. Where microlitic crystallization is absent, spherulitic structures characteristic of devitrification are commonly observed. The margins of pseudotachylyte veins are often very sharp, dark, and fine-grained, cutting cleanly across quartz and feldspar grains. Sometimes, veins have irregular color variations sub-parallel to their walls, which have been interpreted to be relics of flow banding. But where the host rock contains an abundance of mafic minerals, especially biotite (a mica group mineral), these tend to be preferentially assimilated by the melt (Spray 1992, Sibson 1975) and the contact becomes ragged with cuspate offshoots of the pseudotachylyte into the host rock. Correspondingly, the composition of melt is enriched in those components comprising the melted minerals. Intense cracking and fragmentation has been observed in the host rock wall, adjacent to veins, along with channel expansion. Both effects have been linked to the dramatic rise in pressure of fluid inclusions due to the flash melting of the host rock that typically generates frictional melt. Sibson (1975) calculates that an increase in temperature of only 50° C can cause a fluid pressurization of 1 kbar. That kind of overpressurization can cause either wall rock (or a clast containing a fluid inclusion) to spall explosively and thus produce fresh fragmentation products. This type of fragmentation can be expected to exist in regions of both the wall rock and the clast which are above a critical temperature. Below this critical temperature, fluid inclusions have not been overpressurized. Based on melting and recrystallization of the matrix (Swanson 1992, Sibson 1975) and experimental studies (Spray 1995, Logan and Tuefel 1986), it has been inferred that flash temperatures as high as 1100°-1200° C must have been attained in frictional melts.

Lithic clasts: Pseudotachylyte clasts can be either primary (generated by comminution of the host rock) or secondary (plucked from the fragmented wall rock by pressurized frictional melt, especially in injection veins). The clast size distribution in pseudotachylytes has been also found to be fractal in nature (Ray 1998, Shimamoto and Nagahama 1992, Scholz 1990), with a fractal dimension close to 1.5. Other researchers (Spray 1992) have obtained fractal dimensions close to 2.6. Based on this fractal distribution "law", both Ray (1998) and Shimamoto and Nagahama (1992) have argued that clasts smaller than 5μm do not typically survive frictional melting. In consequence, the power spectrum of pseudotachylyte clast size distribution shows a corner frequency corresponding to this size. Clasts can be classified into angular and rounded, based on the degree of their melting (Curewitz and Karson 1999). Sibson (1975) argues that pseudotachylytes contain a roughly equal mixture of quartz and feldspar porphyroclasts, with occasional quartzo-feldspathic rock fragments. Quartz porphyroclasts being the most resistant to melting, are typically angular and with an intensely cracked and strained appearance. On the other hand, porphyroclasts of plagioclase feldspar, though often faulted internally with some development of strain induced twinning, tend to be sub-rounded and embayed with rather blurred outlines, perhaps resulting from partial melting. The porphyroclasts are almost always randomly oriented, but occasionally, a shape alignment indicative of flow is apparent.

1.4 Goals of this project

The primary goal for this project is to understand if, and how, individual asperities contribute to frictional melting, and whether asperity scale interactions play an important role in frictional melt generation. These are important questions since it is thought that frictional melting is initiated at asperity tips. Another issue of interest is whether individual asperities can produce temperatures high enough for frictional melting to occur, or whether it would require multiple asperity interactions.

1.5 Outline of Thesis

Chapter 2 explores the characteristics of structures and fault surfaces within which pseudotachylytes are found. Chapter 3 discusses pseudotachylyte formation mechanisms that have been inferred by earlier researchers. It discusses both wear and melt processes, and attempts to provide a generalized sequence for pseudotachylyte generation. It also presents a description of the proposed model, including a list of assumptions. Chapter 4 presents a summary of results, discussion and conclusions. Appendix A provides a detailed description of the numerical method, discretization, the FORTRAN 90 source code, *COND2D*, and code validation tests. Appendix B contains the FORTRAN 90 source code. Appendix C contains all rock and mineral property data relevant to modeling frictional melting, in the form of both tables and figures. Appendix D contains four MATLAB codes for post-processing *COND2D* output files, and used to generate plots presented in Chapter 4 and Appendix A.

2.0 PHYSICAL CHARACTERISTICS OF PSEUDOTACHYLYTE BEARING FAULTS AND STRUCTURES

2.1 Fault structures and rock associations in pseudotachylyte generation zones

A fault is defined as a fracture with relative displacement between its two faces. Fault structures are patterns of fracture, deformation, or shear found within and around faults zones. In this section, we are primarily interested in structures in pseudotachylyte-bearing faults that have been observed in the field. These structures enable us to set structural controls and boundary conditions on the frictional melting model developed in Section 4.

Rocks that occur within fault zones provide primary evidence for the processes that occurred there (Scholz 1990). Therefore, studying fault structures (either at the field scale or at the microscopic scale) is useful in identifying the mechanistic processes that created them. This in turn can be used to make a qualitative determination of the nature of the stress fields that instigated faulting, the direction of fault movement, and the extent of fault displacement. In addition, a study of fault structures may provide information on the sequence of faulting, fault reactivation. The extent of the deformation of certain rocks, or recrystalized minerals, can provide information on the energetics of faulting. Further, studying the structures in pseudotachylyte-bearing faults also provides qualitative information on the viscocity of the melt, degree of overpressure, and the nature of melting of clasts trapped in the pseudotachylyte matrix. Finally, structures like gouge trails, cavities, and pits, formed in the fault block walls due to (a) the preferential deformation and/or melting of minerals with low strengths and melting points, and/or (b) the presence of fluids, can provide specific process information for that fault.

Brittle faults are confined to the schizosphere (the brittle upper 12 km of the crust) and ductile shear zones are confined to the plastosphere (the plastic flow zone 10-15 km) (Figure 2-1). The upper crust is characterized by a breccia, gouge, and cataclasites, formed by brittle processes, whereas the plastic lower crust is characterized by metamorphic rocks and mylonites (see Scholz 1990 for textural classifications of fault rocks.

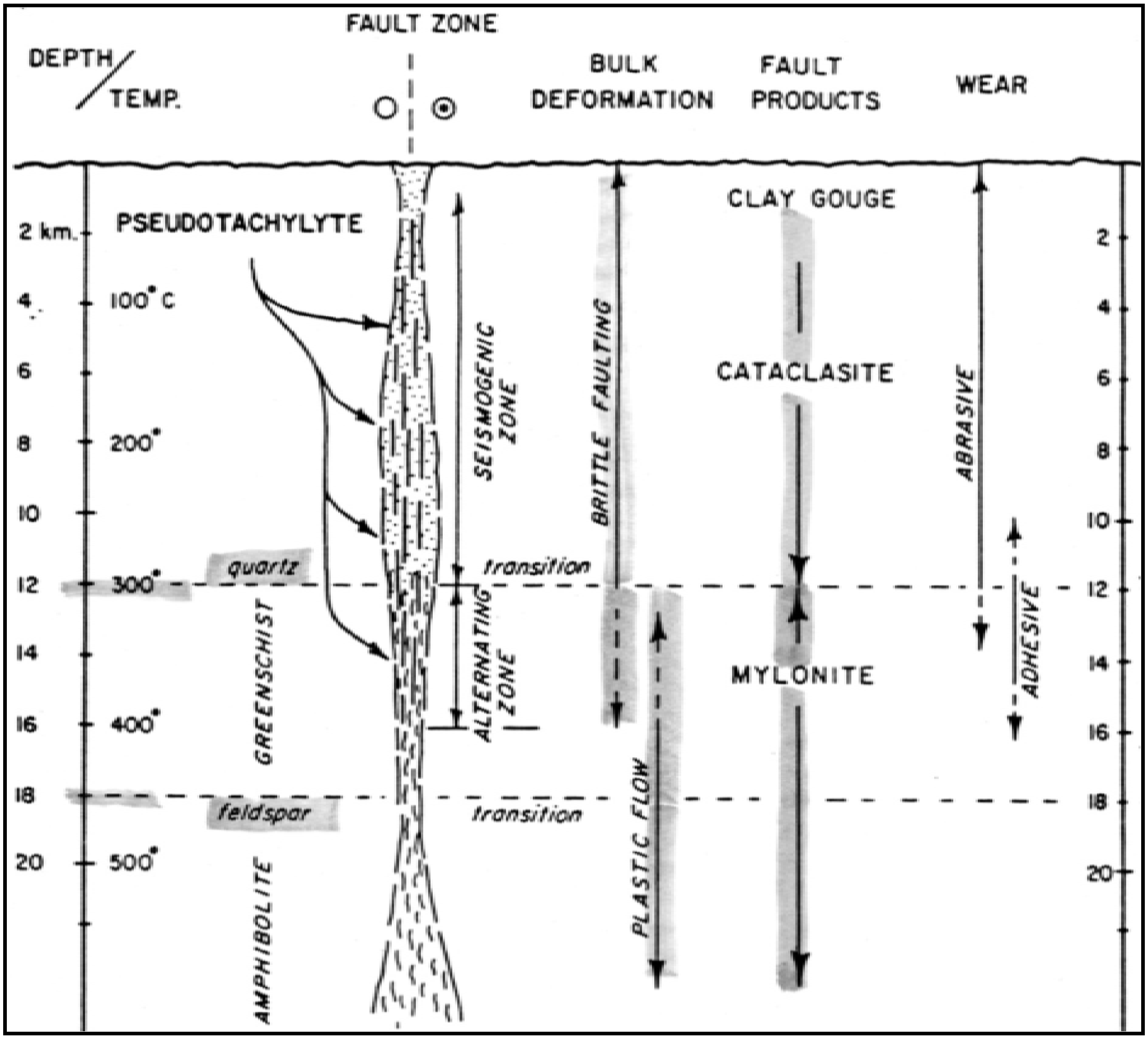


Figure 2-1. Profile through a conceptual strike-slip seismogenic zone, showing the brittle-plastic transition, variation of deformation, and wear mechanisms with depth in the crust, and the distribution of selected pseudotachylyte occurrences (some from non strike-slip sources) within both mylonitic and cataclastic fault zones. Reproduced from Swanson (1992).

Most pseudotachylytes have either been formed in the "shallow" brittle zone, or in the brittle-plastic transition zone (Figure 2-1). Some of them might have possibly undergone multiple periods of displacement before reaching the surface, while most are now exhumed due to erosion. Some pseudotachylytes might have formed deeper, in the transition zone, and have since been uplifted. Characteristic structures in pseudotachylyte formed in the brittle and ductile zones is presented below.

Brittle zone: The brittle cataclastic regime (or cataclasite regime, Figure 2-1) develops frictional melts in conjunction with active cataclasis (fragmentation) of the adjoining wall rocks from abrasive wear in the brittle deformation regime. The pseudotachylyte in some of these exposures shows multiple sequences of melting and cataclasis (Swanson 1992).

Ductile zone: The ductile shear regime (or mylonite regime, Figure 2-1) produces frictional melts that are reworked by continued plastic deformation, expressed as intermittent brittle rupturing within a background of continuous plastic shearing (Swanson 1992). Some pseudotachylyte veins produced in this regime show evidence of plastic deformation along with the adjoining host rock and development of internal foliations during shear. Flattened, recrystallized porphyroclasts and mineral aggregates are aligned parallel to these internal fabrics.

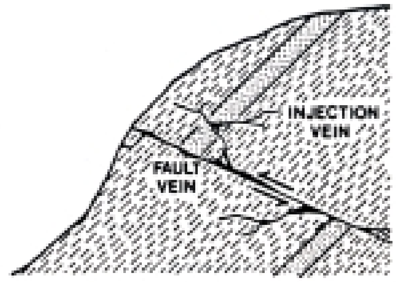
Pseudotachylyte bearing faults exposed at the surface are associated with a number of structures, including: fault and injection vein arrays, pseudotachylyte generation zones, reservoir zones, en echelon linkage duplexes, and side wall ripouts (Figure 2-2). In addition, when viewed at a larger scale, several occurrences of multiple pseudotachylyte fault vein arrays are found in distinctive structural settings that indicate repeated rupturing with identical deformation mechanisms in successive earthquake events (Swanson 1992). These arrays include en echelon arrays and complex brittle zones (Figure 2-2). Each of the above structures is briefly discussed below.

Fault veins and injection veins: Pseudotachylyte is most commonly found in fault veins and injection veins (Figure 2-2a & b) (Swanson 1992, Sibson 1975). The fault veins are typically a few millimeters to a few centimeters thick and may show variations in thickness due to irregularities in the fault surfaces. Injection veins are the most common reservoir for generated melts. These veins typically lead the melt away from generating surfaces, at near-orthogonal angles to the fault veins, into the cooler wall rocks.
Generation zones: Generation zones include paired slip surfaces that isolate tabular zones of host rock (Figure 2-2c, d & e) (Swanson 1992). These distinctive parallel fault configurations are defined by pairs of overlapping layer-parallel slip surfaces that serve as the dominant displacement structures. The fault bounded slabs between these overlapping surfaces exhibit a complex strain history. Internal fracture assemblages consisting of orthogonal dilatant veins and conjugate shear fractures indicate fault parallel extension associated with the injection of pseudotachylyte.

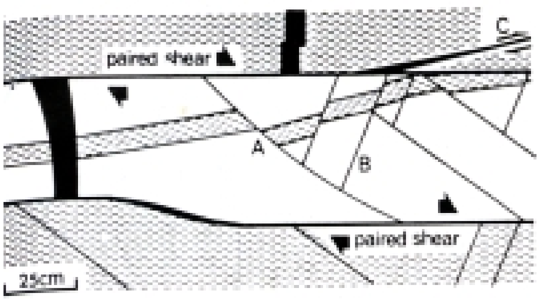
Reservoir zones: These are large, dike-like dark pseudotachylyte bodies that are commonly a few meters wide and occupy extensional voids in fault zones (Figure 2-2f). They are embedded with considerable quantities of variably sized, angular and rounded clasts. These tend to collect frictional melt that is squeezed out of the generation zones during fault displacement (Curewitz & Karson 1999, Scholz 1990, Sibson 1975).

Strike-slip duplexes: Using detailed mapping, the paired tabular structures mentioned above have been shown to be elongate areas of extensive overlap between the ends of en echelon strike-slip fault segments (Swanson 1992). Internal deformation within the tabular zones (by conjugate faulting between the slip surfaces) serves as the mechanism of displacement transfer and finite strain accommodation between the coupled fault segments during slip (Figure 2-2g). Extensional and contractional geometries of internal fracturing within the fault-bounded slabs depend on the sense of slip and stepping direction between the overlapping slab segments.

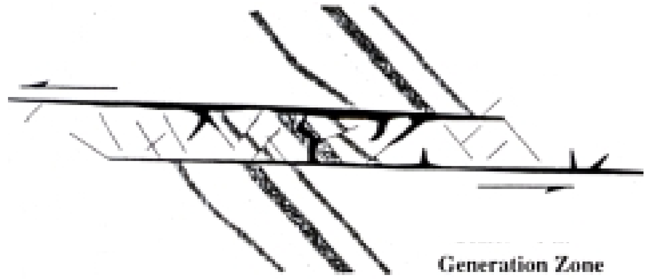
(a)



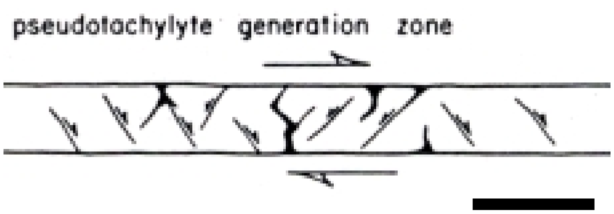
(b)



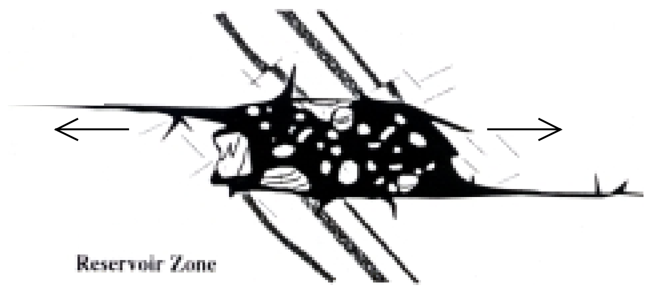
(c)



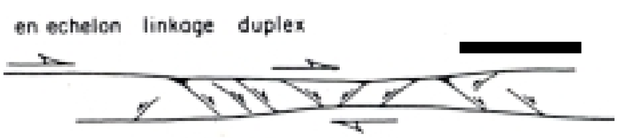
(d) Scale ~ 1 m



(e) Bar ~ 1 m

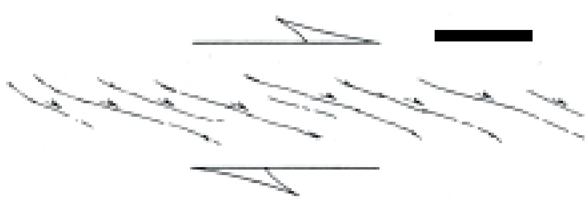


(f) Scale ~ 10 m

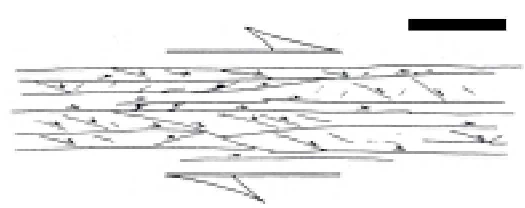


(g) Bar ~ 1 m

(h) Bar ~ 1 m



(i) Bar ~ 100 m



(j) Bar ~ 100 m

Figure 2-2. (Previous page) Schematic diagram showing the geometry of pseudotachylyte bearing faults. Basic structures: (a) & (b) Fault vein and injection veins; (c), (d) and (e) Generation zones; (f) Reservoir zone; (g) Strike-slip en echelon linkage duplex; (h) Sidewall ripouts. Structures associated with repetitive rupturing with identical fault styles and deformation mechanisms: (i) en echelon arrays; (j) brittle zones. Reproduced from Curewitz and Karson (1999), Swanson (1992), Grocott (1981), and Sibson (1975).

Whereas contractional duplexes tend to thicken with displacement through internal imbrication, extensional duplexes with severe listric fault rotations may thin catastrophically and lead to the formation of breccia within pseudotachylyte.

Sidewall ripouts: Associated with both the mylonitic (ductile) and cataclastic (brittle) fault zones, these consist of coupled extensional and contractional ramps that define tabular to plano-convex fault lenses adjacent to the dominant slip surfaces (Figure 2-2h) (Swanson 1992). They are interpreted as mesoscale examples of adhesive wear that were generated as tabular ripouts up to 35 m or more in length during slip along the main fault.
Adhesion of the fault blocks during slip ruptures one of the walls, ripping out a lens, and translating it along strike during displacement. This ripped out slab acts as an asperity temporarily, plowing its way through the adjoining wall rock, until (a) the cessation of slip occurs, or (b) it is broken up during continuing displacement.

En echelon arrays: These shear systems are indicative of intermittent coseismic slip (Figure 2-2i) (Swanson 1992). Individual shear elements occur as oblique slip surfaces or fault zones that re-orient themselves towards lower and lower angles with respect to the shear direction, and develop localized pseudotachylyte or ultramylonite shear bands.

Brittle zones: Thin pseudotachylyte veins (mm thick) are commonly found in well-defined zones of intense shear fracturing up to several hundred meters in width, particularly within anisotropic (foliated) host rock (Figure 2-2j) (Swanson 1992). These occur in complex, sub-parallel, overlapping arrays up to kilometers in length. The brittle zone itself appears to have a paired shear or duplex structure, with slip localization occurring along the outer boundary zones. Repeated rupturing in these brittle zones suggests a history of paleoseismic activity and the structural similarity between events is due to the strong structural control exerted by host rock anisotropy.

2.2 Fractal nature of fault surfaces

All real surfaces have a surface topography. Friction can be visualized in terms of shearing of points of contact between surfaces, at the topographic highs. These topographically high contact points, or protrusions on each of the contacting surfaces, have been termed *asperities*. It has been shown that this topography is fractal (or self-similar) in nature, for both natural fractures as well as natural rock surfaces, over a wide range of scales (11 orders of magnitude) (Power & Tullis 1995, Scholz 1990, Power et. al 1988). For statistically self-similar surfaces, a small portion of the surface, when magnified, looks statistically the same as a larger portion of the surface (Mandelbrot 1983). The procedure to determine this self similarity is as follows:

- Detrending the surface roughness profiles – i.e., remove any large scale (wavelength) features like slope or cyclicity
- Express the profile as a sum of sine and cosine waves using a suitable Fast Fourier Transform (FFT) algorithm
- Calculate the amplitudes of the waves as a function of their wavelengths (which represent different scales of the profile, or *profile lengths*)
- Calculate the point power spectral density as the square of the amplitude at each wavelength, and normalize with respect to profile length to allow for comparison of data from different profile lengths
- Finally, plot the point power spectral density as a function of wavelength

A detailed account of the method used by the workers above is given in Power et al. (1988). The absolute vertical level of the power spectrum indicates how rough or steep a surface is, while the slope of the spectrum tells how the roughness changes with scale. For statistically self-similar surfaces, the power spectral density curve is a straight line with a slope of exactly 3 on a log-log plot (Figure 2-3) (Berry and Lewis 1980).

It has also been found that fault surfaces are highly anisotropic. For any surface, the profile amplitude-wavelength ratio is defined as the ratio of the average value (say, root mean square) of surface roughness (length units) to that of the wavelength of the roughness profile in any given direction. Compared to the slip parallel direction, the profile amplitude-to-wavelength ratio is 1-2 orders of magnitude larger in the direction perpendicular to fault displacement. This means fault surfaces are much smoother parallel to the slip direction than perpendicular to it. This has been observed for the San Andreas Fault (Scholz 1990). Also, for fractal surfaces, the profile amplitude-wavelength ratio increases with wavelength (Scholz 1990). As shown in Figure 2-3, the power spectrum of such a surface has a slope that is close to 3, indicating that natural fault surfaces are nearly self-similar. The researchers above conclude that the fractal dimension of natural fault surfaces to be slightly over 1 [$D = (5 - Slope)/2$].

It has been argued that contact between moving fault blocks occurs at a few distinct contacting asperities, whose area is much smaller than the total fault surface area (Section 3.1.1 below) (Scholz 1990, Power et al. 1988, Sibson 1975). The implication of this is that as fault displacement progresses, contacting asperities at a lower scale (wavelength) get sheared off during slip and the contacts progressively shift to higher and higher wavelength asperities. That means that no matter what the thickness of the gouge (wear particles from fault motion – products of comminution discussed earlier), there will always be places where asperities directly abut (Scholz 1990). The fractal nature of fault surfaces provides a basis for assuming that asperity surfaces are always in contact and their contact areas are the primary sources of frictional heat generation. This is discussed in more detain in Section 3.4.

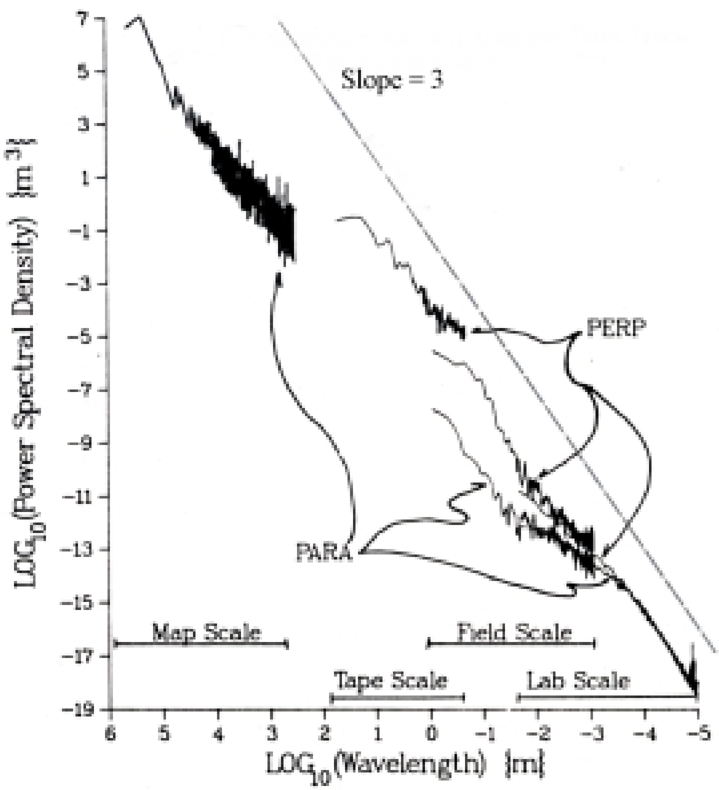


Figure 2-3. Power spectra for natural fault surfaces over 11 orders of magnitude, calculated from (a) profiles measured parallel to the slip direction (PARA) and (b) perpendicular to the slip direction (PERP). The spectra show a nearly self similar character, with a slope close to 3 (Berry & Lewis 1980). Adapted from Power & Tullis (1995).

3.0 MECHANISMS FOR PSEUDOTACHYLYTE FORMATION

3.1 Friction and deformation during slip

The earliest understanding of friction came from Leonardo de Vinci, who discovered two main laws of friction through careful experimentation, and further observed that friction is less for smoother surfaces. But his discoveries remained hidden, until they were rediscovered by Amontons, who, in his paper of 1699 (see Scholz 1990) described two laws of friction:

- *Amontons' first law:* The frictional force is independent of the area of contacting surfaces.
- *Amontons' second law:* Friction is proportional to the normal load.

He also observed that frictional force is about one third the normal load, regardless of the surface type or material. Rock friction is typically two-thirds the normal load (Scholz 1990). In the years following his paper, a mechanism of friction was sought rigorously, and the importance of surface roughness on friction was subsequently recognized. Friction was explained in terms of various kinds of interactions between protrusions on surfaces, or *asperities*, which were thought to act either as rigid or elastic springs. During the next 100 years, the difference between static and kinetic friction was also recognized.

The modern concept of friction is generally attributed to Bowden and Tabor (1950, 1964), who investigated many different frictional phenomena for a wide range of materials. Central to their work was the adhesion theory for the friction of metals. They envisioned that all real surfaces have a topography, so that when they are brought together, they only touch at a few points, or asperities (Figure 3-1). The sum of all such contact areas is the real area of contact, A_r, which is generally much smaller than the total area of contact, A_T. It is this real area of contact that is responsible for friction. They assumed that yielding occurs at the contacting asperities, causing the area of contact to increase, until it is just sufficient to support the normal load, L_T.

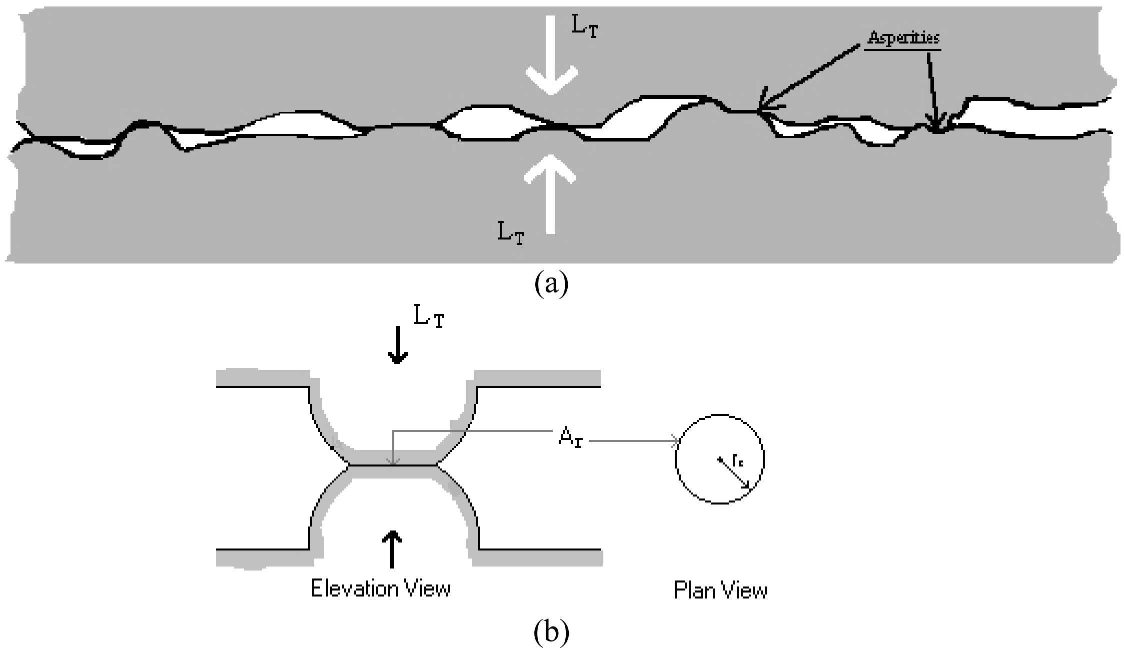


Figure 3-1. Asperity contacts during sliding of two surfaces. (a) Multiple Contacts of sliding surfaces, (b) A single idealized hemispherical contact.

Therefore from the definitions of L_T, A_r and A_T from the last page, if σ_n is the "macroscopic" normal stress on the fault, then

$$L_T = H \,.\, A_r = \sigma_n \,.\, A_T \tag{3-1}$$

where, H is the penetration hardness, a measure of the strength of the material. This deformation of asperities in response to normal load explains Amontons' second law. It must be realized that Equation 3-1 is a constitutive law describing contact between surfaces, based on plastic or elastic yielding. They supposed that adhesion occurred at the contact points due to the very high compressive stresses there, welding the surfaces together at *junctions*. In order to accommodate slip, these junctions have to sheared through, so that the friction force F is the sum of the shear strength of the junctions:

$$F = \tau_y \,.\, A_r \tag{3-2}$$

where, τ_y is the shear strength of the material. Equation 3-2 describes a constitutive law for shearing. Because any constitutive law governing this shear interaction of asperities is bound to predict a shear force proportional to A_r regardless of the exact mechanism assumed, Equation 3-1 also implicitly satisfies Amontons' first law, as long as the equation itself is linear in L_T. Combining Equations 3-1 and 3-2, friction can be described by a single coefficient of friction, μ:

$$\mu \equiv F/\,L_T = \tau/\sigma_n = \tau_y\,/H \equiv constant \tag{3-3}$$

That is, as load increases, so does the real contact area, A_r, so that the ratio $\tau/\sigma_n \equiv \mu$ remains a constant. It must be kept in mind that different mechanisms (elastic or plastic or both) might be involved in the two processes described in Equations 3-1 and 3-2, and the interaction between them could be complex.

Logan and Teufel (1986) determined experimentally - using thermodyes and a triaxial test apparatus - that this real area of contact is strongly dependent on the applied normal stress, and that the single-asperity contact area increases roughly linearly with increasing normal stress ($A_r \propto \sigma_n$). This is in agreement with the fractal asperity size distribution for fault surfaces, discussed in Section 2.2. As the normal stress increases, asperities of larger wavelengths come in contact, leading to an increase in "single-asperity" contact area. The asperity contact area is also inversely proportional to the strength of opposing asperities ($A_r \propto \sigma_n/H$). They also argue that the higher the material strength, the smaller the asperity contact area – contact area for limestone (calcite) is roughly 10 times that for sandstone (quartz), since quartz is about 20 times stronger than calcite (at room temperature). They obtain maximum real contact areas of 16% and 18% for sandstone and limestone, respectively (in the presence of confining pressure, and when opposing asperities are made up of the same material). Nadeau and Johnson (1998) used moment release rates to estimate earthquake source parameters for the Parkfield segment of the San Andreas Fault. They argue that the real (or asperity) contact area there is less than 1%. Both sets of researchers obtained typical asperity dimensions of the order of a millimeter.

3.1.1 Rock friction

Much less work has been done on the frictional properties of minerals and rocks, but the observed phenomena are much the same, and therefore, adhesion theory is assumed to be valid, especially at deeper levels in the crust. It has been postulated that frictional slip within the upper crust is dependent on the abrasion of a population of asperity contacts between sliding surfaces (Rabinowicz 1995, Swanson 1992, Scholz 1990). The localized high stresses at the contacting asperities lead to either localized brittle fracturing, and/or plastic shearing. Except at depths within the plastosphere, plastic shearing is unlikely (Figure 2-1). In the schizosphere, as fault slip commences (i.e., as relative displacement occurs), fault surface refinement progresses through wear of contacting asperities, thereby increasing the real area of contact between the sliding surfaces (Scholz 1990, Logan and Teufel 1986). It should be kept in mind, however, that the adhesion theory of friction can only be used as a conceptual framework. Webster and Sayles (1986) argue that, although Bowden and Tabor (1954) described the proportionality between contact area and load by postulating that the applied normal load is entirely supported by plastic asperity contact, Archard (1957) later showed that the proportionality can also be achieved with elastic asperity deformation, i.e. it makes no difference what the deformation mechanism is! In general abrasive wear is prevalent at lower temperatures (upper crust, Scholz 1990), and adhesive wear at higher temperatures (lower crust, Swanson 1992).

For hard materials such as the silicates, contacts can be assumed to be highly elastic, and the contact area of an asperity can be obtained from Hertz's solution for contact between an elastic sphere on an elastic substrate (Wang and Scholz 1994, Scholz 1990). Hertzian contact theory for a spherical asperity predicts that the elastic deformation, and hence contact area (A_r), are both proportional to $L_T^{2/3}$ (Wang and Scholz 1994, Scholz 1990, Timoshenko and Goodier 1970), where L_T is the total normal load on the fault surface. That is:

$$A_r = k_1 L_T^{2/3} \tag{3-4}$$

For a large number of such self-similar hemispherical asperities (successively smaller scale spherical asperities superimposed on top of larger ones) in elastic contact with a flat substrate, a linear relationship between A_r and L_T is obtained asymptotically (Archard 1957). In other words, contact area A_r is proportional to L_T (Equation 3-1) in the limit of a large number of superimposed scales. Thus the microscopic and macroscopic constitutive friction laws are dramatically different. While Equation 3-3 defines a constitutive law for μ at the macroscopic scale, the constitutive law for the microscopic scale becomes (from Equations 3-2, 3-3 and 3-4):

$$\mu = \tau_y k_1 (L_T)^{-1/3} = \tau/\sigma_n \tag{3-5}$$

which has been shown to be true for hard materials (Scholz 1990). It must be kept in mind that frictional shear resistance evolves during coseismic slip, from static to lower dynamic values, as the fault surfaces evolve. Once friction is lowered to its dynamic value, further increases in strain rate or slip velocity cause it to decrease only a few percent more for an order of magnitude increase in slip velocity (Rabinowicz 1995, Scholz 1990).

Contact geometry: The elastic contact surface, between ball and race of a ball bearing, as well as that between a ball and a flat surface, has been shown to be elliptical (Spence and Kaminski 1996, Harris 1966). Wang and Scholz (1994) used Timoshenko and Goodier's (1970) results and postulated a circular contact area between two elastic, hemispherical fault surface asperities in contact with each other (Figure 3-1). For simplicity (and for reasons elaborated in Chapter 4.0), a circular asperity contact geometry is assumed in this thesis.

In studying the friction of any class of materials over any given range of conditions, interfacial deformation mechanisms specific to the conditions and materials become important. Analytical and numerical analyses of elastic asperity contacts have been undertaken in the field of tribology for the purposes of analyzing ball bearing frictional forces and deformations (using Finite Difference (FD), or Finite Element (FE) schemes – see Lowell and Khonsari 1999, Lowell et al. 1997, Lowell et al. 1996, Webster and Sayles 1986, Harris 1966). Analyses have even considered spheres in contact with highly anisotripic flat surfaces (Kuo and Keer 1992). Singh and Paul (1974) have developed an analysis for "non-Hertzian" contact problems with frictionless surfaces containing asperities of arbitrary shape. All these analysis were for lubricated metals, under controlled conditions more relevant to engineering applications. Fault motion occurs under more chaotic and uncontrolled conditions. Nonetheless, results from such analyses can be used as a starting point for better understanding of rock friction mechanisms. Such analysis of friction is beyond the scope of the current project. As in studies by Archard (1953, 1957) and Scholz (1990), the adhesive theory of friction and Hertzian contact theory are the basis of the heat flux calculations of Section 3.4.

3.2 *Wear and gouge formation during slip*

Since friction during slip within the upper crust is dependent on the abrasion of asperity contacts between sliding surfaces, surface damage during sliding results in wear due to the interlocking and ploughing of asperities (Rabinowicz 1995, Swanson 1992, Scholz 1990). The localized high stresses at the contacting asperities lead to either localized brittle fracturing and/or plastic shearing. Abrasion dominated wear, characteristic of the brittle zone (up to a depth of about 12 km), changes to adhesion dominated wear, and ultimately to continuous adhesion wear through plastic deformation at depths greater than about 18 km.

The abrasive wear domain is characterized by brittle behavior and unstable frictional slip with fracturing of asperities, development of loose wear particles, and the production of a cushion of cataclasite. The adhesive wear domain is characterized by semi-brittle behavior and stable frictional slip with plastic deformation of the asperities and material transfer to opposing faces of slip. It is this surface refinement that produces a deformed layer of processed asperities that may, ultimately, lead to shear heating and frictional melting as the surface evolves. As mentioned at the end of Section 2.2, it is important to remember that no matter what the gouge thickness predicted by the following models, asperities are always in contact. Further, asperity size increases with increasing displacement and increasing gouge volume.

One of the first empirical relationships between slip (D) and pseudotachylyte thickness (T) came from Sibson (1975), who obtained:

$$T = \sqrt{\frac{D}{436}} \tag{3-4}$$

where, T and D are in centimeters. He made a case that the gneissic rocks he studied came from melts formed during seismic slip, and were therefore dimensionally controlled by frictional heating, rather than wear. To argue this, he first calculated frictional shear stress τ_f:

$$\tau_f = \frac{q}{D}T = \frac{4.75\text{x}10^{10}\, ergs/cm^3}{D}\sqrt{\frac{D}{436}} = \frac{5.4\text{x}10^7}{T}\, dynes/cm^2 \tag{3-5}$$

where the number in the numerator of the middle equality is the energy required to melt a unit volume of acid gneiss. He argued that if the melt were assumed to be a Newtonian fluid, further movement is opposed only by its viscous resistance to shear. The resistance to shear would be directly proportional to the rate of shear straining. The shear-strain rate would be inversely proportional to layer thickness (T).

One of the earliest theoretical derivations of wear in fault zones was by Archard (1953), whose method is independent of the specific wear mechanism. His method can be summarized as follows (Scholz 1990): Assuming i) a linear relationship governs the relationship between the normal force and contact area, ii) a hardness parameter H, iii) a total normal force on the fault of L_T, and iv) circular contacts of diameter d, then there are n contacts given by

$$n = \frac{4L_T}{\pi H d^2} \qquad (3\text{-}6)$$

Assuming that each contact junction exists for an effective working distance of d_e, i.e., $d_e = \alpha d$, where α is a constant with a value near unity (Rabinowicz 1995), each junction must be replenished $1/ d_e$ times per unit of travel, so that the number of junctions per unit of travel is given by

$$n_D = \frac{n}{d_e} = \frac{4L_T}{\alpha \pi H d^3} \qquad (3\text{-}7)$$

If the probability that any junction will shear off is k, and on the assumption that the fragment formed by shearing is a hemisphere of diameter d, the wear rate is given by:

$$\frac{\partial V}{\partial x} = \frac{k\pi d^3}{12} n_D = \frac{kL_T}{3\alpha H} \qquad (3\text{-}8)$$

where, V is the volume of the gouge, x is the slip coordinate, and $\pi d^3/12$ is the fragment volume. Therefore, the volume of gouge, or new material, formed per unit displacement, D is

$$V = \frac{kL_T D}{3\alpha H} \qquad (3\text{-}9)$$

which, neglecting the porosity change, produces a gouge zone of thickness T given by

$$T = \frac{\kappa \sigma_n D}{3H} \qquad (3\text{-}10)$$

where σ_n is the normal stress and $\kappa = \mathrm{k}/\alpha$ is a dimensionless wear coefficient parameter. This model predicts a linearly increasing gouge zone thickness with increasing fault displacement. One limitation is that this model cannot predict wear rates resulting from different materials on either side of the fault, as it does not consider the differences in grain boundary strength between the two rocks (Scholz 1990). Another limitation is that the model applies only to steady-state wear. A complete wear curve also contains an early “running-in” phase, in which high initial wear rates decay exponentially with sliding until a steady-state rate is finally achieved. The usual explanation for running in wear is that the starting surfaces are rougher than those that are in equilibrium with the sliding conditions. Fresh surfaces have an initially high wear rate that is proportional to this excess roughness (Scholz 1990, Power et al. 1988, Queener et al. 1965).

The next advance in wear zone determination was by Power et al. (1988), who assumed that since natural fault surfaces are fractal, both the RMS roughness (root mean square roughness – the square root of the sum of squares of profile amplitudes along a particular direction) and the average centerline roughness of the fault increase with increasing slip (Section 2.2). That is, the amplitude of the asperities, on average, increases with their wavelength. No matter how thick the gouge becomes there will always be places where asperities directly abut. In these regions, wear is expected to be high. In Power et al. (1988) model, the surfaces are continually running-in because steady-state smoothness is never achieved. Their model closely parallels that of Queener et al. (1965), except for the initial assumption of average asperity roughness increasing with fault displacement. Since this thesis is concerned with heat generation at a single asperity contact, and is independent of wear, average roughness and wear were not considered. The above discussion of wear was presented for the sake of completeness. Because of its higher surface area to volume ratio, however, wear material may be easier to melt.

3.3 A generalized frictional melting sequence for pseudotachylyte generation

In Section 2.1, both brittle and ductile regimes for pseudotachylyte formation were discussed. This section summarizes the main events in the frictional melting sequence. The summary will lay a foundation for the overall model developed in Section 3.4. This section also indicates the current conceptual ideas about how frictional melting occurs during fault motion. So, only the conceptual model outlines of current models are provided here. Details of adhesion-dominated plastic zone frictional melt generation mechanisms are discussed first.

As discussed in Section 2.1, adhesive wear-dominated melt generation operates at lower crustal levels. The adhesive sequence develops within active mylonitic fault zones that may be dominated by anisotropy controlled shear fracture propagation (Swanson 1992). In such rocks, the reactivation of pre-existing planar anisotropy during rupture provides a near-planar slip surface with few initial asperities and low initial wear rates during slip. Rapid surface refinement with a transition to total adhesion, as the real area of contact approaches the total area, leads directly to plastic smearing and laminar plastic flow without the extensive development of cushions of cataclasite. The surface refinement process is greatly accelerated, thereby enhancing adhesion, plastic flow, and frictional melting during slip. This results in a much greater potential for pseudotachylyte generation (Swanson 1992).

In the abrasive wear-dominant regime at the upper crustal levels, the abrasive wear sequence develops within active fault zones dominated by cataclasis. The sequence of events can be described as follows (Swanson 1992, Sibson 1975) (also refer to Section 2-1):

i. Initial rupture propagation consisting of oblique tension fracture arrays at shallow levels and en echelon R-shear arrays at deeper levels.
ii. Surface refinement proceeds through forward clast rotation and comminution of the initiation breccia, or through P-shear linkages in the en echelon array. Asperity reduction is through brittle fracture, brecciation, comminution with high initial wear rates ("running-in" wear of Section 3.2), frictional heating, and the initiation of melting of comminution products. Friction will vary from static to lower dynamic values in case of development of a new throughgoing surface, and may drop suddenly due to melting and thermal pressurization of the fault zone. However, the fault planes themselves remain thin (~ a few mm to 1 cm) keeping asperities in contact [see (iv) below] and allowing further melt to be generated. Wall rocks are flash melted and, in some cases, superheated during shear. Peak average temperatures of 1000° C and as much as 1520° C have been estimated from theoretical calculations (McKinzie and Brune 1972), host rock melt relations (Sibson 1975), and quartz glass compositions (Wenk and Weiss 1982). Offset,

pseudotachylyte-generating shear fractures may be linked by a set of irregular injection veins in a ladder network (Figure 3-2(i)).

iii. Continued slip leads to refined particulate flow within a cushion of cataclasite as it builds up along the fault surface. Grain size reduction proceeds to some critical level, where further strain becomes localized along oblique R-shears within the cataclasite layer, or along the wall rock / fault zone interface. Pseudotachylyte in these active cataclasite fault zones tends to be thin fault veins sporadically developed along the margins of evolved cataclasite layers where shear strain has localized with high enough slip rates for frictional melting. Some pseudotachylytes may develop from a comminuted precursor, particularly at shallow crustal levels (also see Jacques and Rice 2002).

iv. As slip continues, pseudotachylyte from the bounding fault veins along the margins of the cataclasite are injected into the growing void (induced by slip, see Figure 3-2(ii)), while the fault planes themselves (on either side of the cataclasite) remain almost “barren”, thereby retaining the frictional resistance required for further pseudotachylyte generation.

v. Continued injection of pseudotachylyte, tensional fracturing of breccia fragments within the fault zone, attrition brought about by rotational grinding, explosive decrepitation (spalling) from fluid inclusion overpressurization, and corrosion by melt, all contribute to the rounding of the clasts in the quasi-conglomerate that exists at this point (Figure 3-2(iii)).

vi. Melt lifetimes may range from microseconds to several minutes or hours (or even days), depending on slip velocity, slip duration, and reservoir dimensions. Hydrated micas are preferentially melted, because of lower melting points, followed by feldspar and lastly, quartz. The melt solidifies during post-seismic quiescence, but preserves features related to processes associated with the slip event. While glassy veins and chill margins suggest rapid solidification, microlitic textures indicate slow, static crystallization.

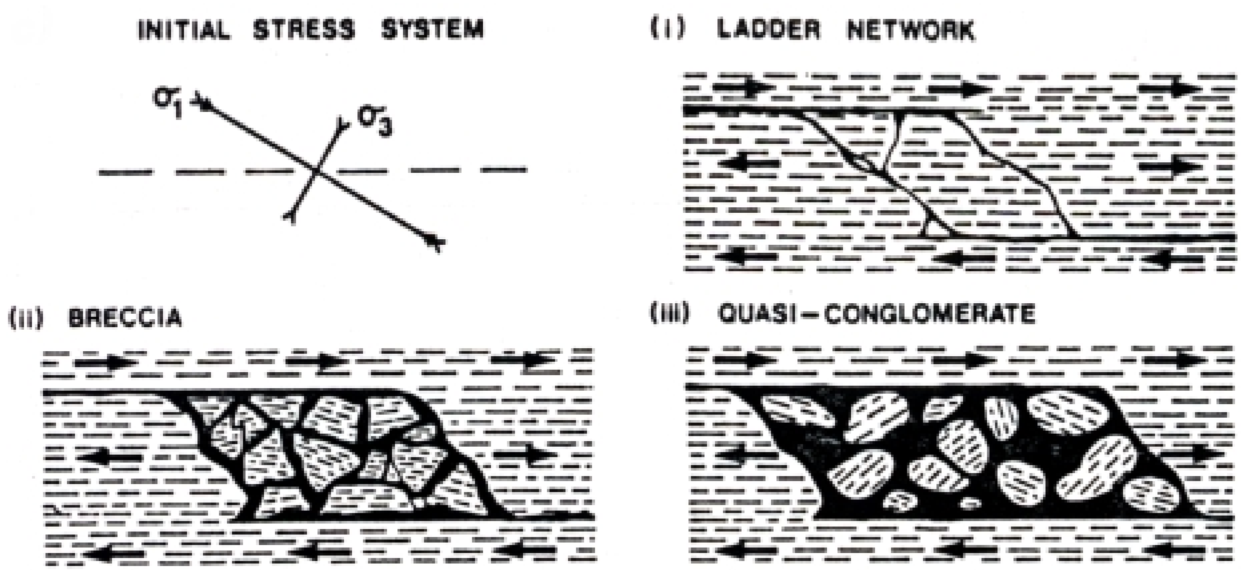


Figure 3-2. Mechanism for quasi-conglomerate and pseudotachylyte formation. Reproduced from Sibson (1975).

3.4 *Proposed frictional melting model*

3.4.1 Background

Although significant insights into the formation and mechanics of pseudotachylyte formation have been obtained over the last twenty years, not much research has focused on heat generation at the asperity scales and its implications on asperity-level frictional melting. Although temperature rises have been "constrained" based on slip along fault surfaces modeled as semi-infinite half-spaces (McKenzie and Brune 1972, Sibson 1975, Cardwell et al. 1978, Swanson 1992, Killick and Roering 1998, Kanamori et al. 1998), such analyses do not have sufficient spatial resolution to consider asperity level mechanisms of frictional melting. Archard (1958-59) analyzed the flash (maximum) temperatures attained during frictional sliding for a hemispherical asperity sliding over a flat surface, using physical rather than mathematical arguments. He used a simple thermal resistance model for low velocities. For intermediate and large velocities, he assumed one dimensional linear heat flow into a semi-infinite solid, thus neglecting asperity effects. Barber (1967,1970), while analyzing the heat distribution between two sliding surfaces, developed an approximate transient heat flow solution for small times. However, this analysis falls short of obtaining the complete transient temperature distribution. This could become important at larger asperity scales. Yovanovich (1966) investigated the problem of steady state heat transfer between metallic spheres constrained elastically between two semi-infinite half-spaces, by arguing that symmetry reduces the problem to cylindrical coordinates. Yovanovich (1966) also considered conductive heat transfer between the gas surrounding the sphere and the half spaces and radiative heat transfer between the sphere and the half spaces. He assumes that the spheres do not experience any significant heating. These two assumptions (steady state temperature distribution and a lack of significant heating) are not appropriate for asperity interactions during frictional melting. This problem is a highly transient process and produces extremely large temperatures compared to the bulk rocks of the fault walls.

Another body of work on frictional contact of asperities, carried out in engineering tribology, attempts to understand slip rate dependence of dry friction in metals at high rates [Bowden and Thomas (1954), Ettles (1986), Lim and Ashby (1987), and Molinari et al. (1999)]. These same concepts were applied by Rice (1999) to flash heating in rock with contacts of the order of a few micrometers in length. This is near the lower bound of elastic asperity areas used in this study. However, the Rice (1999) model is 1-D and the slip weakening temperature is assumed to be 900° C. Above this temperature, shear stress is assumed to be negligible. If the 900° C cap were correct, then no melt would be generated from frictional contact at asperity tips, based on the temperatures quoted in the previous section. Also, this temperature cap is assigned without actually considering the thermal evolution of the asperity itself.

Although Carslaw and Jaeger (1959) present solutions to the spherical heat conduction equation, the presented solutions are for linear problems. Most are for symmetric boundary conditions. As described below, the boundary conditions for this problem are highly abrupt and asymmetric. We are concerned with a finite, hemispherical body (the asperity), which has an "instantaneous"

AND “point” heat source at its tip (Figures 3-6) instead of at its base. In the latter case, the solution could be directly deduced from the results of the above authors.

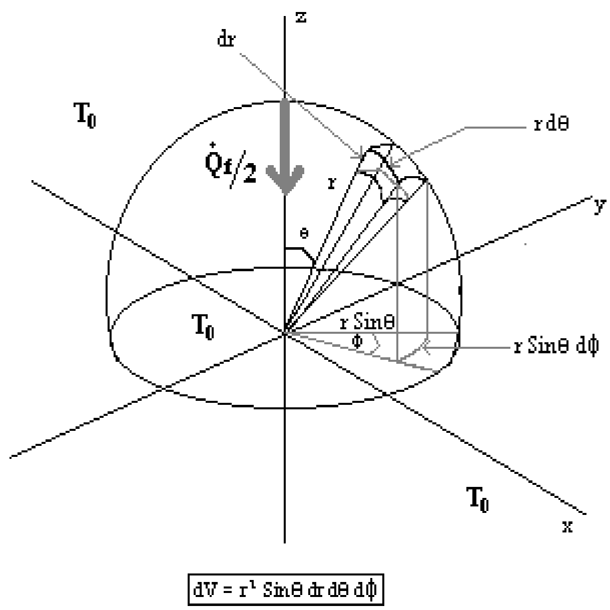


Figure 3-3. Problem setup for determining the temperature distribution within a single hemispherical asperity.

Thus, there is a need for a model for estimating asperity scale temperature distribution from frictional heating. A single asperity pair interaction is the simplest scenario for which this can be developed to understand asperity scale fault dynamics. This model can be used to determine if high temperatures can be attained after a single contact “event” or if it requires multiple contacts. The presented model can also be used to check the temporal evolution of the flash temperature pulse, and to see if and how a sharp temperature pulse in one asperity affects adjacent asperity temperatures. Although the overall energetics determine the presence or absence of frictional melt, we assume that it is the asperities that generate the bulk of the frictional heating and melting. The main focus of this thesis is to understand PT formation in brittle fault zones. We want to estimate the maximum attainable flash temperatures at the asperity scale, the effect of asperity size and contact shear stress on the evolution of the temperature distribution within an asperity, and understand inter-asperity thermal interactions (if any).

3.4.2 The Conceptual Model

3.4.2.1 *Model Outline and Assumptions*

Figure 3-4 summarizes some of the salient points of the model adopted here, and the following list provides a detailed outline of the model framework and assumptions:

i. This analysis assumes a vertical strike-slip fault lying in the brittle crust. Motion is purely strike-slip, such that gravitational work is negligible.
ii. The materials on either side of the fault have identical mechanical and thermal properties. The properties are homogenous and isotropic. The property values are assumed to be scale independent. Thermal properties are strongly dependent on temperature (e.g., conductivity).
iii. Asperities are hemispherical (Figures 3-1 to 3-4), and individual asperity contacts are assumed to be elastic (Hertzian), resulting in circular contact areas (Figure 3-4). Barber (1970) and Cameron et al. (1964) concluded that the shape of the heat source has negligible effect on the temperature distribution for two sliding solids (for circular, square or band sources).
iv. Individual asperity contact areas are small enough, and velocities large enough, that the contact duration is of the order of < 1-4 milliseconds. Therefore, the asperity contact process can be considered adiabatic. All frictional work at the contact is converted into heat energy input to the asperity. This means that once the heat flux pulse vanishes (when the asperities separate), a zero heat flux boundary condition can be used for the rest of the duration of simulation.
v. Interaction between the fault gouge and the asperity is ignored. Deformation within the fault gouge is also neglected.
vi. Because fault surfaces are fractal in nature (Power et. al, 1988, Scholz 1990, Power and Tullis, 1995), asperities are always in contact during fault slip. As gouge is being produced by the shearing off of asperities of a particular wavelength, contacts at a larger wavelengths are exposed.
vii. Friction (or shear stress) is assumed to be independent of fault slip rate.
viii. For the linear problem (constant thermal properties), the superposition principle can be used to determine the temperature distribution at any depth can be computed from the average geothermal gradient added to (or subtracted from) that at a given depth. This is not true for the non-linear problem.
ix. A pure conduction heat transfer model can approximate the actual flash temperature profiles and their evolution with reasonable accuracy. More complicated concepts like the effects of melt convection and radiation, different geometries, and melt fronts are ignored. Ignoring radiative heat transfer is reasonable because, except at discrete "points" (asperities) the bulk of the host rock does not attain considerable temperatures (see below) for typical durations of fault slip. Convective heat transfer can be ignored since we are only interested in temperatures <u>up to</u> melting.
x. It is assumed that the fault zone containing the asperities is bounded by two semi-infinite half slabs of low thermal conductivity (a realistic assumption for rocks). Thus, the fraction

of heat that diffuses in a direction perpendicular to fault motion is small compared to the heat generated within the fault zone due to friction [Barber (1970)]. Heat diffusion perpendicular to the fault surfaces is characterized by a penetration depth given by $(\kappa_h\ t_0)^{1/2}$ where k is the rock thermal diffusivity and t_0 is the duration of faulting (Kanamori et al. 1998). Since Prandtl numbers ($r_c.V_{slip}/\kappa$) for fault slip are typically greater than 1, the flash temperature pulse "penetration depth" into the asperity is very small. In other words, as fault displacement progresses, the rate of increase of asperity size (from exposure of higher wavelength asperities) is larger than the rate at which heat penetration depth increases within the asperity. Hence, as a first approximation, it seems reasonable that only a single asperity needs to be considered as the flash temperature pulse generated in it may not ever propagate out of its domain (i.e., neighboring asperities are not affected). Also, for this same reason, it is reasonable to consider a full spherical domain, defined by adding an image of the hemispherical asperity within the bulk rock, for solving this problem. Such an assumption will allow us to take advantage of the symmetry of a 2D spherical problem. This is illustrated in Figure 3-5. The two θ boundary conditions and the boundary condition at $r = 0$, shown in that figure, are now such symmetry conditions. The fourth boundary condition is given by Equation (8) (see Figure 3-3). These boundary conditions are less restrictive than prior studies.

Data from earlier theoretical, field and experimental studies provide constraints for the model parameters used here. These are presented in detail in Table C-1 (Appendix C)

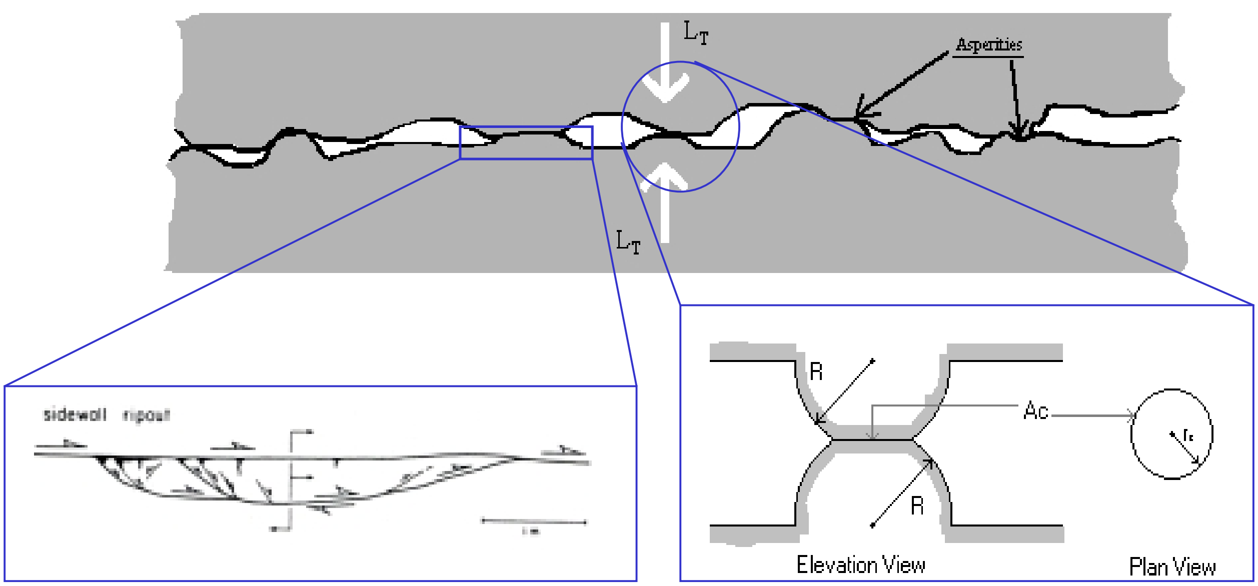
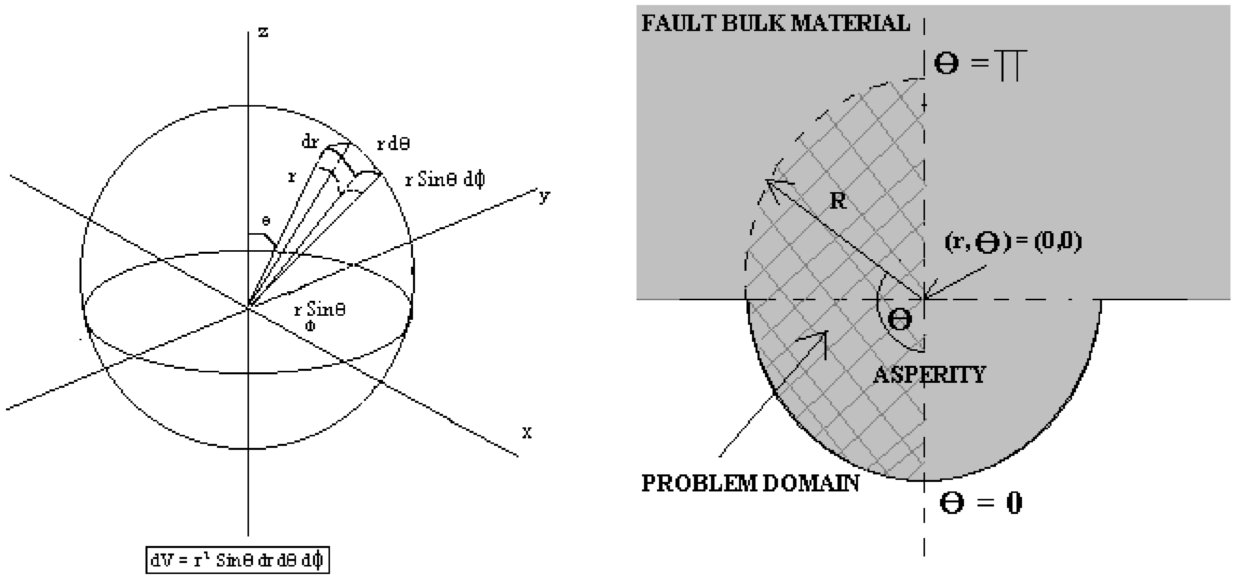


Figure 3-4. Schematic representation of asperities on a real fault surface, and their hemispherical idealization. The image at the bottom right shows an elevation view of two hemispherical asperities of identical radii *R*, in elastic contact with each other. The contact results in a circular contact area, A_c, between them, with a contact radius, r_c, as shown at the right of that figure.



(a) **(b)**

Figure 3-5. (a) Full spherical domain used in solving the problem defined above. (b) This shows a cross-section of the fault, along a plane passing through the centers of opposing asperities. The 2D problem domain (cross-hatched area) is rotated 90^0 with respect to the asperity cross-section. This assumption is valid because of the extremely low thermal diffusivities of rock materials.

3.4.2.2 *Asperity contact area, and duration of contact*

As mentioned in the previous section, we assume elastic deformation of hemispherical asperities. Elastic deformation implies that the two asperities are rigid (made up of extremely hard materials), and that the deformation produced is very small compared to the asperity dimensions. The elevation view of two such contacting asperities is shown in Figures 3-1 and 3-4. Due to the fractal nature of the fault surfaces (Sections 2.2 and 3.1.1), it must be kept in mind that these asperities represent only one of the many scales of asperities present on a natural fault surface area.

The expression for Hertzian contact between two hemispherical asperities of radii R_1 and R_2 having different elastic properties is (Timoshenko and Goodier 1970):

$$r_c = \frac{\pi\sigma_{\max}}{2E'}\left(\frac{R_1 R_2}{R_1 + R_2}\right) = \frac{3\pi\sigma_n}{4E'}\left(\frac{R_1 R_2}{R_1 + R_2}\right) \qquad (3\text{-}11)$$

where σ_{max}, the maximum stress, is 1.5 times the average stress, σ_n. E' is defined as:

$$\frac{1}{E'} = \frac{(1-\nu_1^{\,2})}{E_1} + \frac{(1-\nu_2^{\,2})}{E_2} \qquad (3\text{-}12)$$

where E_1 and E_2 are the elastic moduli of the two fault surfaces and ν_1 and ν_2 are their Poisson's ratios. If the two asperities have the same radii, R, as shown in Figure 3-4, and are made up of the same material, then

$$r_c = \frac{3\pi\sigma_n}{4}\left(\frac{1-\nu^2}{E}\right)R \tag{3-13}$$

and the contact area is defined as

$$A_c = \pi r_c^2 \tag{3-14}$$

Logan and Teufel (1986) have shown experimentally that the contact area per asperity, A_c, as well as the total real contact area (= A_c x asperity density) increases nearly linearly with an increase in normal stress, although the asperity density saturates quickly with increasing normal stress. For typical values of parameters in Equation (3-13), the ratio (r_c/R) is roughly 6% (assuming: $\mu \in [0.6,0.85]$, $\nu \in [0.20,0.25]$, $E \in [20,75]$ GPa, $\tau \in [0.001,1]$ GPa for quartz; $\in$ is a symbol for "belonging to the range"). Because this ratio is so small, it is also approximately equal to the angular contact extent in radians, θ_0:, which can be defined from Figure 3-1 and 3-4 as

$$\theta_0 = Tan^{-1}(r_c/R) \approx (r_c/R) \tag{3-15}$$

The duration of asperity contact is given by the time taken for either asperity to traverse a distance of twice the contact area diameter, $2d_c$, at the slip velocity, V_{slip}:

$$t_0 = 2d_c/ V_{slip}:= 4r_c/ V_{slip}: \tag{3-16}$$

Equations (3-15) and (3-16) will be used in the next section to compute the heat flux boundary condition.

3.4.2.3 *Heat generation*

The contact area between the two asperities changes with time, as the upper asperity moves relative to the lower asperity due to fault motion (Figure 3-6). The rate of work done per unit asperity surface area during a differential fault displacement ds, occurring in a time increment, dt, is

$$\frac{1}{A(s)}\frac{dw_f}{dt} = F(s).ds = \tau.\frac{ds}{dt} = \tau.U \tag{3-17}$$

where $A(s)$ is the instantaneous area of contact and s is the distance between the asperity centers in plan view. As the fault motion continues at a constant velocity, U, this area first increases and then decreases. It can be seen that the overlap area (shaded area in Figures 3-6 and 3-7) between the asperities varies from 0 initially, to A_c [$=\pi a^2$, see Equation (3-14)] at maximum overlap, to 0 again, as the upper asperity first approaches, then completely covers, and finally leaves the lower asperity. The overlap area at any distance s between the asperity centers is shown in gray shades in Figure 3-6 and black, in Figure 3-7. For this moving boundary scenario, the boundary heat flux will vary as shown in the bar graphs below asperity contacts in Figure 3-7. This boundary condition can be described in terms of time-dependent Heaviside functions (see below). Using the moving boundary condition depicted in Figure 3-7, however, requires the solution of the heat conduction problem (next section) in a 3D domain. Due to (1) the extremely fast interactions between the asperities (contact durations of the order of a few milliseconds), (2) the small asperity sizes, (3) extremely low thermal diffusivities in rocks, and (4) the assumption of homogeneous and isotropic material properties, it is possible that the additional development time and computational cost required for a 3D code will not yield results that are significantly different from those of a 2D code with a more symmetric boundary condition. Therefore, a 2D (azimuthally symmetric) adiabatic boundary condition was developed for this problem, assuming a *point heat flux pulse, q_f* at the hemispherical surface of the asperity. This boundary condition is similar to Equation (3-17), but it is defined with respect to the hemispherical asperity spatio-temporal domain:

$$q_f = \{k(T)\}\frac{\partial T}{\partial r}\bigg|_{r=a} = \tau V_{slip}\left[H(\theta) - H(\theta - \theta_0)\right]\left[H(t) - H(t - t_0)\right] \qquad (3\text{-}18)$$

where θ_0 and t_0 are given by Equations (3-15) and (3-16) respectively, and H is the Heaviside function defined as: $H(x\text{-}a) = 0$ *if* $x < a$; $H(x\text{-}a) = 1$ *if* $x \geq a$.

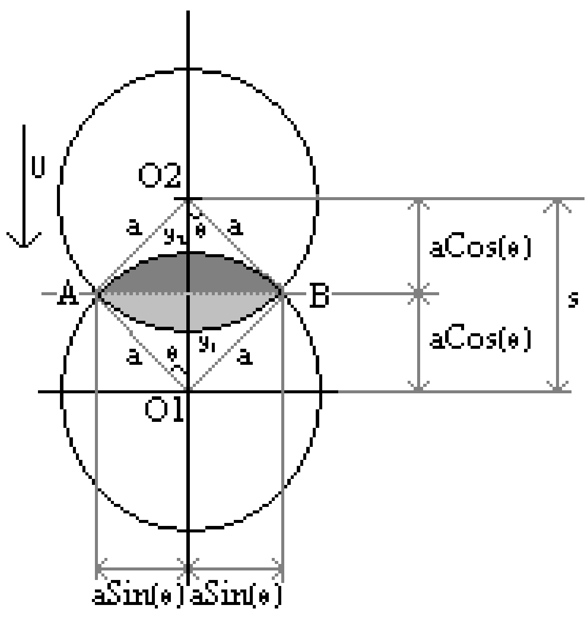


Figure 3-6. Plan view of asperity motion depicts a change in overlapped contact area with distance between asperity centers. The figure shows the two contact areas, A_c, of the asperities of the same size moving past each other.

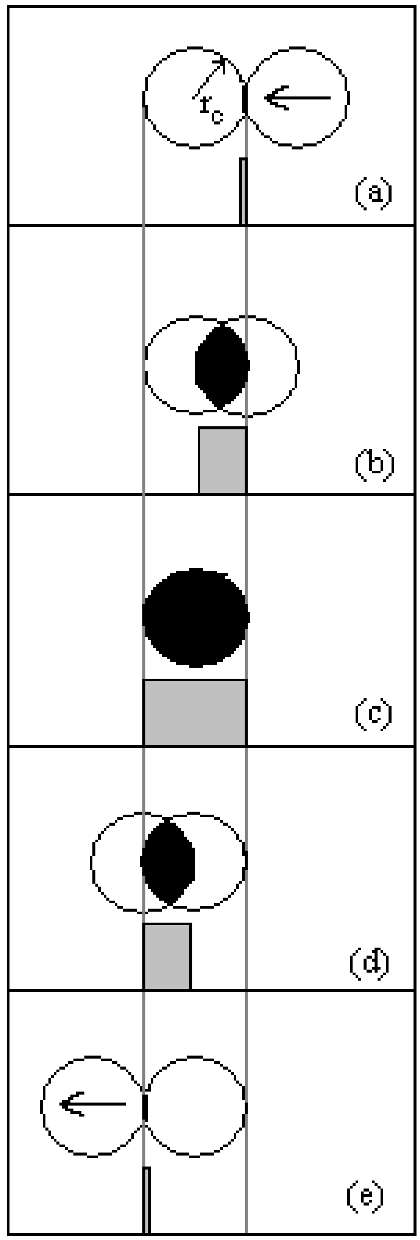


Figure 3-7. Moving pulse boundary condition: Heat flux (height of gray rectangles) as a function of the relative motion between asperity contact areas (Plan view – similar to Figures 3-1 and 3-4). The shaded area gives the total heat input to the contact area. The pulse can be compactly expressed as a function of both space and time dependent Heaviside functions. The two vertical gray lines "fix" the bottom contact area, while the top contact area moves relative to it from right to left. Use of this boundary condition would require a full 3D solution of the heat conduction equation.

3.4.3 Mathematical statement of the problem and its solution

3.4.3.1 *Background*

The temperature distribution for a single hemispherical asperity (Section 2.1) can be obtained using energy conservation for the hemispherical asperity in the spherical coordinate system (r, θ, ϕ). Spherical azimuthal symmetry is assumed (symmetrical in the ϕ direction about an axis passing through the centers of the two contacting asperities), as discussed in detail in Section 2.1. The assumptions were discussed in Section 1.4.2.1. The nonlinear 2-D transient heat conduction problem in r and θ can be stated as

$$\rho c_P \frac{\partial T(r,\theta,t)}{\partial t} = \frac{1}{r^2}\frac{\partial}{\partial r}\left(k(T).r^2\frac{\partial T}{\partial r}\right) + \frac{1}{r^2 Sin\theta}\frac{\partial}{\partial \theta}\left(k(T).Sin\theta\frac{\partial T}{\partial \theta}\right) \quad (3\text{-}19)$$

where k is the thermal conductivity of the asperity material, C_P, its specific heat, and ρ, its density. It must be noted that the domain of solution of the 2D problem domain is shifted 90^0 from the asperity cross-section, as discussed in assumption j of Section 2.1, and depicted in Figure 3-5. Due to 2D spherical symmetry, the problem can be solved in the cross-hatched domain of Figure 3-5(b), and then replicated in the other semicircle, to obtain the complete cross-sectional temperature distribution (for instance, see surface plots in Chapter 4).

Based on the assumptions of Section 1.4.2.1, the boundary conditions are:

$$\begin{aligned} \left.\frac{\partial T}{\partial r}\right|_{r=0} &= 0 = \left.\frac{\partial T}{\partial \theta}\right|_{\theta=0} = \left.\frac{\partial T}{\partial \theta}\right|_{\theta=\pi} \\ k(T)\left.\frac{\partial T}{\partial r}\right|_{r=R} &= q_f \quad [\text{ From Eq.}\,(3-18)] \end{aligned} \quad (3\text{-}20)$$

The initial condition for this problem is the ambient host rock temperature:

$$T_{initial}\,(r,\ \theta,\ 0) = T_0 \qquad 0 \le r \le R\ ,\ 0 \le \theta \le \pi \quad (3\text{-}21)$$

where T_0 is the ambient rock temperature in Kelvin, and R, the asperity radius.

3.4.3.2 *Solution Methods*

It must be kept in mind that the domain of solution for the 2D problem domain is shifted 90^0 from the asperity cross-section, as depicted in Figure 3-5(a). Due to 2D spherical symmetry, the problem can be solved in the cross-hatched domain shown in that figure. The results can then be replicated in its complementary semicircle (non-cross-hatched part of the domain in Figure 3-

5(a)), to obtain the complete cross-sectional temperature distribution (see temperature surface plots in Chapter 4).

Analytical Solution: Only an outline of this procedure is given as an analytical solution has limited applicability to the problem being discussed (For details, see Strauss 1992 or Asmar 2000). A few generalizations can be made, however. It is a (mathematical) property of any solution of the heat diffusion equation that its maximum (or minimum) value is attained either at the boundaries of the problem domain or at the initial time. This is called the ***Maximum Principle***. For the conditions of this problem, the maximum temperature can be expected to occur around the heat source (i.e., on the contact surface and/ or at time $t=0$). This temperature can be used to determine whether there will be any melting of the asperities. A similar procedure was used by Cardwell, et al. (1978) and McKinzie and Brune (1972) to analyze melt zones in faults with "planar slips". If the maximum temperature exceeds the melting temperature of the gouge or asperity, then partial melting can be expected to occur.

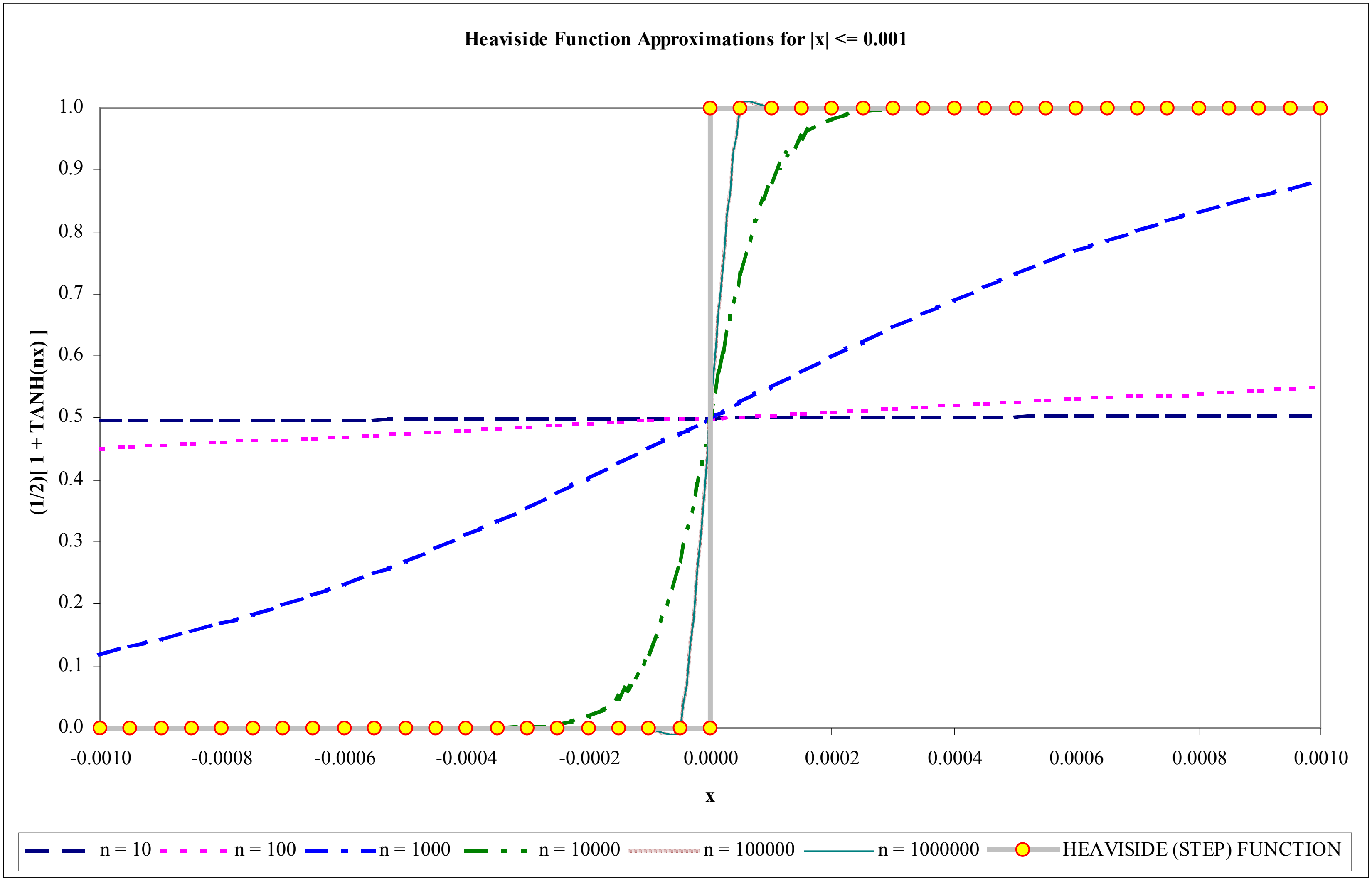
An analytical series solution was attempted first, using the separation of variables technique. In order to do that, a transformation of variables has to be applied, in order to make the boundary conditions (3-20) homogeneous. The series solution to this transformed equation is then expressed in the form of spherical Bessel functions and Legendre functions (Eigenfunction expansion). The transformed equation contains a "source term" (a term on the RHS of equation (3-19), in addition to the standard first and second partial derivative terms that appear there. Therefore, the coefficients have to be determined by solving a system of ODEs in time, whose dependent variables are the coefficients. FORTRAN 90 codes were written to compute these coefficients to any user defined accuracy (up to machine limit). Due to the extremely non-smooth boundary conditions, however, the "Fourier" coefficients are highly oscillatory and decayed very slowly with an increase in the number of terms. In the end, time and system resource constraints made it impossible to compute the analytical solution.

Numerical Solution: A very detailed explanation of the procedure used here is presented in Appendix A, and the code appears in Appendix B. A brief outline is provided here for the sake of completeness. Before outlining the problem handled in the actual code, it should be noted that the Heaviside functions (defined just below Equation 3-18) used in the boundary conditions have to be approximated for numerical computation. The sharper these functions (i.e., the closer these functions are to a step function), the steeper the gradients at the boundary itself. As the boundary becomes steeper, we run into resolution problems (Appendix A). One way of approximating the Heaviside function is

$$H(x-a) = (1/2)*[\ 1 + TANH\{n(x-a)\}\ ] \qquad (3\text{-}22)$$

The larger the value of n, the sharper the step function (Figure 3-8). All approximations are plotted as various types of lines, while the actual Step Function is displayed as dotted data. As will be seen from the results in the next section, the typical time and length scales of this problem are less than 0.001 (seconds and meters, respectively). So, a good approximation for n will have to be ≥ 100,000. From Figure 3-8 we can see that the higher this value, the better the approximation, and the steeper the gradient at $x = 0$. Details regarding the actual n-value chosen for the results presented in Chapter 4 is presented in Section 4.1.

Figure 3-8. Illustrating of the effect of *n* on the Heaviside function approximation given by Equation (3-22). The Heaviside Step Function itself is plotted using circular data points, for clarity.



A FORTRAN 90 code was developed to solve a very general problem: non-linear, transient, pure conduction in 2 dimensions, in the variable u, with the self-adjoint form

$$\left\{a_1(x,y,t).\frac{\partial}{\partial x}\left(a_2(x,y,t).k_t(u).\frac{\partial}{\partial x}\right)+b_1(x,y,t).\frac{\partial}{\partial x}\left(b_2(x,y,t).k_t(u).\frac{\partial}{\partial y}\right)\right\}(u)+f(u,x,y,t)=\rho_0 c_P(u)\frac{\partial u}{\partial t} \qquad (3\text{-}23)$$

This can be compactly written in terms of the non-linear functional, N, as

$$\frac{\partial u}{\partial t}=N_1(u,u_x,u_y,u_{xx},u_{yy})+\left(\frac{f(u,x,y,t)}{\rho_0 c_P(u)}\right)=N(u,u_x,u_y,u_{xx},u_{yy}) \qquad (3\text{-}24)$$

with the general non-linear boundary conditions:

$L(u,u_x) = f_L(y,t)$ (3-25a)
$R(u,u_x) = f_R(y,t)$ (3-25b)
$B(u,u_y) = f_B(x,t)$ (3-25c)
$Tu,u_y) = f_T(x,,t)$ (3-25d)

where, L, R, B, and T represent the left, right, bottom, and top (non-linear) boundary functionals. For most standard heat conduction applications, each of the above functionals further take the generalized Robin form

$F(u,u_{xi}) = F_1(u)\ .\ u_{xi} + F_2(u)$ (3-26)

where i = 1 or 2 (corresponding to the two principal problem coordinates, x and y). The same code can be used to compute numerical solutions to corresponding linear problems. The code can be used to solve problems in any of the three "standard" geometries, cartesian, cylindrical and spherical, without any modification to its core routines. Of course, problem setup is very elaborate. This is described in detail in Appendix A. The next Chapter provides a summary and discussion of results, as well as conclusions based on the research conducted here.

4.0 RESULTS AND DISCUSSION

4.1 *Summary of Model Runs*

A summary of the model runs is presented in Table 4-1. Table 4-2 presents a summary of the model parameters. As discussed in Logan and Teufel (1986) and Sammis et. al (1999), small asperities (of the order of millimeters) may be subject to compressive stresses comparable to their ultimate compressive strength. This is reasonable since small asperities are less likely to have zones of weakness. Experimental confirmation of this result was compiled extensively in Touloukien et. al., (1981). The strength of the small asperities increases (theoretically up to the ultimate compressive strength of the material), with decreasing asperity size. Here, we consider asperities of sizes 1 mm to 10 cm. Since we are only interested in the influence of shear stress on the temperature distribution generated by frictional heating, we do not attempt to predict or estimate the stresses for specific scenarios. Therefore, for each asperity size, a range of shear stresses was used. These ranges varied from 10-100 MPa (narrowest range, for a 1 cm asperity) to 10-1000 MPa (widest range, for a 1 mm asperity). Larger asperities were assumed to experience a narrower range of shear stresses due to their larger contact areas. Since pseudotachylytes (PT) are common in granitic rock, quartz and feldspar were used as typical asperity materials.

Run Resolutions: In Table 4-1, each case was run for at least four resolution levels , or until the convergence rate predicted in Appendix A (numerical methodology) was obtained. This sometimes required going up to five or six resolution levels. Each resolution level increase corresponds to a halving of each of the two space steps and a halving of the time step. This results in an overall increase in resolution of 8 times. Correspondingly, the number of computations, and the run duration increase roughly 8 times with each increase in resolution level. In some cases, optimal convergence was not achieved even at levels 5 or 6. Time constraints did not permit running at even higher resolutions.

Step function approximation: In addition to the resolution level for the problem domain, the resolution of the step function approximation for the boundary condition (Equations 3-18 and 3-22, Figure 3-7) is also important. The effect of n is further illustrated schematically in Figure 4-1. The larger the value of n, the smaller the dispersion outside the (contact duration or contact area boundaries, respectively, for time and length scales) shown in Figure 4-1. The effect of this is that the higher values of n resulted in larger temperature maxima (as much of the energy that lay outside the ”contact rectangle” is now “concentrated” within it; see Figure 4-1). Also, the larger the value of n, the higher the resolution required for solutions to converge, and therefore, the larger the run times. Intuitively, maximum temperature is expected to occur at the time of asperity separation. A value of $n = 100,000$ was found to be sufficient for convergence of the maximum temperature times to the asperity separation times. From the foregoing discussion, the values obtained for $n = 100,000$ are actually lower-bounds on the “actual” maximum temperatures, but do not differ from them by more than the problem uncertainty range. For this study, this is a reasonable criterion since the exact values of peak temperature are not as critical as their order of magnitude.

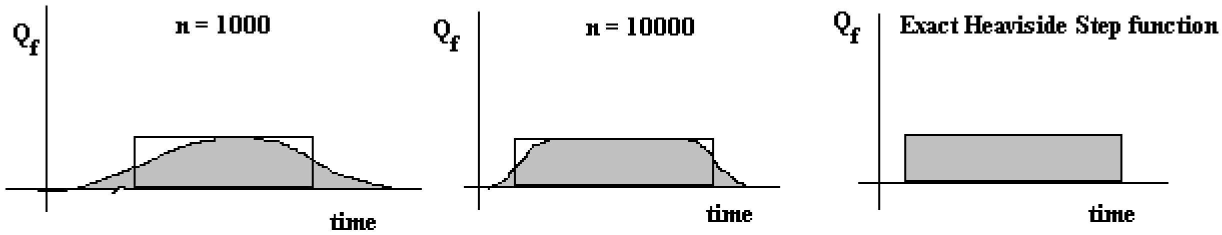


Figure 4-1. Effect of the parameter n on the "sharpness" of the temporal Heaviside function used in Equation 3-18. As n gets larger, the TanH approximation (shaded profile) contains more of the heat input within the time of contact duration (represented by the transparent rectangle). This results in slightly higher maximum temperatures.

Table 4-1. Run Summary: About 330 runs were carried out, covering 75 different cases.

Quartz, Nonlinear Runs (27 cases):		
Asperity Radius (mm)	**Shear Stresses (Mpa)**	**Resolution Levels.**
1	10, 50, 100, 200, 500, 1000	1-4 (5 or 6 at high Shear Stresses)
5	10, 50, 100, 200, 500	-do-
10	10, 50, 100, 200, 500	-do-
50	10, 50, 100	-do-
100	10, 50, 100	-do-
Depth tests: 1 km and 2 km		
1	500, 1000	-do-
10	100, 200	-do-
Slip Velocity test: $V_s^* = V_s/2$	100	1-4
Feldspar, Nonlinear Runs (26 cases)		
Asperity Radius (mm)	**Shear Stresses (Mpa)**	**Resolution Levels.**
1	10, 50, 100, 200, 500, 1000	1-4 (5 or 6 at high Shear Stresses)
5	10, 50, 100, 200, 500	-do-
10	10, 50, 100, 200, 500	-do-
50	10, 50, 100	-do-
100	10, 50, 100	-do-
Depth tests: 1 km and 2 km		
1	500, 1000	-do-
10	100, 200	-do-
Quartz, Linear Runs (10 cases)		
Asperity Radius (mm)	**Shear Stresses (Mpa)**	**Resolution Levels.**
1	10, 100, 500, 1000	1-4 (5 at high Shear Stresses)
10	10, 100, 500	-do-
100	10, 50, 100	-do-
Feldspar, Linear Runs (10 Runs)		
Asperity Radius (mm)	**Shear Stresses (Mpa)**	**Resolution Levels.**
1	10, 100, 500, 1000	1-4 (5 at high Shear Stresses)
10	10, 100, 500	-do-
100	10, 50, 100	-do-

Table 4-2: Fault and material parameters used in the runs.

Material Property	Quartz	Feldspar
Poisson's Ratio, ν	0.2	0.3
Young's Modulus, E	94 GPa	40 GPa
Density, ρ	2650	2620
Thermal Conductivity	Linear case: 4.3 $W.m^{-1}.K^{-1}$ Nonlinear case: See Appendix C	Linear case: 1.35 $W.m^{-1}.K^{-1}$ Nonlinear case: See Appendix C
Specific Heat	Linear case: 1123 $J.kg^{-1}.K^{-1}$ Nonlinear case: See Appendix C	Linear case: 767 $J.kg^{-1}.K^{-1}$ Nonlinear case: See Appendix C
Melting Point	2050^0 K	1500^0 K
Fault Property	**Value**	
Coefficient of friction, μ	0.6	
Relative slip velocity, V_{slip}	1 m/s (except as noted in Table 4-1 above)	
Shear Stress	See Table 4-1 above	

A total of ~330 runs were carried out for the roughly 75 cases mentioned in Table 4-1. Further details on convergence are presented in Section 4.2. The output from the FORTRAN 90 code, *COND2D* (Appendix B), was processed using codes written in MATLAB (Appendix D) and MS-Excel. Plots of thermal properties as a function of temperature are presented in Appendix C.

4.2 Convergence of solutions.

To visually check on convergence, the MATLAB codes *DevolRuns.m, ConvTestPlots.m,* and *DsnapRuns.m* (Appendix D) were written to generate several types of convergence plots for every one of the 73 cases presented in Table 4-1. For illustrative purposes, one set of plots is presented below. Figure 4-2 presents the temporal evolution of global maximum temperature (which occurs at the right boundary). As discussed in Appendix A, the steep gradient resulting from a large boundary shear stress necessitates the use of very high spatial resolutions to obtain convergence. This results in significant run times (typically 24 hours or longer per run). To achieve convergence, and still complete the runs in a reasonable time, use is made of a specific characteristic of the solutions to the problem posed here. Namely, due to the very small thermal diffusivities (~ 10^{-6} m^2/s) of the minerals modeled here, a localized temperature pulse generated over a very short contact time at the boundary dissipates very close to the boundary. These can be seen in the convergence plots of Figures 4-2 and 4-3. Therefore, much of the asperity area (problem domain area) does not influence the problem solution.

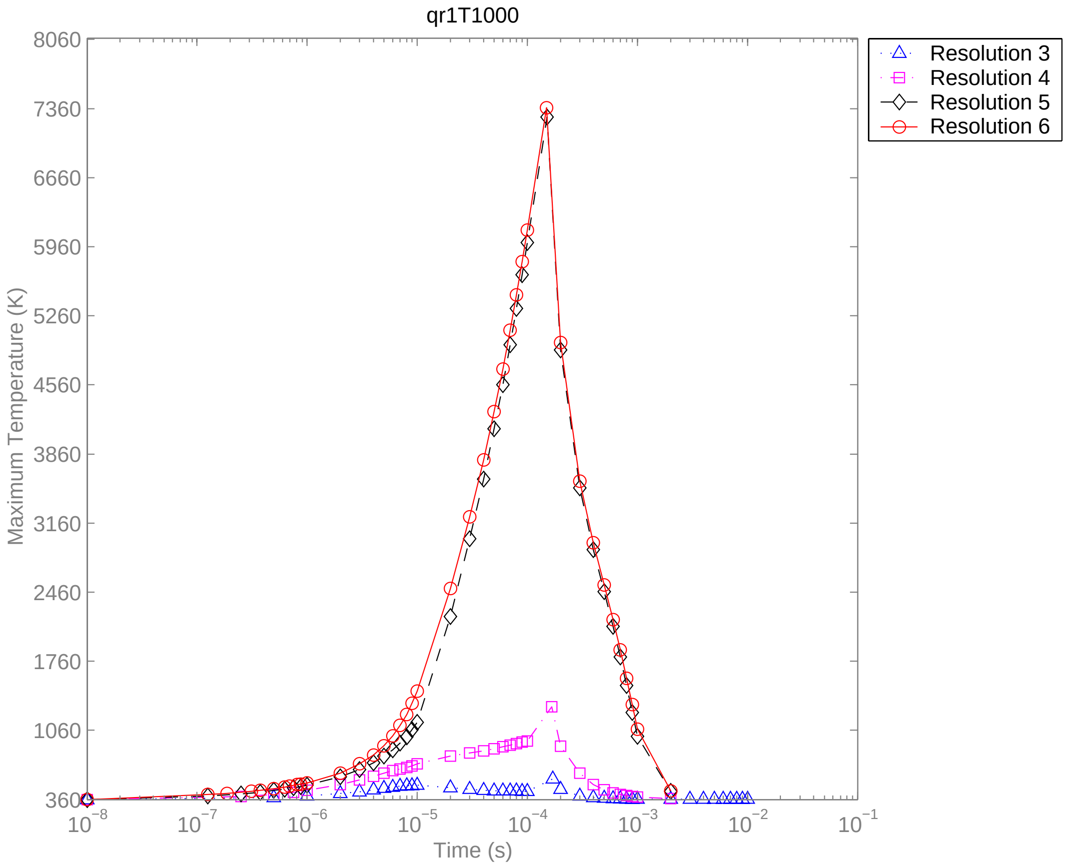


Figure 4-2. Demonstration of convergence of solution as a function of increasing resolution. The code *QR1T1000* denotes a quartz asperity of 1 mm radius experiencing a boundary shear stress of 1000 MPa (1 Gpa).

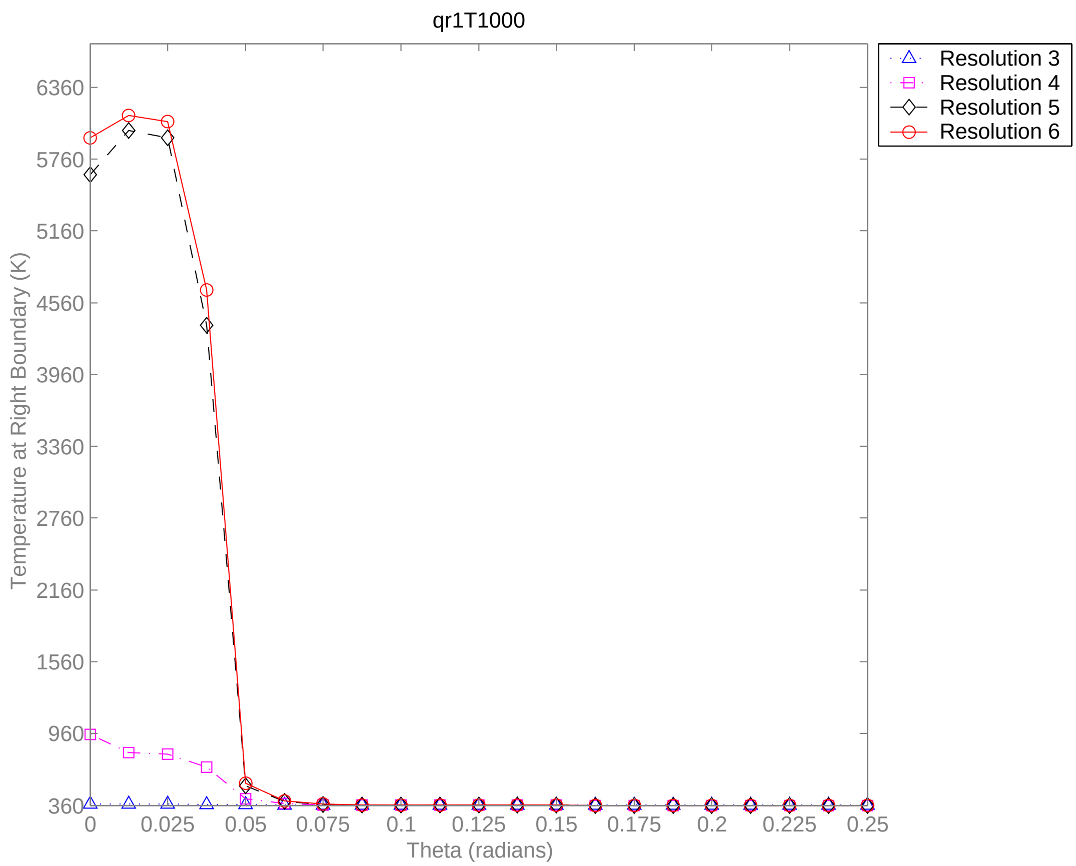
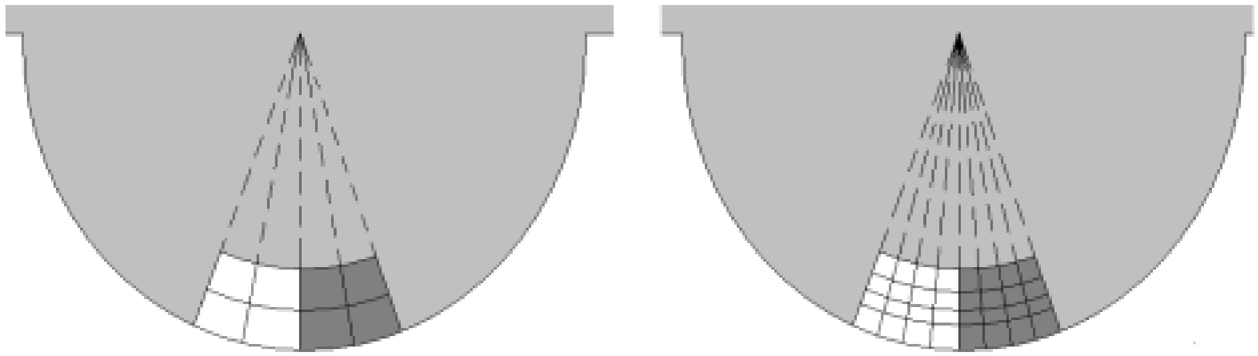


Figure 4-3. Demonstration of the effect of resolution on the base of the temperature pulse. As the resolution increases, the pulse is "drawn inward", thus reducing its far-field effect. As the resolution increases from 3-6, the extent of the x-axis experiencing ambient temperatures remains nearly unchanged. The data shown here are for a quartz asperity of 1 mm radius experiencing a shear stress of 1000 MPa.

Significant time savings can be obtained if the problem domain were to be cropped to as small a value as practical. For the numerical method adopted here (Douglas-Gunn time splitting, Appendix A), the decrease in run time is directly proportional to the reduction in area achieved from "domain cropping". While successively reducing the domain size, all three flux boundary conditions, located within the body of the fault [Equations (3-20)] must still be satisfied to within the limits of the uncertainty in temperature due to parameter uncertainties. Cropping also allows a concomitant increase in resolution, because the problem domain is much smaller. Typical cropped area for the asperity being considered is shown for two resolution levels in Figure 4-4. For all the cases specified in Table 4-1, a cropped area was iteratively obtained from a low resolution (fast) run, such that the temperatures at the domain boundaries were less than 1% of the peak temperature at that resolution.



(a) **(b)**

Figure 4-4. Cropping the problem domain: The area in white is the domain for which the Fortran 90 code, *COND2D*, was run. The dark gray area has temperatures that are a mirror image of the white area, about their common boundary. The resolution level for (b) is one higher than (a), having nearly twice the grid points as the latter.

The cropping process described above can be justified by looking at a snapshot of the temperature values at the asperity surface in the region of its contact area (Figure 4-3). Based on several such runs, it was observed that:

(a) Compared to those in the vicinity of the peak itself, grid nodes far from the peak of the temperature pulse (Figure 4-3) are not as sensitive to resolution increases. This is a consequence of the low thermal diffusivities mentioned above.

(b) The area occupied by the "base" of the temperature pulse (x-axis in Figure 4-3) remains nearly constant with changes in resolution. In many cases it actually gets slightly smaller at higher resolutions (since it is better resolved), thus "drawing" in the temperature perturbation, and slightly reducing its far-field influence.

Therefore, using a lower resolution run to iteratively determine this "minimum" area is reasonable. This will become clearer in Section 4.1.3.1, where 3D temperature surface plots for the cropped domain are shown at specific times. In a number of cases, although the theoretical (2^{nd} order) convergence rate is not achieved for the range of resolutions attempted (limited due to the time constraints on this project), the plots indicate convergence to within 10° K (and more commonly to within about 1° K), which is probably within the parameter uncertainty range for this model.

4.3 *Temperature Distribution - Nonlinear runs*

4.3.1 Temperature Surface Plots and area of potential melting

As discussed in the previous section, the problem proposed here is solved on only a small area of the original problem domain. The figures illustrated in this section represent a "zoom" of the asperity domain adjacent to the contact area/heat generation zone. Figures 4-5 to 4-7 depict the surface temperature. Each are color coded magnitude plots for the relevant sub-domains at each of four different times. Figure 4-5 is a nonlinear run for a feldspar asperity, and Figures 4-6 and 4-7 are nonlinear runs for quartz asperities. In each figure, the yellow end of the color bar is scaled to the melting temperature of the corresponding mineral in °K (Table 4-2). It must be noted that the fraction of asperity area represented by the sub domain in Figures 4-4 to 4-7 can be calculated from

$$f_A = \frac{\theta_0}{\pi} \cdot \left[1 - \left(\frac{r_i}{r_0} \right)^2 \right] \tag{4-1}$$

where, f_A is the fraction represented by the sub domain area, θ_0 is the angle subtended by the sub-domain, A_{sb}, at the geometrical center of the hemispherical asperity, r_i is the inner radius of A_{sb}, and r_0 is the asperity radius (or outer radius of the sub-domain). Typical values were $\theta_0 = 10^{-2}$ to 10^{-1} radians and (r_i/r_0) = 80-99%. The largest value of f_A, ~ 1%, corresponds to the maximum θ_0 and the minimum (r_i/r_0). This value is for the smallest asperities (1 mm radius), as may be intuitively expected. The area occupied by the pulse, the yellow region, can be computed from the area of the base of the pulse in the above figures. This pulse area is only a fraction of this sub-domain area. A typical value for this fraction is 3-5%, with a maximum of ~10%. So, at best only 0.1% of the smallest asperities can melt during any <u>single</u> asperity encounter.

<u>Melting - Quartz vs. feldspar:</u> To compare the results for quartz and feldspar, the following must be noted: the thermal conductivity for feldspar increases with increasing temperature, up to its melting temperature and is then assumed to decrease (Figure C-2). At its maximum, it is ~ 30% of the maximum quartz conductivity (at ambient temperature). The specific heat of both minerals increases with increasing temperature (Figures C-3 and C-4). The specific heat of feldspar is less than that of quartz over the range of temperatures depicted in the above figures. This means that the thermal diffusivity of quartz near its melting temperature of ~ 2050° K is much smaller than that for feldspar near its melting temperature (~1500 °K). Therefore, all else being equal, we would expect the temperature maxima produced for quartz asperities to be much larger and more spatially restricted than that for feldspar, near their melting points. This implies more melting for feldspar asperities, even though quartz asperities have the potential to produce much higher temperatures. This can be observed by comparing Figures 4-5 (feldspar) and 4-6 (quartz), which are for the same asperity sizes and boundary shear stresses. Results suggest that for feldspar, the area around the temperature pulse that is perturbed by it is larger, the closer the surrounding temperature approaches to the melting temperature. Given asperities of the same size, relative to quartz asperities, feldspar asperities are more likely to experience melting at

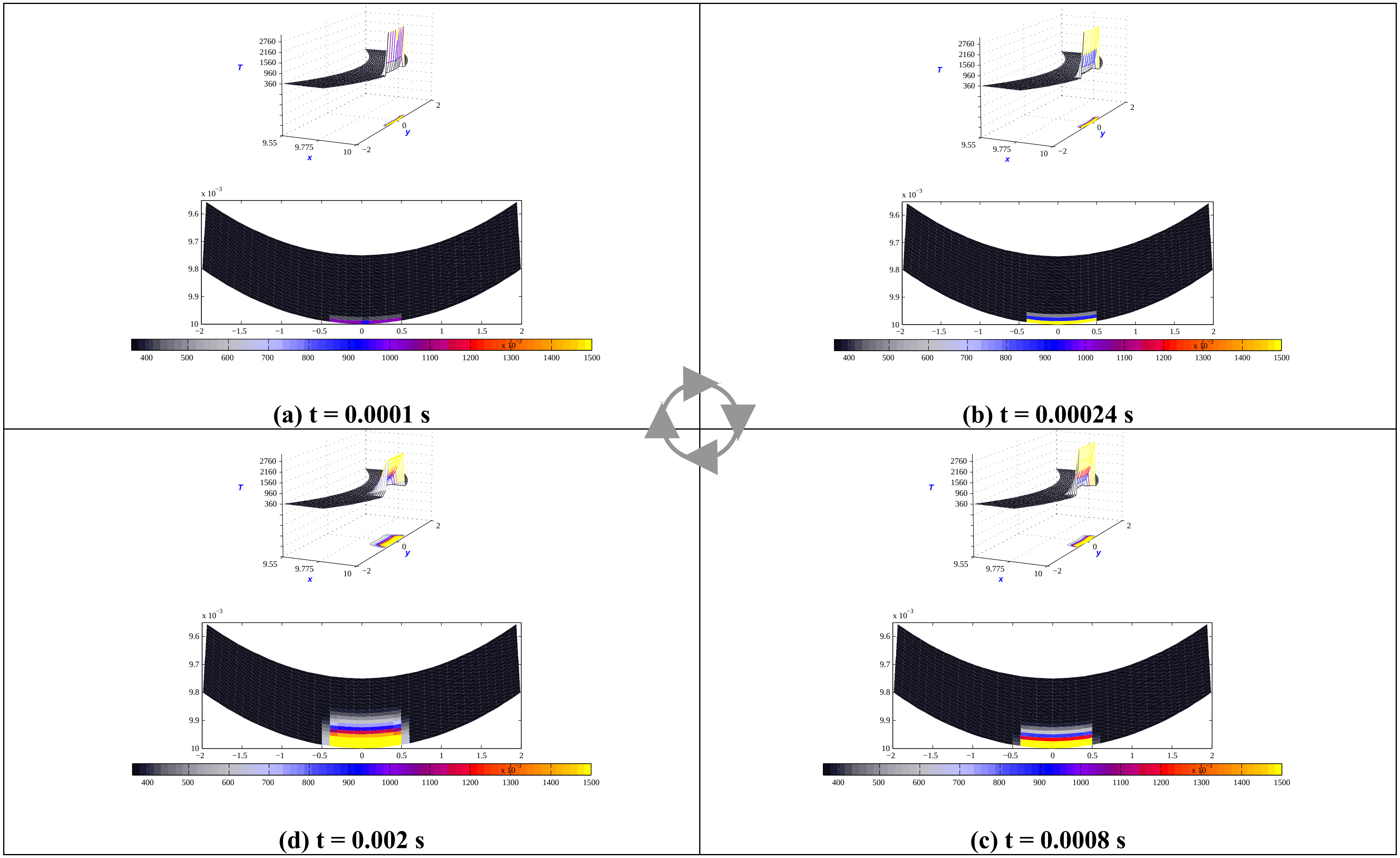


Figure 4-5. Surface temperature plots for the NONLINEAR run: *FR10T500* (10 mm feldspar asperity with 500 MPa boundary shear stress). The color bar scales from black (360° K) through grays, blues, reds, and finally, yellows (1500° K, the melting point for feldspar). <u>Axes RANGE</u>: X = 9.6 to 10 mm; Y = -2 to 2 mm. Compare with Figure 4-4.

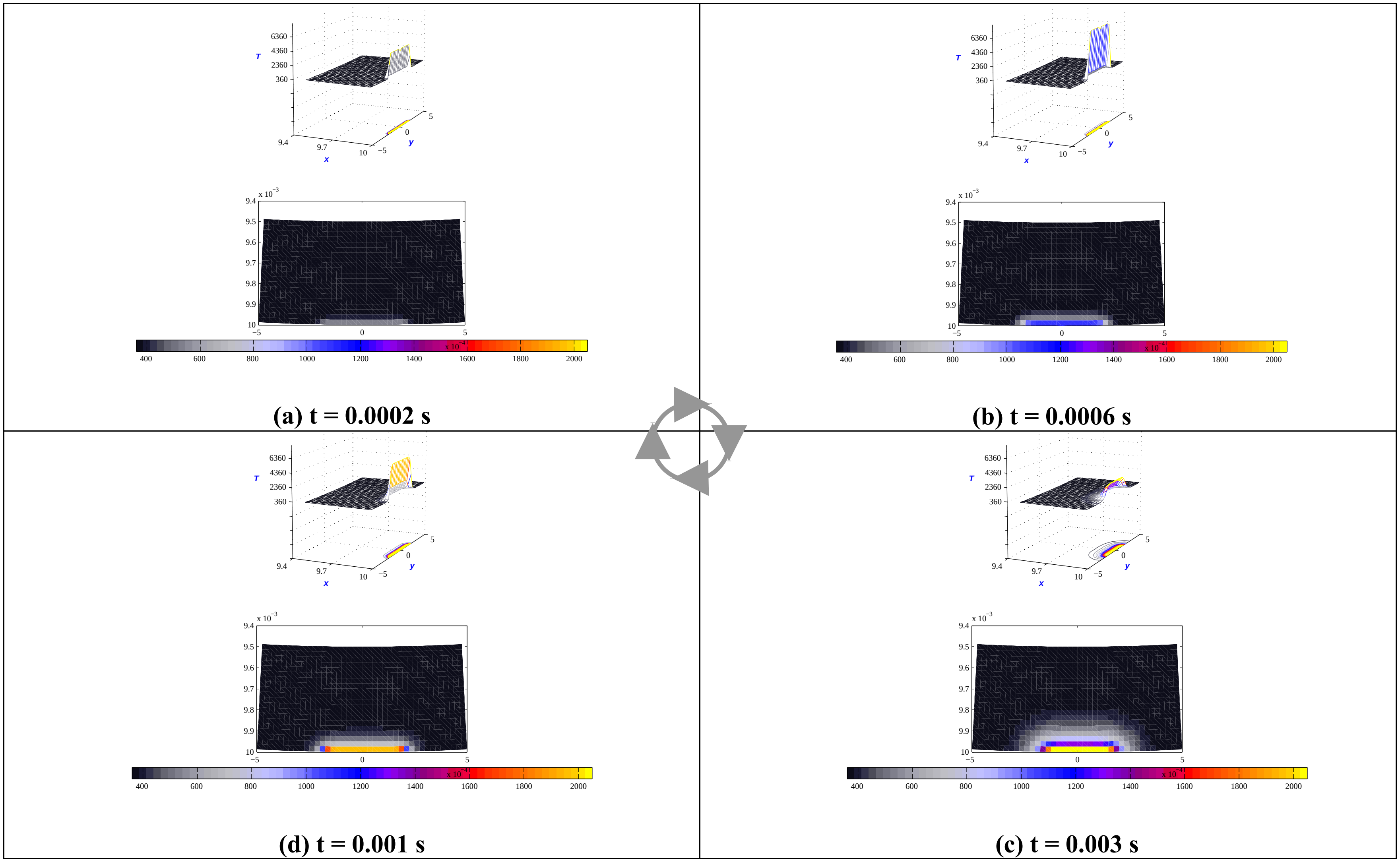


Figure 4-6. Surface temperature plots for the NONLINEAR run: Q*R10T500* (10 mm quartz asperity with 500 MPa boundary shear stress). The color bar scales from black (360° K) through grays, blues, reds, and finally, yellows (2050° K, the melting point for quartz). Axes RANGE: X = 9.4 to 10 mm; Y = -0.5 to 0.5 mm. Compare with Figure 4-4.

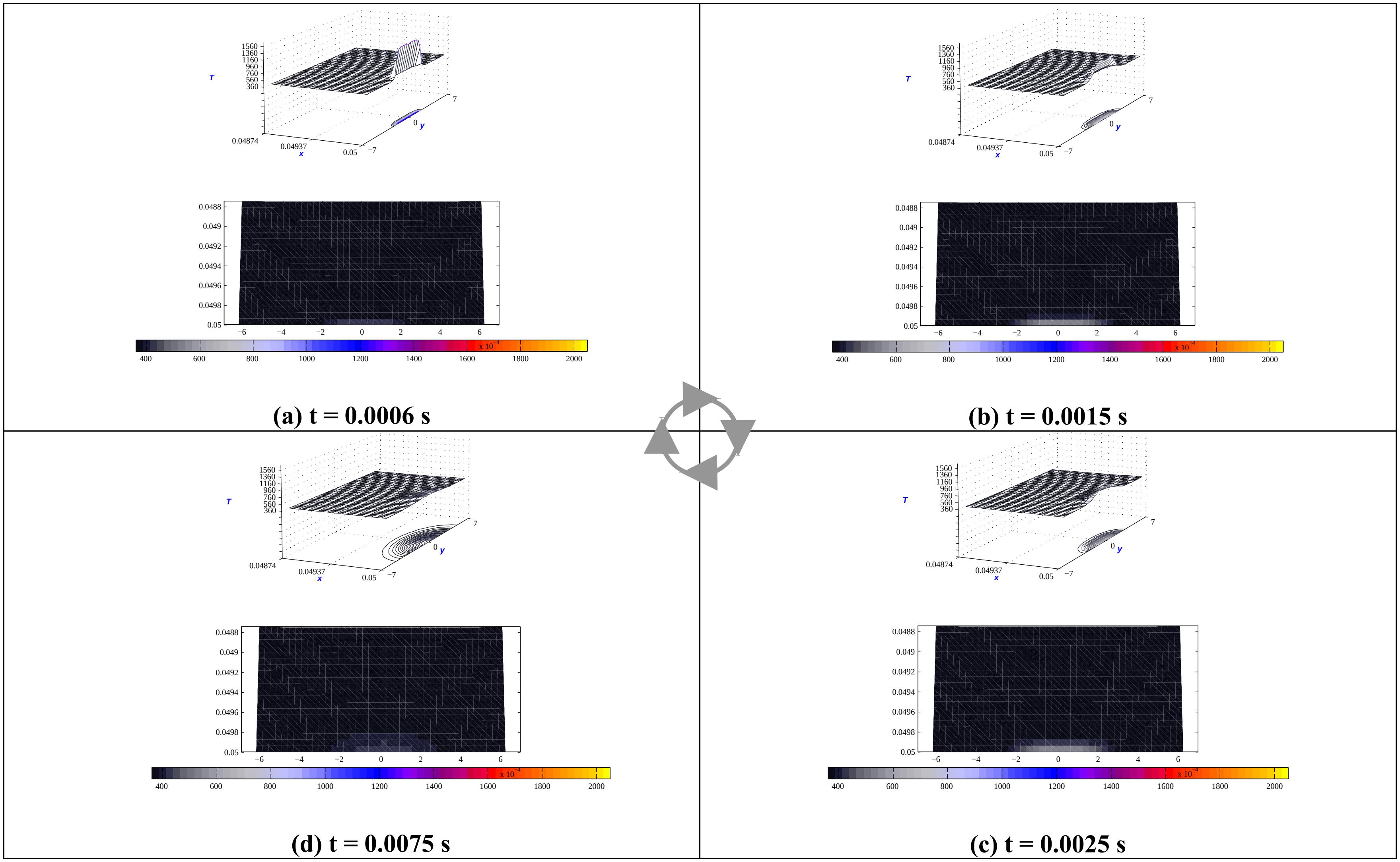


Figure 4-7. Surface temperature plots for the NONLINEAR run: Q*R50T100* (50 mm quartz asperity with 100 MPa boundary shear stress). The color bar scales from black (360° K) through grays, blues, reds, and finally, yellows (2050° K, the melting point for quartz). Axes RANGE: X = 48.74 to 50 mm; Y = -0.7 to 0.7 mm. Compare with Figure 4-4.

lower shear stresses. This agrees with observations from field samples – pseudotachylyte matrix is made up of melts derived from feldspars and micas, with embedded quartz clasts.

Diffusion length scales: The characteristic linear 1D diffusion length is defined as

$$L_{1D} = \sqrt{(\kappa t)} \qquad (4\text{-}2)$$

where κ is the material thermal diffusivity, and t, is the time scale of interest. For feldspar (linear case: k *(mean)* = 1.5 $Wm^{-1}K^{-1}$, ρ = 2620 kg/m^3, C_P *(mean)* = 767 $Jkg^{-1}K^{-1}$; κ = 6.7 x 10^{-7} m^2/s) at time, t = 0.002 s, $L_{1D,feldspar}$ ~3.66 x 10^{-5} m. In comparison, the characteristic penetration depths of the temperature pulses (non-black regions) for the nonlinear feldspar model run presented in Figure 4-5 is ~2.75 x 10^{-4} m (t = 0.002 s). Similar comparisons suggest that the nonlinear penetration depths for feldspar are as much as an order of magnitude greater than the linear predictions for high shear stresses (Figure 4-6), , and at least twice the linear predictions for lower shear stresses. For quartz (linear case: k *(mean)* = 3.3 $Wm^{-1}K^{-1}$, ρ = 2650 kg/m^3, C_P *(mean)* = 1123 $Jkg^{-1}K^{-1}$; κ = 1.2 x 10^{-6} m^2/s) at time, t = 0.003 s, $L_{1D,quartz}$ ~5.98 x 10^{-5} m; at time , t = 0.0075 s, $L_{1D,quartz}$ ~1.1 x 10^{-4} m. In comparison, the penetration depths for the two nonlinear quartz models presented in Figures 4-6 and 4-7 are 2.8 x 10^{-4} m (t = 0.003 s) and 2 x 10^{-4} m (t = 0.0075 s), respectively, for the identical time scales. Similar comparisons suggest that the nonlinear penetration depths for quartz are ~2 to 4 times greater than the linear 1-D predictions (larger deviation for higher shear stresses, Figures 4-6 and 4-7).

In general, higher shear stresses lead to much larger temperature pulses and larger boundary thermal gradients compared to scenarios with lower shear stresses (due to the cubic relationship described in the next section). For feldspar, higher temperatures lead to larger thermal conductivities (Figure C-2), and hence, larger penetration depths compared to quartz. This is corroborated by the penetration depths obtained above from Figures 4-5 and 4-6. It should be noted that although specific heat increases with temperature, its fractional change is much smaller for both minerals (Figures C-3 and C-4). So, the larger fractional change in thermal conductivity influences thermal diffusivity more strongly than specific heat. For quartz asperities, small temperature pulses diffuse farther into the asperity (Figure 4-7) due to higher thermal conductivities and lower specific heats at lower temperatures (Figure C-1). The opposite happens for large temperature pulses (which typically occur at high stresses). Since conductivities are lower and specific heats are higher, the temperature pulse is more concentrated (Figure 4-6). Since the thermal conductivity is a maximum close to feldspar's melting point, feldspar asperities, the pulse penetration depth is larger, the closer its magnitude is to the melting point, as indicated in Figure 4-5. Figures 4-5 to 4-7 seem to imply that in the lateral (θ) direction, both the linear and non-linear cases show diffusion lengths that are an order of magnitude larger. This result is, however, an artifact that arises because much of the circumferential extant of the heat pulse corresponds to the actual asperity contact area (or heat generation zone).

On a real fault, each asperity may encounter a number of opposing asperities (depending on asperity size distribution on the fault surfaces), before it gets abraded or melted away. This repetitive process potentially produces much more melt than predicted by this model.

The observation that the temperature pulse remains and dissipates locally helps to justify the assumption of a fully spherical geometry (Section 3.4.1, Figure 3-5) for a hemispherical asperity, which includes part of the fault rock. In addition, unless there are repeated asperity encounters (when repeated temperature pulses at the boundary can potentially melt significant quantities the asperity), inter-asperity interaction can be safely ignored for the time scales of individual asperity interactions. The above discussion provides one explanation for the rarity of pseudotachylytes – namely, that melting is so hard to initiate.

4.3.2 Peak Temperatures

Figures 4-8 depicts peak temperatures obtained for all the nonlinear quartz models as a function of shear stresses, for different asperity radii. These figures also show the best fit trendlines to the data. Before discussing the graph, it is illustrative to see how these two parameters affect temperature distribution in an asperity. The temperature rise ultimately depends on the total heat input into the system. For the 2D problem, this heat input, q_f, is given by:

$$q_f = \tau . r_c . V_{slip} . t_0 \qquad (4\text{-}3)$$

where τ is the boundary shear stress, V_{slip} is the relative slip velocity between opposing asperities, and t_0 is the asperity contact duration. The contact duration is given by

$$t_0 = 2d_c / V_{slip} := 4r_c / V_{slip}: \qquad (3\text{-}16)$$

Substituting (3-16) into (4-3) gives

$$q_f \propto \tau . r_c^2 \qquad (4\text{-}4)$$

r_c, the radius of the asperity contact area, can be obtained from the Hertzian solution [Equation (3-13)]

$$r_c = \frac{3\pi\sigma_n}{4}\left(\frac{1-\nu^2}{E}\right)R \propto \tau R \qquad (4\text{-}5)$$

where R is the asperity radius and the normal stress has been represented in terms of the shear stress and coefficient of friction in the proportionality. Based on this result, the total heat input to the system is given by:

$$q_f \propto \tau^3 . R^2 \qquad (4\text{-}6)$$

Based on Equation (4-6), we would expect the temperature in the asperity to increase as the <u>square</u> of the asperity radius, and as the <u>cube</u> of the boundary shear stress. This behavior is observed in Figure 4-8, which can be used as an independent validation for the code (more mathematically rigorous validation tests are presented in Appendix A). Since the coefficients are different for each fit, however, a power law fit may be more appropriate.

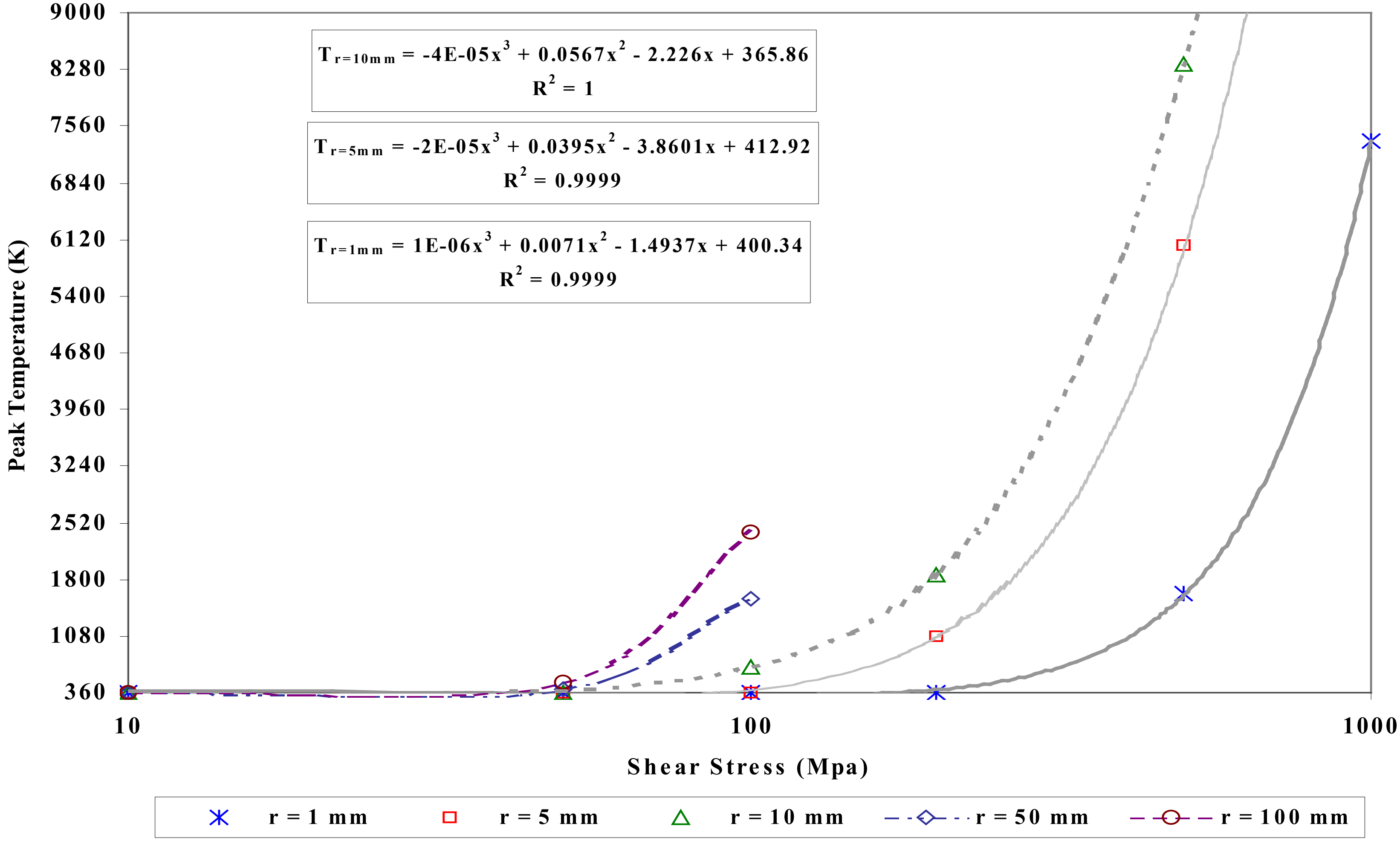
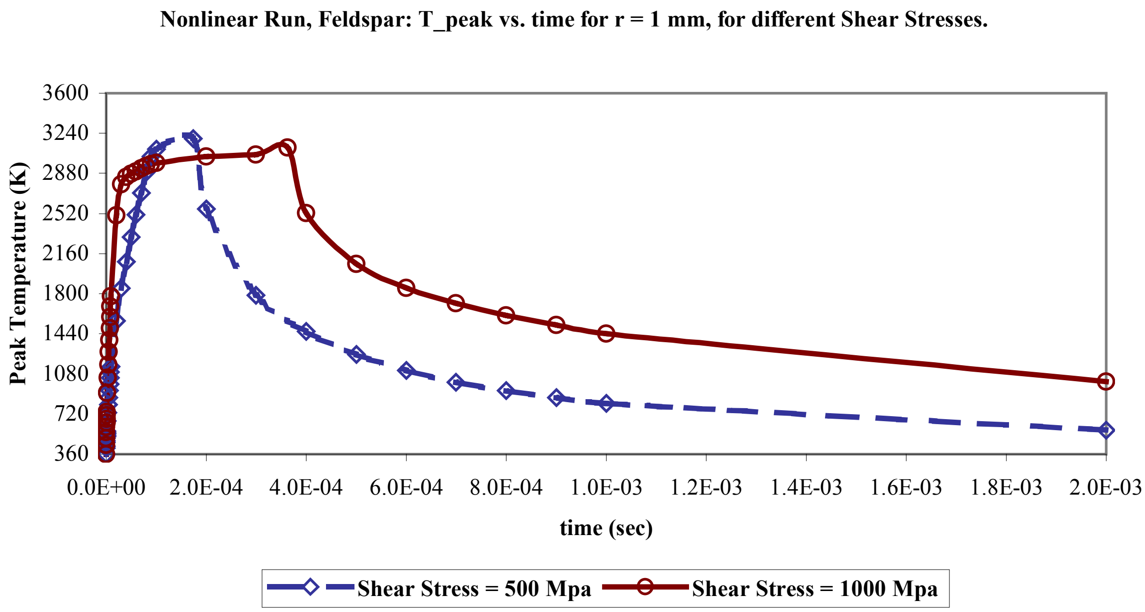


Figure 4-8. Peak temperatures for quartz (nonlinear runs) as a function of shear stress, for different asperity radii. Where sufficient data points were available, the best fit trendlines (cubic polynomials) fit the data perfectly, in agreement with Equation 4-6.

4.3.3 Temperature evolution profiles

Figure 4-9 presents sample temperature evolution profiles for nonlinear feldspar runs at different shear stresses. Figure 4-10 presents sample temperature evolution profiles for nonlinear quartz runs for two combinations of asperity size and shear stress. The curves shown in these two figures share certain similarities.. Each curve has a rapid temperature rise phase (phase 1), and a slow dissipation phase (phase 2). The rate of temperature increase in phase 1 is limited by the rate of work done. Time, $t = 0$, corresponds to the start of asperity contact. Asperity separation time is denoted by the time at which maximum temperature occurs. As dissipation progresses post asperity separation, the driving thermal gradient rapidly decreases, eventually leading to an asymptotic decay of the temperature (similar to exponential decay). In general, the higher the temperature attained, the faster the initial decay in phase 2. In consequence, the temperature pulses get sharper and more pointed as the magnitude of maximum temperature attained increases. Lower temperatures generate a broader profile. However, comparing Figures 4-9 and 4-10, it can be seen that the "temperature plateau" observed for feldspar asperities at high shear stresses are absent in quartz at high stresses, for the same asperity sizes. This can be attributed to two characteristics of feldspar: (1) the contact durations for feldspar are longer because of its lower Young's modulus, which leads to a larger contact area, and (2) the conductivity of feldspar increases with temperature and does not decrease much from its peak value (Figure C-2) due to the assumed quadratic profile. Therefore, once a certain high temperature is reached (~ 3000 °K, Figure 4-9), any further heat input is conducted away due to the high conductivity at that temperature. The process is self-propagating as long as the heat source exists since conductivity does not change much for feldspar in the range 1500 – 3000 °K. In contrast, the conductivity of quartz decreases dramatically with temperature, and owing to a high Young's modulus typical quartz contact areas are half that of feldspar asperity contact areas (all else being equal). Therefore, no such "conduction plateau" is observed (Figure 4-10). As discussed in the previous section, peak temperatures are usually attained for intermediate asperity sizes, for large shear stresses. Since the contact duration increases with both shear stress and asperity size, the time of attainment of this peak temperature increases if either one, or both parameters increase.

Effect of slip velocity: In Equation (4-4) above, slip velocity cancels out of the heat flux boundary condition for the definition of individual asperity encounters. So, for linear problems, it is reasonable to assume that slip velocity has no effect on temperature maxima. However, a slower velocity will stretch the temperature evolution profile (like those shown in Figures 4-9 and 4-10). For the nonlinear problem, however, this assumption is not valid because the evolution of temperature and thermal gradients is strongly dependent on the temperature distribution over the entire domain at previous times. This "path dependence" of temperature profile evolution is illustrated in Figure 4-11. For this particular case, doubling the slip rate from 0.5 to 1 m/s increases the peak temperature attained by ~ 30%. Due to the dependence of gradients on shear stress, the nonlinear effect is expected to be much stronger for large shear stresses.



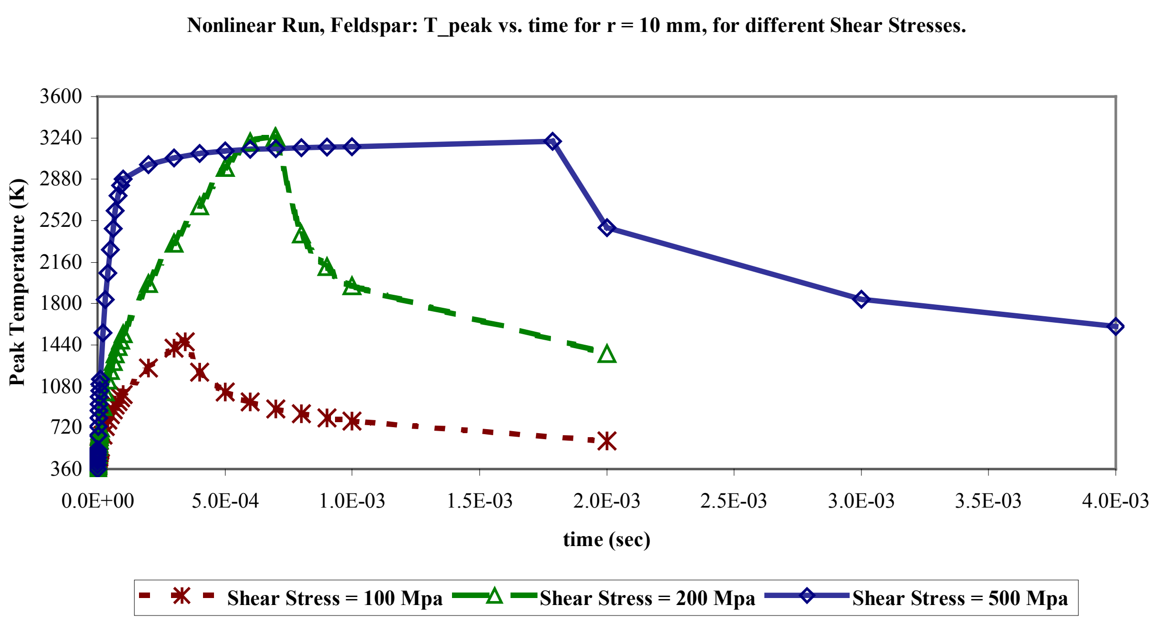


Figure 4-9. Temperature evolution profiles for different asperity radii and shear stresses for a sample set of nonlinear feldspar runs.

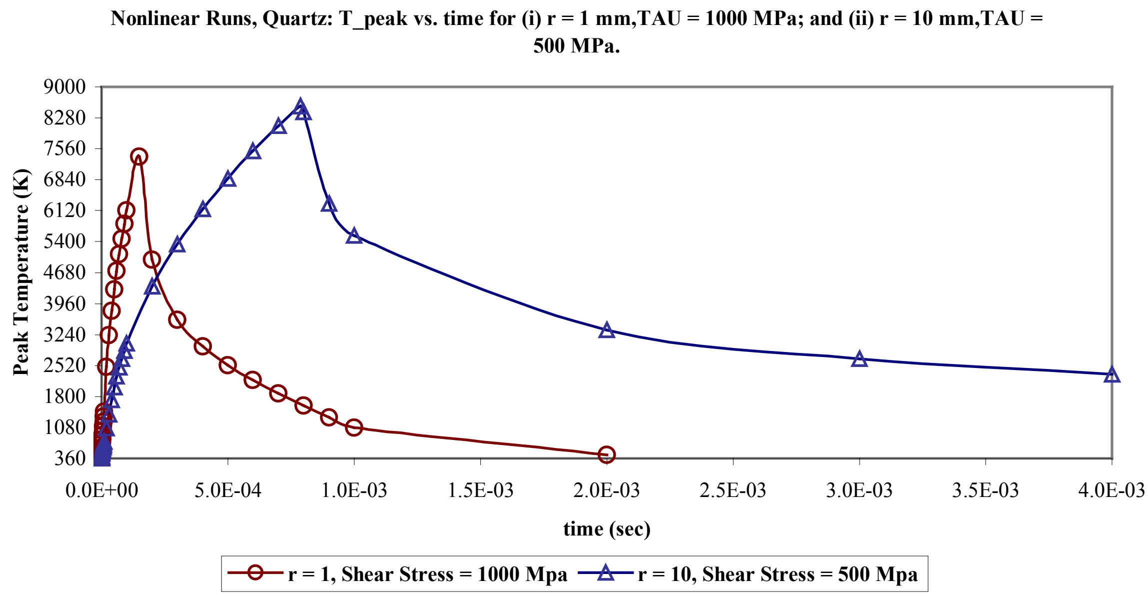


Figure 4-10. Temperature evolution profiles for different asperity radii and shear stresses for a sample set of nonlinear quartz runs.

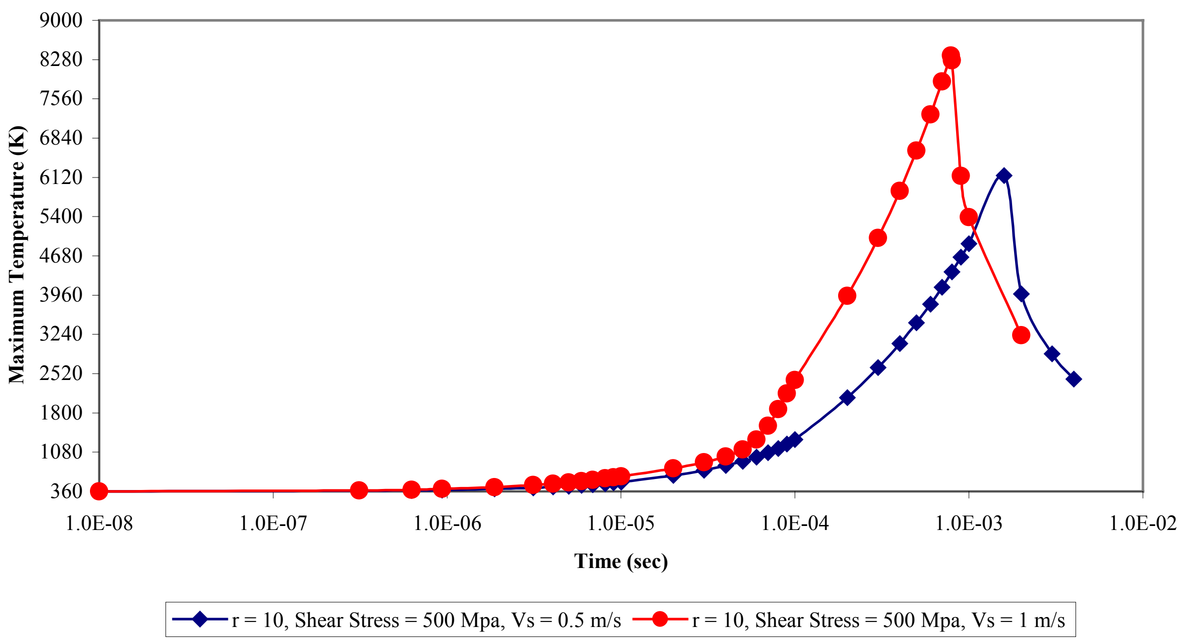
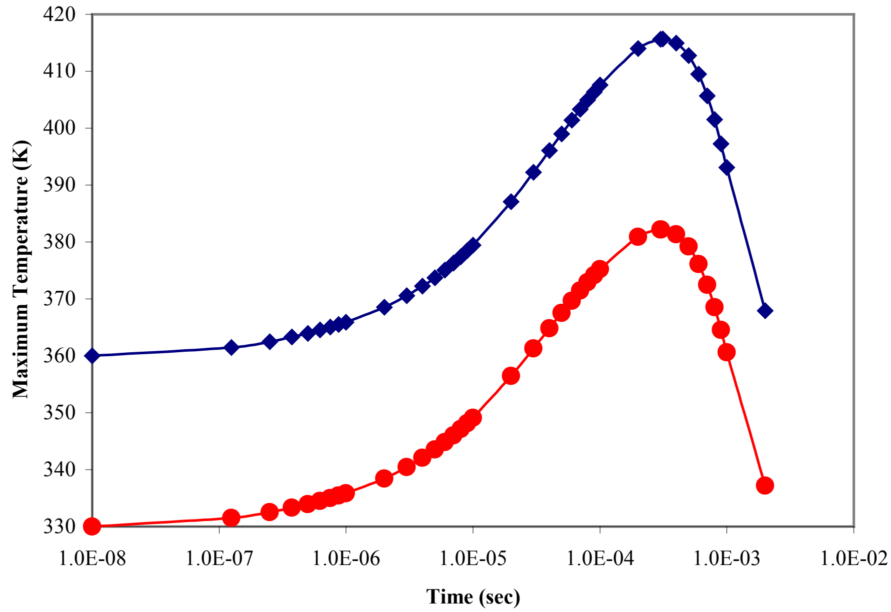


Figure 4-11. Effect of halving the slip velocity on the evolution of peak temperatures, on a 1 mm asperity experiencing a boundary shear stress of 100 MPa. Difference in Global Maximum Temperatures = 1.12^0 K ~ 1^0 K

Fault depth: For the linear problem, thermal effects at depth can be simulated by adding or subtracting a temperature increment determined by using the geothermal gradient (i.e., 30° C/km). For the nonlinear problem, however, since the initial temperature effects the initial domain diffusivity values, the same procedure cannot be used. In other words, changing the initial condition changes the "path" taken by the peak temperature (as discussed above), and therefore, the temperatures attained can be significantly different. In fact, the larger the driving thermal gradients (say, due to large shear stresses), the greater expected nonlinear deviation from this linear result. The results are illustrated in Figure 4-12, for sample low and high shear stresses. These plots show that decreasing the initial temperature by 30° K causes the maximum temperature to drop by a much higher value at a shear stress of 1000 MPa. At 500 MPa, however, the peak temperature drops by roughly the same magnitude as the change in initial temperature. For high shear stresses, however, the effect of changes in depth cannot be predicted without considering nonlinear effects. Nonetheless, the temperature change is a small fraction of the maximum temperature.

(a)



(b)

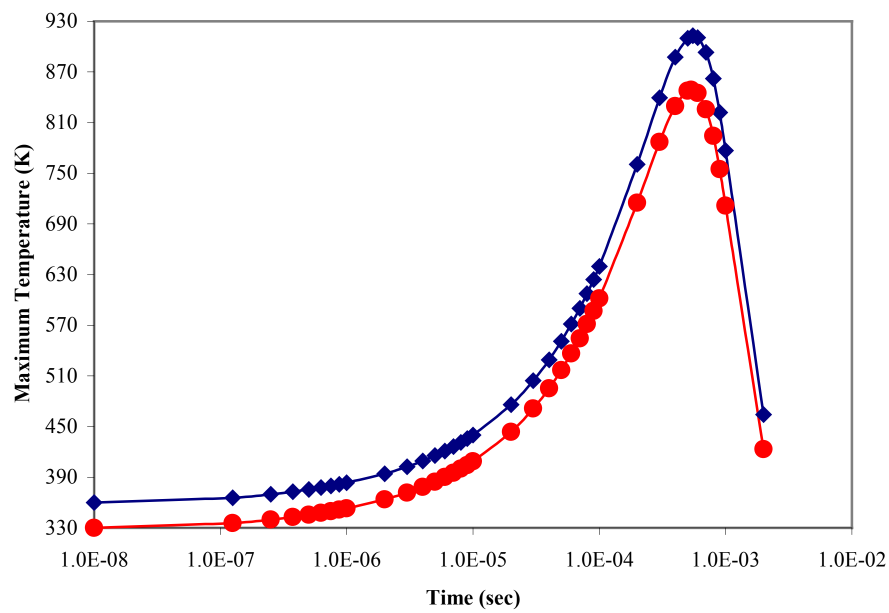


Figure 4-12. Effect of initial temperature on peak temperature evolution for the nonlinear quartz problem (1 mm asperity). Blue curve is for 2 km depth (T_0 = 360° K), Red curve for 1 km depth (T_0 = 330° K). (a) Shear Stress, 500 MPa: Global Maximum Temperature Difference = 33.47^0 K, (b) Shear Stress, 1 GPa: Global Maximum Temperature Difference = 63.78^0 K. Note that the temperature scales are not the same in (a) and (b).

4.4 Linear vs. nonlinear runs

As stated in Section 4.1, the published "constant" thermal conductivity value for quartz (4.3 $W.m^{-1}.K^{-1}$, see Table C-1) is 33% more than the temperature weighted average of the temperature dependent conductivity (computed using the Trapezoidal rule and data shown in Appendix C). On the other hand, the published "constant" value of thermal conductivity for Feldspar (1.35 $W.m^{-1}.K^{-1}$) is roughly 10% less than the temperature-weighted average. The linear feldspar models were, therefore, run with a thermal conductivity that was less than that of the nonlinear case on average. The resulting low diffusivity means that the peak temperatures observed for the linear feldspar runs were higher than their nonlinear counterparts. On the other hand, the conductivity of the linear quartz runs was higher on average than that for the nonlinear runs. Therefore, the peak temperatures produced in the linear quartz model were less than those in their nonlinear counterparts.

Unlike the linear case, a change in the initial condition (ambient temperature at fault depth) is critical in estimating peak temperatures for the nonlinear case. This was discussed in detail in the previous section.

The successful completion and convergence of the non-linear model runs is very sensitive to gradients within the problem domain. Although convergence of the linear runs is sensitive to the presence of steep gradients in the domain, they yield some result as long as all the parameters are within reasonable ranges. Hence, before using the results, extra care must be taken to make sure that the linear models do converge.

Based on the dramatic variation of thermal properties of most minerals (including the two used in this study), results from the linear models can be misleading. It is important to generate and use nonlinear modeling results when the relevant data is available. Temperature dependence of other model parameters like elastic properties and coefficient of friction are expected to further enhance nonlinear effects.

4.5 Conclusions

The main conclusions from this study are:

- While back of the envelope calculations can be used to determine rough orders of magnitude for parameters used to characterize heat conduction in asperities (like diffusion lengths), they cannot estimate the actual fraction of the asperity that could be experiencing near-melt temperatures. It is found for instance that the rate of propagation of the asperity temperature pulse along the radial direction is ~2-4 times higher than the predictions from the 1-D characteristic length scales for quartz, and roughly an order of magnitude higher than 1-D scales for feldspar.
- The temperatures obtained for certain combinations of asperity size and shear stress indicate that the local temperature rise can be as high as 8500° K for nonlinear quartz asperities, and 3200° K for feldspar asperities. In contrast, temperatures obtained from the infinite fault

plane models of Cardwell et. al. (1978), Oxburg and Turcotte (1974), and McKenzie and Brune (1972) were much higher, when calculated from their dimensionless plots. In fact in those models the temperature rise is directly proportional to the length and duration of fault slip, and yield extremely high values for the fault "plane" (~ $10^{5\circ}$ K).

- All else being equal, a larger volume of a feldspar asperity will melt compared to a quartz asperity. This follows from the fact that thermal conductivity of feldspar increases with increasing temperature, and is much higher than that of quartz, close to the feldspar melting point. However the melt volumes are very small (~ 0.3%). Pseudotachylyte occurrence is rare probably because it is very hard to initiate substantial frictional melting.
- Given the localized nature of any asperity scale melting, only repeated inter-asperity contact can create high enough temperatures to cause significant melting. Although rare, significant melting is suggested by kilometer long pseudotachylyte veins like those found in the Homestake Shear Zone (HSZ) in Colorado Rockies. Understanding the problem will require a fresh look at asperity size distributions on a fault surface and improved characterization of the surfaces. In conjunction with state-of-the-art thermal modeling, we suspect that the role of wear will also become important at the fault/macroscopic scale.
- For melting to occur, high shear stresses (500 – 1000 MPa) are required (due to the cubic dependence of peak temperatures on shear stress). Larger asperities would attain higher temperatures due to larger contact areas and contact durations compared to smaller asperities.

APPENDIX A: DETAILS OF NUMERICAL APPROACH

NUMERICAL SOLUTION OF THE GENERAL NONLINEAR 2D DIFFUSION EQUATION WITH GENERAL NONLINEAR BOUNDARY CONDITIONS:
DELTA-FORM OF NEWTON-KANTOROVICH SCHEME, IN CONJUNCTION WITH DELTA-FORM DOUGLAS-GUNN TIME SPLITTING.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

A-1. INTRODUCTION

In order to understand global tectonics and its evolution, fully coupled modeling of the earth's crust and mantle are required. Realistic geodynamic modeling of the earth will require integration of thermal transport (predominantly conduction/advection and convection), geo-hydrodynamics (ground water flow through porous media), geochemistry, and the thermo-viscoelastic response (Maxwell's solid) of the crust and mantle (as in the case of post-glacial crustal rebound) (see for instance, Ranalli 1995, Turcotte and Schubert 2001). Computing power exists today for such "full-spectrum" modeling. Within this framework, there is a need to develop a robust and flexible code for solving a coupled nonlinear system of generalized geo-thermal-hydrodynamic-viscoelastic equations. Towards this end, developing a general *single equation* 2D diffusion code is merely a first step.

A-1.1 Problem Specification

The problem for which the solution is being attempted is that of a general nonlinear transient pure conduction in 2 dimensions, in the variable u, with the self-adjoint form:

$$\left\{a_1(x,y,t).\frac{\partial}{\partial x}\left(a_2(x,y,t).k_t(u).\frac{\partial}{\partial x}\right)+b_1(x,y,t).\frac{\partial}{\partial x}\left(b_2(x,y,t).k_t(u).\frac{\partial}{\partial y}\right)\right\}(u)+f(u,x,y,t)=\rho_0 c_P(u)\frac{\partial u}{\partial t} \qquad \text{(A-1a)}$$

This can be compactly written in terms of the nonlinear functional, N, as follows:

$$\frac{\partial u}{\partial t}=N_1(u,u_x,u_y,u_{xx},u_{yy})+\left(\frac{f(u,x,y,t)}{\rho_0 c_P(u)}\right)=N(u,u_x,u_y,u_{xx},u_{yy}) \qquad \text{(A-1b)}$$

with general nonlinear boundary conditions:

$L(u,u_x) = f_L(y,t)$ (A-2a)
$R(u,u_x) = f_R(y,t)$ (A-2b)
$B(u,u_y) = f_B(x,t)$ (A-2c)
$Tu,u_y) = f_T(x,,t)$ (A-2d)

where, *L, R, B*, and *T* represent the left, right, bottom, and top (nonlinear) boundary functionals. For most standard heat conduction applications, each of the above functionals further take the generalized Robin form:

$F(u,u_{xi}) = F_1(u).\ u_{xi} + F_2(u)$ (A-3)

It was the goal here to develop a code that can handle the problem represented by equations (A-1)-(A-3). It will be shown later that the linear problems in any regular coordinate system are all special cases of the respective nonlinear problems. Therefore, the same code can be used to compute numerical solutions for linear or nonlinear problems - by setting the *linear_flag* to 1 or 0, respectively. For most geological applications it is sufficient to consider the three standard geometries: Cartesian, Cylindrical and Spherical. The table below provides the values of a_1, a_2, b_1, and b_2, for these three coordinate systems.

Table A- 1. Definitions of coefficients in Equation (A-1a), for the three standard coordinate systems.

↓ **Parameter / System→**	**Cartesian** (*coord_flag* = 1)	**Cylindrical** (*coord_flag* = 2)	**Spherical** (*coord_flag* = 3)
a_1	1	$1/x$	$1/x^2$
a_2	1	x	x^2
b_1	1	$1/x^2$	$1/\{x^2.Sin(y)\}$
b_2	1	1	$Sin(y)$
$a_{2,x}$	0	1	$2x$
$b_{2,y}$	0	0	$Cos(y)$

If required, however, the code is flexible enough to accommodate other user-defined geometries by allowing the definition of appropriate ***analytic (non-singular)*** expressions for the coefficients defined above. In this case *coord_flag = 0*. Of course, if the defined coefficients are not analytic, then appropriate modifications need to be made to approximate the PDE at the non-analytic points, and this requires modifications to the subroutine computing the coefficients and RHS vector of the tridiagonal system (see Sections A-2 and A-3). In this case, the code needs to be re-validated using known analytical solutions.

A-1.2 Existence and uniqueness of solutions

Before discussing the numerical implementation, the first issue is to figure out if anything can be said about the solutions to this general nonlinear equation, containing the second partial derivatives of the dependent variable, u. To the best of the author's knowledge, no such analysis exists for the particular problem chosen above. There have been numerous publications on the existence, uniqueness and stability of the solutions to the nonlinear heat conduction equation in various forms encountered in material science, plasma physics, thermal physics, engineering, and numerical analysis of the same. However, none that the author came across seem to discuss the appearance of second partial derivatives. As will be shown below, for realistic physical problems, and in the coordinate systems mentioned above, the derivatives of the functional w.r.t the second derivative of the dependent variable, u, i.e., N_{uxx}, and N_{uyy}, at least, are bounded. Although mathematically quite tenuous, this could imply that analyses similar to those for $N(u,u_x,u_y)$ may be still be applicable to this particular set of parabolic problems. In this respect, it is pertinent to discuss results from four papers on the numerical analysis applied specifically to the heat conduction problem, presented only as a sampling of how the analysis of nonlinear problems has evolved:

The first one is by Bellman (1948), who analyzed the existence and boundedness of solutions of the nonlinear heat conduction equation on a rectangular domain:

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}+\frac{\partial^2 u}{\partial z^2}+f(u,x,y,t)=g(x,y,t)\frac{\partial u}{\partial t} \qquad \text{(A-4)}$$

He analyzed the stability of this problem in the sense of Liapounoff-Poincare, and proved that if:

- *BCs are Dirichlet,*
- *IC exists and is bounded, and*
- *"RHS" function can be represented as a bounded series,*

then, a uniformly bounded solution exists, and is unique. He further goes on to extend this (albeit "hand-wavingly") to cylindrical polar coordinates, but cautions against extending the results to spherical/elliptical domains until further work was carried out (by him).
The second one is that by Douglas and Rachford (1956). In this, the authors develop their well known, alternating direction implicit time splitting scheme for *linear* 2- and 3-D heat conduction problems (and linear parabolic problems in general, along with an iterative scheme for the steady-state elliptic problem). They prove, using Taylor's series expansions for the derivatives, that for a any type of closed domain, if the initial and boundary values are such that u_{xxxx}, u_{xxyy}, u_{yyyy}, and u_{tt} are bounded, then the solution of the discrete split equations converges to that for the unsplit linear heat equation, to within ~ $O(h^2 + k)$. So, from arguments of the boundedness of the second derivatives presented above, a similar result may hold for equation (1a).

The following two papers illustrate typical numerical analysis procedures for the nonlinear heat conduction problem (and parabolic equations in general). The first one is by Dendy (1977), where the heat conduction equation of the form:

$$\left\{\frac{\partial}{\partial x}\left(a(x,y,u)\frac{\partial}{\partial x}\right)+\frac{\partial}{\partial x}\left(b(x,y,u)\frac{\partial}{\partial y}\right)\right\}(u)+f(x,y,t,u,u_x,u_y)=\frac{\partial u}{\partial t} \qquad \text{(A-5)}$$

which is in some ways significantly different from Equation (A-1a), in structure – It does not have a heat capacity term (in front of the time derivative), the "RHS" function is dependent on the first partial derivatives of the dependent. However, it still retains the nonlinear self-adjoint form of Equation (A-1a), and contains and since the "RHS" function does not contain any second derivatives, it may not influence the solution properties significantly (since the self-adjoint operator has first derivatives appearing in it, if they do not exist then the solution may not be easily computed). This is very close to the problem at hand, and its significance lies in the fact that, upon rewriting Equation (A-5) in its discretized self-adjoint form, it can be cast in the standard Douglas-Gunn time split form, with each step containing the discrete adjoint operator in a single direction. Dendy then goes on to prove (something not proven in Douglas and Gunn 1964) that for this nonlinear case, ***if:***

- *a_y and b_x are uniformly bounded,*
- *a_u, b_u are Lipschitz continuous w.r.t. u, and*
- *f is Lipschitz continuous w.r.t. u,u_x and u_y,*

then, *for a sufficiently small time step, the norm of the error varies as ~ $O(h^2 + k^2)$*, i.e.., 2nd order convergence rate can be obtained even in this nonlinear case! A more recent paper by Broadbridge et al. (1999) carries out a background study in terms of the qualitative properties of the solution of the radiant plasma heat conduction equation of the form:

$$\frac{\partial}{\partial x}\left(\rho(x).D(u).\frac{\partial u}{\partial x}\right)+f(t)=\frac{\partial u}{\partial t} \qquad \text{(A-6)}$$

for scale-invariant solutions, symmetries, and existence of solutions. One of the relevant conclusions from that paper to this project is that they find that if all the functions appearing in the above equation are "smooth", the initial profile of u is compatible with the boundary data, and all these data AND the coefficients are *strictly* positive, then Equation (A-6) possesses a "classical solution" for small enough time, and under further restrictive conditions, the number of local extrema of the solution, *u* do not increase with time.

Of course, Equation (A-1a) is more general than any of the equations presented above. In fact, Equations (A-4), (A-5) and (A-6) are special cases of that equation. From the above analyses, however, it seems reasonable to expect that the self-adjoint form of the heat conduction equation will have a *unique, bounded solution*, under restrictions of smoothness of all pertinent data.

A-1.3 Solution Method adopted

The method used here is the δ-form of the Newton-Kantorovich (N-K) procedure (or Quasi-linearization, which is actually a misnomer, since the nonlinear equations are fully linearized in this procedure) (see for instance, Kantorovich and Akilov 1964 & 1982), in conjunction with the δ-form of the Douglas-Gunn (D-G) scheme (Douglas and Gunn 1964, McDonough 2002). This combination renders the discretization in a form that is very efficient to implement. If it works at all, the Newton-Kantorovich scheme yields quadratic (or near quadratic) convergence, making it an easy choice from amongst direct substitution or Picard iteration methods for solving a nonlinear equation (or systems of equations). The D-G procedure is more general and robust (especially for non-smooth source functions, and at higher resolutions, more accurate), compared to the Peaceman-Rachford ADI method, which cannot be extended to more than 2 dimensions, or the Douglas-Rachford method, which is only first order accurate in time (McDonough 2002).

The solution procedure implemented here is limited by the machine specific maximum allowable array sizes, as it is designed to use *global solves* in each direction. This pitfall can be avoided by using some kind of Domain Decomposition and/or Multi-grid algorithms for the spatial discretization in conjunction with some form of Time Splitting for the temporal discretization. The state-of-the-art in computing Parabolic PDEs focuses on such methods in order to obtain solutions at higher grid resolutions. A recent example is a paper by Yu (2001), who has developed a local space-time adaptive scheme for solving 2-D parabolic problems, based on multiplicative Schwarz Domain Decomposition. He uses an *a posteriori* error estimator to determine the resolution of the grid required in each region of the problem domain – high "activity" results in finer space-time meshes, and *vice versa.* He solves an equation identical in form to (5) above, with mixed boundary conditions, assuming that the system is well posed. So, even for a 2-D code, what is being attempted here is merely a "starting" point. More complex issues involving integration of the 3D Finite Difference heat conduction and Finite Element viscoelastic codes will have to be ultimately resolved before this code can be used for realistic geophysical modeling.

A-2. DISCRETIZATION OF THE GENERAL DIFFUSION EQUATION.

Using Trapezoidal rule to integrate Equation (A-1b) between time levels n and n+1, we end up with:

$$u^{n+1^{(m+1)}} = u^n + \frac{k}{2}\left(N^{n+1^{(m+1)}} + N^n\right) \tag{A-7}$$

where *m* denotes the iteration counter. If the time step size, *k*, is small enough, then the first guess at the advanced time step will be the value at the previous time step. For the linear case, the iteration counter *m* is dropped, and the equivalent of Equation (7) is:

$$u^{n+1} = u^n + \frac{k}{2}\left(L(u)^{n+1} + L(u)^n\right) \tag{A-7'}$$

Here *L* is a linear operator (in the case of the heat conduction equation, this will be the *linear* form of the self adjoint operator and the *"RHS"* function, *f*, presented in Equation (A-1a): see Equation (A-15a') below). The nonlinear terms on the RHS of Equation (A-7) can be linearized by expanding N^{n+1} at $(m+1)^{th}$ iteration, in terms of N^{n+1} at the m^{th} iteration, to get:

$$u^{(m+1)} = u^n + \frac{k}{2}\left(N^{(m)} + N_u{}^{(m)}\delta u^{(m)} + N_{u_x}{}^{(m)}\delta u_x{}^{(m)} + N_{u_y}{}^{(m)}\delta u_y{}^{(m)} + N_{u_{xx}}{}^{(m)}\delta u_{xx}{}^{(m)} + N_{u_{yy}}{}^{(m)}\delta u_{yy}{}^{(m)} + N^n\right) \tag{A-8}$$

where, for notational convenience, the *n+1* advanced time level superscript has been suppressed. Also, we introduced the new term,

$$\delta u^{(m)} = u^{(m+1)} - u^{(m)} \tag{A-9}$$

Substituting for $u^{(m+1)}$, and rearranging Equation (A-9), we get:

$$\left[I - \frac{k}{2}\left\{\left(\frac{N_u{}^{(m)}}{2} + N_{u_x}{}^{(m)}D_{0,x} + N_{u_{xx}}{}^{(m)}D_{0,x}{}^2\right) + \left(\frac{N_u{}^{(m)}}{2} + N_{u_y}{}^{(m)}D_{0,y} + N_{u_{yy}}{}^{(m)}D_{0,y}{}^2\right)\right\}\right]\delta u^{(m)} = \left\{u^n + \frac{k}{2}\left(N^{(m)} + N^n\right)\right\} - u^{(m)} = R^{(m)} \tag{A-10}$$

The right hand side is nothing but the residual of the original semi-discrete equation (A-7). So, as $R^{(m)} \to 0$, $u^{(m+1)} \to u^{(m)}$, and therefore, $\delta u^{(m)} \to 0$. The convergence tolerance for $R^{(m)}$ must be at least k^3, for the iterations to converge (McDonough 2002), and *k* must be very small for the linearization to be applicable, unless u is known to be extremely smooth. Also, the functional $N_u^{(m)}$ has been split between the two directional operators equally, simply for preserving symmetry between the two directions. For the linear case, an equivalent relation to (A-10) will be:

$$\left[I - \frac{k}{2}\left\{\frac{\left(a_1 a_{2,x} k_t D_{0,x} + a_1 a_2 k_t D_{0,x}{}^2\right) + \left(b_1 b_{2,y} k_t D_{0,y} + b_1 b_2 k_t D_{0,y}{}^2\right)}{\rho_0 c_P}\right\}\right] u^{n+1} = \frac{k}{2.\rho_0 c_P}\left[\left(f^{n+1} + f^n\right) + k_t.\left\{\begin{array}{l}\left(a_1 a_{2,x} D_{0,x} + a_1 a_2 D_{0,x}{}^2 + b_1 b_{2,y} D_{0,y} + b_1 b_2 D_{0,y}{}^2\right)^{(n+1)} \\ + \left(a_1 a_{2,x} D_{0,x} + a_1 a_2 D_{0,x}{}^2 + b_1 b_{2,y} D_{0,y} + b_1 b_2 D_{0,y}{}^2\right)^{(n)}\end{array}\right\}.u^n\right] \tag{A-10'}$$

Now, comparing Equation (A-10) with the standard form of the Douglas-Gunn algorithm:

$$(I + A)\ \delta u^{(m)} = (I + A)\ [u^{(m+1)} - u^{(m)}] = s^{(m)} - Bu^{n} = S^{(m)} \tag{A-11}$$

where it has been assumed that $\delta u^{n} \cong 0$ (previous time step has converged to within the tolerance specified above), it can be seen that:

$$\begin{aligned} A &= -\frac{k}{2}\left\{\left(\frac{N_u^{(m)}}{2} + N_{u_x}{}^{(m)} D_{0,x} + N_{u_{xx}}{}^{(m)} D_{0,x}{}^{2}\right) + \left(\frac{N_u^{(m)}}{2} + N_{u_y}{}^{(m)} D_{0,y} + N_{u_{yy}}{}^{(m)} D_{0,y}{}^{2}\right)\right\} \\ \mathrm{B} &= -I \\ \mathrm{s}^{\mathrm{n}} &= \frac{k}{2}\left(N^{(m)} + N^{n}\right) - u^{(m)} \end{aligned} \tag{A-12}$$

So, the two level Douglas-Gunn scheme for this problem can be written as:

$$\left[I - \frac{k}{2}\left(\frac{N_u^{(m)}}{2} + N_{u_x}{}^{(m)} D_{0,x} + N_{u_{xx}}{}^{(m)} D_{0,x}{}^{2}\right)\right]\delta v^{(m)} = \left\{u^{n} + \frac{k}{2}\left(N^{(m)} + N^{n}\right)\right\} - u^{(m)} = R^{(m)} \tag{A-13a}$$

and,

$$\left[I - \frac{k}{2}\left(\frac{N_u^{(m)}}{2} + N_{u_y}{}^{(m)} D_{0,y} + N_{u_{yy}}{}^{(m)} D_{0,y}{}^{2}\right)\right]\delta u^{(m)} = \delta v^{(m)} \tag{A-13b}$$

and the value of the next iterate is given by a re-arrangement of Equation (A-9),

$$u^{(m+1)} = \delta u^{(m)} + u^{(m)} \tag{A-14}$$

For the linear case, the corresponding Douglas-Gunn scheme and the delta-form of the stages are represented by:

$$(I + A^{n+1}).\delta v = (I + A^{n+1}).(v - u^{n}) = s^{n} - \{(I + A^{n+1} + B^{n}).u^{n}\} \tag{A-11'}$$

Leading to:

$$\left[I - \frac{k}{2}\left(\frac{a_1 a_{2,x} k_t D_{0,x} + a_1 a_2 k_t D_{0,x}{}^{2}}{\rho_0 c_P}\right)^{n+1}\right]\delta v = \frac{k}{2.\rho_0 c_P}\left[\left(f^{n+1} + f^{n}\right) + k_t.\left\{\begin{matrix}\left(a_1 a_{2,x} D_{0,x} + a_1 a_2 D_{0,x}{}^{2} + b_1 b_{2,y} D_{0,y} + b_1 b_2 D_{0,y}{}^{2}\right)^{n+1} \\ +\left(a_1 a_{2,x} D_{0,x} + a_1 a_2 D_{0,x}{}^{2} + b_1 b_{2,y} D_{0,y} + b_1 b_2 D_{0,y}{}^{2}\right)^{n}\end{matrix}\right\}.u^{n}\right] \tag{A-13a'}$$

$$\left[I - \frac{k}{2}\left(\frac{b_1 b_{2,y} k_t D_{0,y} + b_1 b_2 k_t D_{0,y}{}^{2}}{\rho_0 c_P}\right)^{n+1}\right]\delta u = \delta v \tag{A-13b'}$$

where the superscripts denote time levels, and the value at the next time level is given by:

$$u^{n+1} = \delta u + u^{n} \tag{A-14'}$$

Thus, the *primed* equations above show that the delta-form time-splitting scheme for the linear problem (linear PDE + linear BCs) is very similar in form to the delta-form time-split scheme for the quasilinearized nonlinear equation. NOTE: In order to obtain Equation (A-13a') from Equation (A-13a), we need to set $\underline{\boldsymbol{u}^{(m)} = \boldsymbol{u}^{n}}$, since the RHS of the first stage is computed from the previous time step, instead of the previous iterate as in the linear case. Before expanding the difference operators, it should be noted that the LHS and RHS of (A-13a) and the LHS of (A-13b) contain functional derivatives evaluated with the last iterate of the advanced time step, and in case of the RHS of (A-13a), the nonlinear functional has to be evaluated at the previous time step, *n*. It will be easier to figure these terms out first, before any formal discretization of the time-split scheme itself is carried out. To do this, we have to first expand the self-adjoint form of the functional *N*, defined in (A-1b) and differentiate it according to the subscripts, to obtain:

$$\frac{\partial u}{\partial t} = N = \left[\frac{\left\{k_t.\left(a_1 a_{2,x}.u_x + a_1 a_2.u_{xx} + b_1 b_{2,y}.u_y + b_1 b_2.u_{yy}\right) + k_{t,u}.\left(a_1 a_2.u_x{}^2 + b_1 b_2.u_y{}^2\right)\right\} + f}{\rho_0 c_P}\right] \tag{A-15a}$$

where, the "independent" variables have been suppressed for clarity. For the linear case, we have:

$$\frac{\partial u}{\partial t} = L = \left[\frac{\left\{k_t.\left(a_1 a_{2,x}.u_x + a_1 a_2.u_{xx} + b_1 b_{2,y}.u_y + b_1 b_2.u_{yy}\right)\right\} + f}{\rho_0 c_P}\right] \tag{A-15a'}$$

Therefore, differentiating (A-15a) with respect to u, u_x, u_y, u_{xx}, and u_{yy}, we obtain, (for both the linear and nonlinear cases):

$$N_u = \left[\frac{\left\{\left(k_{t,u}.c_P - k_t.c_{P,u}\right)\left(a_1 a_{2,x}.u_x + a_1 a_2.u_{xx} + b_1 b_{2,y}.u_y + b_1 b_2.u_{yy}\right) + \left(k_{t,uu}.c_P - k_{t,u}.c_{P,u}\right)\left(a_1 a_2.u_x{}^2 + b_1 b_2.u_y{}^2\right)\right\} + \left(f_u.c_P - f.c_{P,u}\right)}{\rho_0 c_P{}^2}\right] \tag{A-15b}$$

$L_u = 0$ (since all the derivatives w.r.t u, of k_t and c_p, are all equal to 0). (A-*15b'*)

$$N_{u_x} = \frac{a_1.\left(k_t a_{2,x} + 2.k_{t,u}.a_2.u_x\right)}{\rho_0 c_P} \tag{A-15c}$$

$$L_{u_x} = \frac{a_1.\left(k_t a_{2,x}\right)}{\rho_0 c_P} \tag{A-15c'}$$

$$N_{u_y} = \frac{b_1.\left(k_t b_{2,y} + 2.k_{t,u}.b_2.u_y\right)}{\rho_0 c_P} \tag{A-15d}$$

$$L_{u_y} = \frac{b_1.\left(k_t b_{2,y}\right)}{\rho_0 c_P} \tag{A-15d'}$$

$$N_{u_{xx}} = \frac{a_1.a_2.k_t}{\rho_0 c_P} \tag{A-15e}$$

$$L_{u_{xx}} = \frac{a_1.a_2.k_t}{\rho_0 c_P} \tag{A-15e'}$$

$$N_{u_{yy}} = \frac{b_1.b_2.k_t}{\rho_0 c_P} \tag{A-15f}$$

$$L_{u_{yy}} = \frac{b_1.b_2.k_t}{\rho_0 c_P} \qquad \text{(A-}15f'\text{)}$$

Thus, except as noted under Equations (A-13), all the linear expressions can be derived from their nonlinear counterparts by setting the derivatives of the thermal properties w.r.t temperature, u, to zero – i.e., the linear problem can be solved using the nonlinear code as a special (built-in) case.

For realistic values of k_t and c_p, the last two functional derivatives (A-15e & f) are *always bounded*, since c_p cannot be 0. This will become important in analyzing the discrete equations for determining the coefficients, as shown below. Since these values are always computed with the previous iterate, they are always available at the advanced iteration. In order to compute Equations (A-15), we need to compute $u_x^{(m)}$, $u_y^{(m)}$, $u_{xx}^{(m)}$, and $u_{yy}^{(m)}$, since $u^{(m)}$ is already available (via storage). Although higher order methods can be used here, for higher accuracy (McDonough 2002), 2nd order centered differencing will be used here, for simplicity. The computation of these partial derivatives at interior grid points (*i=2:*N_x*-1, j=2,* N_y*-1*) is straightforward. However, the boundaries require special treatment. The added complication here is that the boundaries could be nonlinear, as shown in Equation (A-2) and (A-3) above. If the BC is linear-Dirichlet, then, it does not matter what the derivative value is, as no computations will be carried out at that boundary – values are just assigned for each time step, that remain fixed as the nonlinear iterations progress. However, if the BC is nonlinear-Dirichlet, or any other type of boundary, it will have to be dealt with through the use of image points outside the problem domain in the BC as well as the PDE, as illustrated for boundary value problems in McDonough (2001). Only, here, if the BCs are nonlinear, the "linearized" BCs have to be used instead of the actual BCs. Given a set of BCs, and previous iteration grid functions, these derivatives can be computed in a straightforward manner – this will be indicated below when considering the different boundaries during the point-by-point discretization. Once functional values and functional derivatives are computed at all the grid points, the coefficients and RHS vectors for the interior, boundary, and corner points can be computed.

A-2.1 Interior Points

Expanding the difference operators in each element of the matrix equations (A-13a) and (A-13b), by using standard centered-difference approximations, we get:

$$\left[\delta v_{j,i}^{(m)} - \frac{k}{2}\left\{\frac{N_{u_{j,i}}^{(m)}}{2}\delta v_{j,i}^{(m)} + N_{u_{x\,j,i}}^{(m)}.\left(\frac{\delta v_{j+1,i}^{(m)} - \delta v_{j-1,i}^{(m)}}{2.h_x}\right) + N_{u_{xx\,j,i}}^{(m)}.\left(\frac{\delta v_{j+1,i}^{(m)} - 2.\delta v_{j,i}^{(m)} + \delta v_{j-1,i}^{(m)}}{h_x^2}\right)\right\}\right] = \left\{u^n + \frac{k}{2}\left(N^{(m)} + N^n\right)\right\}_{j,i} - u_{j,i}^{(m)} = R_{j,i}^{(m)}$$

(A-*13a"*)

and,

$$\left[\delta u_{j,i}^{(m)} - \frac{k}{2}\left(\frac{N_{u_{j,i}}^{(m)}}{2}\delta u_{j,i}^{(m)} + N_{u_{y\,j,i}}^{(m)}.\left(\frac{\delta u_{j,i+1}^{(m)} - \delta u_{j,i-1}^{(m)}}{2.h_y}\right) + N_{u_{yy\,j,i}}^{(m)}.\left(\frac{\delta u_{j,i+1}^{(m)} - 2.\delta u_{j,i}^{(m)} + \delta u_{j,i-1}^{(m)}}{h_y^2}\right)\right)\right] = \delta v_{j,i}^{(m)} \qquad \text{(A-}13b''\text{)}$$

Collecting like terms, we obtain:

$$-\left(\frac{k}{2.h_x^2}\right).\left\{N_{u_{xx}}^{(m)} - \left(\frac{h_x.N_{u_x}^{(m)}}{2}\right)\right\}_{j,i}\delta v_{j,i-1}^{(m)} + \left\{\left(\frac{k}{2.h_x^2}\right).2N_{u_{xx}}^{(m)} + \left(1 - \frac{k.N_u^{(m)}}{4}\right)\right\}_{j,i}\delta v_{j,i}^{(m)} - \left(\frac{k}{2.h_x^2}\right).\left\{N_{u_{xx}}^{(m)} + \left(\frac{h_x.N_{u_x}^{(m)}}{2}\right)\right\}_{j,i}\delta v_{j,i+1}^{(m)}$$

$$= \left\{u^n + \frac{k}{2}\left(N^{(m)} + N^n\right)\right\}_{j,i} - u_{j,i}^{(m)}$$

(A-*13a'''*)

$$-\left(\frac{k}{2h_y{}^2}\right).\left\{N_{u_{yy}}{}^{(m)}-\left(\frac{h_y.N_{u_y}{}^{(m)}}{2}\right)\right\}_{j,i}\delta u_{j-1,i}{}^{(m)}+\left\{\left(\frac{k}{2h_y{}^2}\right).2N_{u_{yy}}{}^{(m)}+\left(1-\frac{k.N_u{}^{(m)}}{4}\right)\right\}_{j,i}\delta u_{j,i}{}^{(m)}-\left(\frac{k}{2h_y{}^2}\right).\left\{N_{u_{yy}}{}^{(m)}+\left(\frac{h_y.N_{u_y}{}^{(m)}}{2}\right)\right\}_{j,i}\delta u_{j+1,i}{}^{(m)}=\delta v_{j,i}{}^{(m)}$$

(A-*13b'''*)

Substituting $\boldsymbol{\rho_x = k/2h_x^2}$ into the first equation and dividing it throughout by $\boldsymbol{\rho_x}$, then substituting $\boldsymbol{\rho_y = k/2h_y^2}$ into the second equation and dividing it throughout by $\boldsymbol{\rho_y}$, we obtain the following "compact form" after rearrangement:

$$\left\{1-\left(\frac{h_x.N_{u_x}{}^{(m)}}{2.N_{u_{xx}}{}^{(m)}}\right)_{j,i}\right\}\delta v_{j,i-1}{}^{(m)}-\left\{2+\left(\frac{4-k.N_u{}^{(m)}}{4\rho_x.N_{u_{xx}}{}^{(m)}}\right)_{j,i}\right\}\delta v_{j,i}{}^{(m)}+\left\{1+\left(\frac{h_x.N_{u_x}{}^{(m)}}{2.N_{u_{xx}}{}^{(m)}}\right)_{j,i}\right\}\delta v_{j,i+1}{}^{(m)}=-\frac{\left\{u^n+\frac{k}{2}\left(N^{(m)}+N^n\right)\right\}_{j,i}-u_{j,i}{}^{(m)}}{\rho_x.N_{u_{xx}\,j,i}{}^{(m)}}$$

(A-16a)

$$\left\{1-\left(\frac{h_y.N_{u_y}{}^{(m)}}{2.N_{u_{yy}}{}^{(m)}}\right)_{j,i}\right\}\delta u_{j-1,i}{}^{(m)}-\left\{2+\left(\frac{4-k.N_u{}^{(m)}}{4\rho_y.N_{u_{yy}}{}^{(m)}}\right)_{j,i}\right\}\delta u_{j,i}{}^{(m)}+\left\{1+\left(\frac{h_y.N_{u_y}{}^{(m)}}{2.N_{u_{yy}}{}^{(m)}}\right)_{j,i}\right\}\delta u_{j+1,i}{}^{(m)}=-\frac{\delta v_{j,i}{}^{(m)}}{\rho_y.N_{u_{yy}\,j,i}{}^{(m)}} \qquad \text{(A-16b)}$$

where, the indexing notation used follows the Fortran 90 rules, i.e., (row#, column#), for ease of implementation. NOTE: Unless otherwise indicated, ALL nonlinear functionals (*N* & its derivatives) are evaluated at the advanced time step, *n+1*.

For the linear case, from the definition of the Linear Operator and its derivatives (Equations (A-15') above), along with Equations (A-13'), these expressions become:

$$\left\{1-\left(\frac{h_x.L_{u_x}}{2.L_{u_{xx}}}\right)_{j,i}\right\}\delta v_{j,i-1}{}^{(1)}-\left\{2+\frac{1}{\rho_x.L_{u_{xx}\,j,i}}\right\}\delta v_{j,i}{}^{(1)}+\left\{1+\left(\frac{h_x.L_{u_x}}{2.L_{u_{xx}}}\right)_{j,i}\right\}\delta v_{j,i+1}{}^{(1)}=-\frac{\left\{\frac{k}{2}\left(L^{n+1}+L^n\right)u_{j,i}{}^n\right\}}{\rho_x.L_{u_{xx}\,j,i}} \qquad \text{(A-}\textit{16a'}\text{)}$$

$$\left\{1-\left(\frac{h_y.L_{u_y}}{2.L_{u_{yy}}}\right)_{j,i}\right\}\delta u_{j-1,i}-\left\{2+\frac{1}{\rho_y.L_{u_{yy}\,j,i}}\right\}\delta u_{j,i}+\left\{1+\left(\frac{h_y.L_{u_y}}{2.L_{u_{yy}}}\right)_{j,i}\right\}\delta u_{j+1,i}=-\frac{\delta v_{j,i}{}^{(1)}}{\rho_y.L_{u_{yy}\,j,i}} \qquad \text{(A-}\textit{16b'}\text{)}$$

NOTE: These linear expressions can also be obtained by "replacing" the functional *N* and its derivatives by the corresponding linear versions (since *L* is a *special case* of *N*) in Equations (A-16), then using the fact that $\boldsymbol{u_{j,i}{}^{(m)} = u_{j,i}{}^n}$. However, the linear functional *L* and its derivatives *must* still be computed at the next time level, *n+1*, in order to obtain 2nd order convergence of grid functions.

The coefficients of δu & δv on the LHS of both sets of equations form tri-diagonal systems that can be efficiently solved using LU-Decomposition. From the expressions presented above in (A-15e and f), and comments presented below these, the denominator of either set of coefficients should not vanish, for real systems. So, in order to guarantee diagonal dominance of the system represented by Equations (A-16), we need, for Equation (A-16b), for instance:

$$\left|2+\frac{4-k.N_{u\,j,i}{}^{(m)}}{4\rho_x.N_{u_{xx}\,j,i}{}^{(m)}}\right|\geq\left|1-\frac{h_x.N_{u_x\,j,i}{}^{(m)}}{2.N_{u_{xx}\,j,i}{}^{(m)}}\right| \qquad \text{(A-17)}$$

Taking LCMs and rearranging, this gives a relationship between $N_u^{(m)}$ and $N_{uy}^{(m)}$, of the form:

$$N_{uy}^{\ 2} \le (\alpha . N_u - \beta)^2 \tag{A-18}$$

where, α and β are constant once h_y, k, k_t, C_P, b_1 and b_2 are fixed. So, for unconditional stability of the LU-Decomposition scheme, from the definition of N_{uy} , Equation (A-15d), we need to have EITHER a constant k_t (so $k_{t,u}=0$) OR $u_y=0$; AND be in the Cartesian system (so $a_{2,x}=0$)! Since the problem proposed to be solved here is the solution of the spherical heat conduction equation with a temperature dependent thermal conductivity, Equation (A-18) may be satisfied for only certain locations in the domain, or maybe, nowhere in the domain! Also, it must be noted that all the functional derivatives change with the location of the grid point, and with time. So, in general, any relation of the form (A-18) cannot hold for the entire spatio-temporal domain of the problem unless $N_{uy}=0$ *AND* $N_u \le 4/k$ (from Equation (A-17)) in the entire domain. Similar relations will hold for Equation (A-16a), for the second orthogonal direction. **Hence, we are not guaranteed a solution to the NONLINEAR problem selected in the previous chapter.** On the other hand, ***the linear problem is guaranteed a solution*** since diagonal dominance is assured [see Equations (A-16')].

A-2.2 Corner Points

The implementation of corner points can be tricky, but here the methodology adopted is as follows:

- If adjacent BCs at a corner are Dirichlet, then the average of the two values is chosen.
- If one of the adjacent BCs at a corner is Dirichlet, the its value over-rides that of the other.
- If both BCs at a corner point are non-Dirichlet, then quite arbitrarily, it is assigned the value of the relevant left or right BC, ignoring the corresponding top or bottom BC.

A-2.3 Boundary Points

A-2.3.1 Left Boundary & Left Corner Points

Consider the general nonlinear BC presented in Equation (A-2) above:

$$Lf(u,u_x) = f_L(y,t_{n+1}) \tag{A-19}$$

If the BC is non-Dirichlet, it can be linearized by expanding the LHS functional, Lf, to the third term in the Frechet-Taylor's series about the previous iterate, to get:

$$Lf^{(m)} + Lf_u^{\ (m)}.\delta u^{(m)} + Lf_{u_x}^{\ (m)}.\delta u_x^{\ (m)} \cong f_L^{\ n+1} \tag{A-20}$$

Rearranging,

$$\left(Lf_u^{\ (m)} + Lf_{u_x}^{\ (m)} D_{0,x}\right)\delta u^{(m)} \cong f_L^{\ n+1} - Lf^{(m)} \tag{A-21}$$

Expanding the centered difference approximation, we can obtain an estimate for the value of the "image point", $\delta u_{j,0}^{(m)}$, and thus, be able to solve the split step equations (A-16), at the left boundary.

Substituting the expression for $D_{0,x}$ into Equation (A-21), we get:

$$\left[\left\{Lf_u{}^{(m)}.\delta u\right\}_{j,1}{}^{(m)} + \left\{Lf_{u_x}{}^{(m)}\right\}_{j,1}\left(\frac{\delta u_{j,2}{}^{(m)} - \delta u_{j,0}{}^{(m)}}{2.h_x}\right)\right] \cong f_{L,j}{}^{n+1} - Lf_{j,1}{}^{(m)} \tag{A-22}$$

Rearranging (A-22), we get:

$$2h_x\left\{Lf_u{}^{(m)}.\delta u\right\}_{j,1}{}^{(m)} + \left\{Lf_{u_x}{}^{(m)}\right\}_{j,1}.\delta u_{j,2}{}^{(m)} - \left\{Lf_{u_x}{}^{(m)}\right\}_{j,1}.\delta u_{j,0}{}^{(m)} \cong 2h_x\left(f_{L,j}{}^{n+1} - Lf_{j,1}{}^{(m)}\right) \tag{A-23}$$

We now use the same notation as in Equations (A-16) for the purpose of substitution - noting that only the left and right boundaries need be considered in the first step of (A-16), and only the top and bottom boundaries need be considered in the second step of (A-16). Therefore, we adopt the same notation for the unknown variables at each stage: v for the first stage, and u for the second stage, for the sake of consistency and minimizing confusion. We thus have:

$$\delta v_{j,0}{}^{(m)} \cong \left(\frac{2h_x.Lf_u{}^{(m)}}{Lf_{u_x}{}^{(m)}}\right)_{j,1}\delta v_{j,1}{}^{(m)} + \delta v_{j,2}{}^{(m)} - 2h_x.\left(\frac{f_L{}^{n+1} - Lf^{(m)}}{Lf_{u_x}{}^{(m)}}\right)_{j,1} \tag{A-24}$$

For the linear case, we get correspondingly:

$$u_{j,0}{}^{n} = 2h_x.\alpha_x.u_{j,1}{}^{n} + u_{j,2}{}^{n} - 2h_x.f_{L,j}{}^{n} \tag{A-25a}$$

and,

$$v_{j,0} = 2h_x.\alpha_x.v_{j,1} + v_{j,2} - 2h_x.f_{L,j}{}^{n+1} \tag{A-25b}$$

Therefore,

$$\delta v_{j,0} = v_{j,0} - u_{j,0}{}^{n} \cong 2h_x.\alpha_x.\delta v_{j,1} + \delta v_{j,2} - 2h_x.(f_{L,j}{}^{n+1} - f_{L,j}{}^{n}) \tag{A-24'}$$

NOTE: For deriving Equation (A-*24'*), use has been made of the definitions of the image points for both v and u_n. $f_L{}^{n+1}$ corresponds to the former next time level, and $f_L{}^n$ corresponds to the last time level, n. Also, α_x is the linear Robin BC parameter (as in: $u_x + \alpha_x.u$), and will be 0 (zero) for the linear Neumann BC. The linear Equation (A-*24'*) can also be obtained from the nonlinear Equation (A-24) as a special case, by setting $Lf^{(m)} = f_L{}^n$, $Lf_u{}^{(m)} = \alpha_x$,, and $Lf_{ux}{}^{(m)} = 1$. Thus, (A-*24'*) is a special case of (A-24).

Setting $i=1$ in both (A-16a and b), and substituting (A-24) into Equation (A-16a) we finally get, for the left boundary:

$$-\left\{2 + \left(\frac{4 - k.N_u{}^{(m)}}{4\rho_x.N_{u_{xx}}{}^{(m)}}\right)_{j,1} - \left(\frac{2h_x.Lf_u{}^{(m)}}{Lf_{u_x}{}^{(m)}}\right)_{j,1}.\left\{1 - \frac{h_x.N_{u_x}{}^{(m)}}{2.N_{u_{xx}}{}^{(m)}}\right\}_{j,1}\right\}\delta v_{j,1}{}^{(m)} + 2\delta v_{j,2}{}^{(m)} = -\frac{\left\{u^n + \frac{k}{2}\left(N^{(m)} + N^n\right)\right\}_{j,1} - u_{j,i}{}^{(m)}}{\rho_x.N_{u_{xx}\,j,1}{}^{(m)}} + 2h_x.\left(\frac{f_L - Lf^{(m)}}{Lf_{u_x}{}^{(m)}}\right)_{j,1}.\left\{1 - \frac{h_x.N_{u_x}{}^{(m)}}{2.N_{u_{xx}}{}^{(m)}}\right\}_{j,1} \tag{A-26a}$$

and,

$$\left\{1-\left(\frac{h_y.N_{u_y}{}^{(m)}}{2.N_{u_{yy}}{}^{(m)}}\right)_{j,1}\right\}\delta u_{j-1,1}{}^{(m)}-\left\{2+\left(\frac{4-k.N_u{}^{(m)}}{4\rho_x.N_{u_{yy}}{}^{(m)}}\right)_{j,1}\right\}\delta u_{j,1}{}^{(m)}+\left\{1+\left(\frac{h_y.N_{u_y}{}^{(m)}}{2.N_{u_{yy}}{}^{(m)}}\right)_{j,1}\right\}\delta u_{j+1,1}{}^{(m)}=-\frac{\delta v_{j,1}{}^{(m)}}{\rho_y.N_{u_{yy}\,j,1}{}^{(m)}} \tag{A-26b}$$

NOTE: Unless otherwise indicated, ALL nonlinear functionals (N & its derivatives) are evaluated at the advanced time step, $n+1$. For a **nonlinear problem with a <u>nonlinear</u> Dirichlet left boundary condition**, we consider the expansion in (A-21) to only the 2nd term:

$$\left(Lf_u{}^{(m)}.\right)\delta v^{(m)} \cong f_L{}^{n+1}-Lf^{(m)} \tag{A-27}$$

and the left grid points are **<u>assigned</u>** as follows:

$$\delta v_{j,1}{}^{(m)}=v_{j,1}{}^{(m+1)}-v_{j,1}{}^{(m)}\cong\left(\frac{f_L{}^{n+1}-Lf^{(m)}}{Lf_u{}^{(m)}}\right)_{j,1} \tag{A-28a}$$

For a **nonlinear problem with a <u>linear or nonlinear</u> Dirichlet left boundary condition**, this reduces to:

$$\delta v_{j,1}{}^{(m)}\cong 0 \qquad\qquad \text{for all } m>0 \tag{A-28b}$$

Irrespective of the linearity of the boundary condition, if the PDE is nonlinear, all functional values for the first iteration ($m = 0$, according to the notation used here) have to be evaluated at the previous time level in order to take into account the time dependence of the Dirichlet condition. This also follows naturally from the fact that the first guess for the advanced time step is the converged value at the end of the last time step. If these were evaluated at the advanced time level $n+1$, then the boundary value will remain the same as at $t = t_0$. So, $v^{(0)} = u^n$, $Lf^{(0)} = Lf^n$, and $Lf_u{}^{(0)} = Lf_u{}^n$:

$$\delta v_{j,1}{}^{(0)}=v_{j,1}{}^{(1)}-v_{j,1}{}^{(0)}\cong\left(\frac{f_L{}^{n+1}-Lf^{n}}{Lf_u{}^{n}}\right)_{j,1} \tag{A-29}$$

It must be kept in mind that for the particular class of problems being considered, as shown in Equation (A-3), the boundary functional takes on the form of a generalized Robin BC:

$$Lf(u,u_x) = Lf_1(u)\,.\,u_x + Lf_2(u) \tag{A-30}$$

In this case, Equations (A-23) through (A-29) can be modified accordingly and everything expressed in terms of Lf_1 and Lf_2.

For the linear problem, the corresponding expressions can be obtained by substituting Equation (A-*24'*) into Equation (A-*16a'*) or using "linear substitutions" in Equations (A-26), namely: $Lf^{(m)} = f_L{}^n$, $Lf_u{}^{(m)} = \alpha_x$,, and $Lf_{ux}{}^{(m)} = 1$:

$$-\left\{2+\frac{1}{\rho_x.L_{u_{xx}\,j,1}}-2h_x\alpha_x.\left\{1-\frac{h_x.L_{u_x}}{2.L_{u_{xx}}}\right\}_{j,1}\right\}\delta v_{j,1}{}^{(1)}+2.\delta v_{j,2}{}^{(1)}=-\frac{\left\{\frac{k}{2}\left(L^{n+1}+L^n\right).u_{j,1}{}^{n}\right\}}{\rho_x.L_{u_{xx}\,j,1}}+2h_x.(f_{L,j}{}^{n+1}-f_{L,j}{}^{n}).\left\{1-\frac{h_x.L_{u_x}}{2.L_{u_{xx}}}\right\}_{j,1} \tag{A-26a'}$$

$$\left\{1-\frac{h_y.L_{u_y}}{2.L_{u_{yy}}}\right\}_{j,1}\delta u_{j-1,1}-\left\{2+\frac{1}{\rho_y.L_{u_{yy}\,j,1}}\right\}\delta u_{j,1}+\left\{1+\frac{h_y.L_{u_y}}{2.L_{u_{yy}}}\right\}_{j,1}\delta u_{j+1,1}=-\frac{\delta v_{j,1}{}^{(1)}}{\rho_y.L_{u_{yy}\,j,1}} \tag{A-26b'}$$

NOTE: Unless otherwise indicated, ALL linear functionals (L & its derivatives) are evaluated at the advanced time step, $n+1$. So, Equations (A-26') are special cases of Equations (A-26) above. Here, for a Neumann BC, $\alpha_x = 0$. For a Robin BC, $\alpha_x \neq 0$. For a linear Dirichlet BC, $Lf^{(m),n} = f_L^{n}$, and $Lf_u^{(m),n} = 1$ in (A-29). That gives:

$$\delta v_{j,1} = f_L^{n+1} - f_L^{n}. \tag{A-29'}$$

Only when f_L is a constant with respect to time, would we have for the linear problem:

$$\delta v_{j,1} = 0. \tag{A-31}$$

Spherical or Cylindrical Coordinates: In case of spherical or cylindrical coordinates, the forms of a_1 presented in the Table A-1 imply that the PDE is not analytic at $x = 0$. In both these cases, however, symmetry arguments require: $\boldsymbol{u_x(r=0) = 0, u_y(r=0) = 0, u_{yy}(r=0) = 0, u_{yx}(r=0) = 0, u_{yxx}(r=0) = 0, u_{yyx}(r=0) = 0, u_{yyxx}(r=0) = 0}$. Therefore, the limiting value of the PDE as $x \rightarrow 0$ can b e evaluated using L'Hospital's rule. For the general nonlinear functional, we have:

$$\lim_{x\to 0}(N) = \lim_{x\to 0}\left[\frac{\left\{k_t.\left(a_1 a_{2,x}.u_x + a_1 a_2.u_{xx} + b_1 b_{2,y}.u_y + b_1 b_2.u_{yy}\right) + k_{t,u}.\left(a_1 a_2.u_x^{\;2} + b_1 b_2.u_y^{\;2}\right)\right\} + f}{\rho_0 c_P}\right] \tag{A-32a}$$

For the spherical system, using the expressions for coefficients a_1, a_2, b_1, and b_2 from Table A-1 above, we obtain:

$$\lim_{x\to 0}(N_S) = \lim_{x\to 0}\left[\frac{\left\{k_t.\left[2.\frac{u_x}{x} + u_{xx} + \frac{1}{x^2}\left(\frac{u_y}{Tan(y)} + u_{yy}\right)\right] + k_{t,u}.\left(u_x^{\;2} + \frac{u_y^{\;2}}{x^2}\right)\right\} + f}{\rho_0 c_P}\right] \tag{A-32b}$$

If all the symmetry conditions above are met, then we obtain:

$$\lim_{x\to 0}(N_S) = \left[\frac{\left\{k_t.\left(2.u_{xx} + u_{xx}\right) + k_{t,u}.u_x^{\;2}\right\} + f}{\rho_0 c_P}\right]_{x=0} = \left[\frac{3.k_t.u_{xx} + k_{t,u}.u_x^{\;2} + f}{\rho_0 c_P}\right]_{x=0} \tag{A-32c}$$

Similarly, for the cylindrical system, we get:

$$\lim_{x\to 0}(N_C) = \lim_{x\to 0}\left[\frac{\left\{k_t.\left[\frac{u_x}{x} + u_{xx} + \frac{u_{yy}}{x^2}\right] + k_{t,u}.\left(u_x^{\;2} + \frac{u_y^{\;2}}{x^2}\right)\right\} + f}{\rho_0 c_P}\right] = \left[\frac{2.k_t.u_{xx} + k_{t,u}.u_x^{\;2} + f}{\rho_0 c_P}\right]_{x=0} \tag{A-33}$$

So, for the general form of the functional presented in Equation (A-15a), we can generalize (A-32c) and (A-33) as:

$$N_{x\to 0} = \frac{k_t.\left(C_{factor} + 1\right).u_{xx} + k_{t,u}.u_x^{\;2} + f}{\rho_0 c_P} = N_0 \tag{A-34a}$$

with $C_{factor}=2$ for the spherical system and *1*, for the cylindrical system. Note that the result is obtained with the assumption that ALL mixed derivatives are zero (by symmetry), so none of the terms originally containing the y derivative remains. Therefore, the derivatives required in the indicial form of (A-34a) (equivalent to Equations (A-15)) are:

$$N_{0,u_y} = 0 = N_{0,u_{yy}} \tag{A-34b}$$

and

$$N_{0,u} = \frac{(c_P k_{t,u} - c_{P,u} k_t).(C_{factor}+1).u_{xx} + (c_P k_{t,uu} - k_{t,u}.c_{P,u}).u_x^{\ 2} + (c_P f_u - f.c_{P,u})}{\rho_0 c_P^{\ 2}} \tag{A-34c}$$

Similarly,

$$N_{0,u_x} = \frac{2.k_{t,u}.u_x}{\rho_0 c_P} \tag{A-34d}$$

$$N_{0,u_{xx}} = \frac{k_t.(C_{factor}+1)}{\rho_0 c_P} \tag{A-34e}$$

Thus, for either spherical or cylindrical coordinate system, at $x = 0$, the implementation of the PDE (A-34a) becomes:

$$-\left\{2+\left(\frac{4-k.N_{0,u}^{\ (m)}}{4\rho_x.N_{0,u_{xx}}^{\ (m)}}\right)_{j,1}-\left(\frac{2h_x.Lf_u^{\ (m)}}{Lf_{u_x}^{\ (m)}}\right)_{j,1}.\left\{1-\frac{h_x.N_{0,u_x}^{\ (m)}}{2.N_{0,u_{xx}}^{\ (m)}}\right\}_{j,1}\right\}\delta v_{j,1}^{\ (m)}+2\delta v_{j,2}^{\ (m)}=-\frac{\left\{u^n+\frac{k}{2}\left(N_0^{\ (m)}+N_0^{\ n}\right)\right\}_{j,1}-u_{j,i}^{\ (m)}}{\rho_x.N_{0,u_{xx}\,j,1}^{\ (m)}}+2h_x.\left(\frac{f_L-Lf^{(m)}}{Lf_{u_x}^{\ (m)}}\right)_{j,1}.\left\{1-\frac{h_x.N_{0,u_x}^{\ (m)}}{2.N_{0,u_{xx}}^{\ (m)}}\right\}_{j,1} \tag{A-35a}$$

NOTE: Unless otherwise indicated, ALL nonlinear functionals (*N* & its derivatives) are evaluated at the advanced time step, *n+1*. Since the nonlinear operator in (A-34a) is now devoid of functional derivatives in the y-direction, the solution after the second split step is the same as the "intermediate solution", δv, obtained after the first step:

$$\delta u_{j,1}^{(m)} = \delta v_{j,1}^{(m)} \tag{A-35b}$$

Equations (34) are still valid for the linear case. However, Equation (34c) and (34d) become:

$$L_{0,u} = 0 = L_{0,u_x} \tag{A-34c'/d'}$$

Therefore, the linear version of Equation (35a) will be:

$$-\left\{2+\frac{1}{\rho_x.L_{0,u_{xx}\,j,1}}-2h_x\alpha_x\right\}\delta v_{j,1}^{\ (1)}+2.\delta v_{j,2}^{\ (1)}=-\frac{\left\{\frac{k}{2}\left(L_0^{\ n+1}+L_0^{\ n}\right)u_{j,1}^{\ n}\right\}}{\rho_x.L_{0,u_{xx}\,j,1}}+2h_x.(f_{L,j}^{\ n+1}-f_{L,j}^{\ n}) \tag{A-35a'}$$

NOTE: Unless otherwise indicated, ALL nonlinear functionals (L & its derivatives) are evaluated at the advanced time step, $n+1$. Again, for the second stage:

$$\delta u_{j,1} = \delta v_{j,1} \qquad \text{(A-}35b'\text{)}$$

Again, the linear case is a special case of the nonlinear case. Now, in order to compute the coefficients and RHS terms of Equations (A-26 / *26'*) and (A-35 / *35'*), we need to be able to compute the values of the derivatives u_x, u_{xx}, u_y, and u_{yy}, at all points on the left boundary. These derivatives are evaluated at the left boundary only if the Left BC is *non-Dirichlet*. If the Left BC is Dirichlet, the values as assigned as per Equations (A-28), (A-29 / *29'*), and (A-31) above. For calculating the derivatives at the boundaries, use can be made of the basic form of heat transfer boundary conditions [Equation (A-3)]. Thus, for non-Dirichlet BCs, and $j = 1, 2, 3, \ldots, N_y$, these derivatives can be expressed as:

$$\left(u_x\right)_{j,1}^{(m)} \cong \left(D_{0,x}\right)_{j,1}^{(m)} = \frac{u_{j,2}^{(m)} - u_{j,0}^{(m)}}{2.h_x} = \left(\frac{f_L^{\,n+1} - Lf_2^{\,(m)}}{Lf_1^{\,(m)}}\right)_{j,1} \qquad \text{(A-36)}$$

Therefore,

$$u_{j,0}^{(m)} = u_{j,2}^{(m)} - 2.h_x.\left(\frac{f_L^{\,n+1} - Lf_2^{\,(m)}}{Lf_1^{\,(m)}}\right)_{j,1} \qquad \text{(A-37)}$$

and,

$$\left(u_{xx}\right)_{j,1}^{(m)} \cong \left(D_{0,x}^{\,2}\right)_{j,1}^{(m)} = \frac{u_{j,2}^{(m)} - 2.u_{j,1}^{(m)} + u_{j,0}^{(m)}}{h_x^{\,2}} = \frac{2.u_{j,2}^{(m)} - 2.u_{j,1}^{(m)} - 2.h_x.\left(\frac{f_L^{\,n+1} - Lf_2^{\,(m)}}{Lf_1^{\,(m)}}\right)_{j,1}}{h_x^{\,2}} \qquad \text{(A-38)}$$

The y-derivatives at the left boundary can be computed as in Equations (A-*13"*), except at the corner points (j=1 and j=N_y). So, for j = 2, 3, …., (N_y–*1)*:

$$\left(u_y\right)_{j,1}^{(m)} \cong \left(D_{0,y}\right)_{j,1}^{(m)} = \frac{u_{j+1,1}^{(m)} - u_{j-1,1}^{(m)}}{2.h_y} \qquad \text{(A-39)}$$

If the bottom boundary condition is not Dirichlet (in which case, it must be assigned that value), then for $j=1$,

$$\left(u_y\right)_{1,1}^{(m)} \cong \left(D_{0,y}\right)_{1,1}^{(m)} = \frac{u_{2,1}^{(m)} - u_{0,1}^{(m)}}{2.h_y} = \left(\frac{f_B^{\,n+1} - B_2^{\,(m)}}{B_1^{\,(m)}}\right)_{1,1} \qquad \text{(A-40)}$$

Therefore,

$$u_{0,1}^{(m)} = u_{2,1}^{(m)} - 2.h_y\left(\frac{f_B^{\,n+1} - B_2^{\,(m)}}{B_1^{\,(m)}}\right)_{1,1} \qquad \text{(A-41)}$$

and,

$$\left(u_{yy}\right)_{1,1}^{(m)} \cong \left(D_{0,y}^{\,2}\right)_{1,1}^{(m)} = \frac{u_{2,1}^{(m)} - 2.u_{1,1}^{(m)} + u_{0,1}^{(m)}}{h_y^{\,2}} = \frac{2.u_{2,1}^{(m)} - 2.u_{1,1}^{(m)} - 2.h_y\left(\frac{f_B^{\,n+1} - B_2^{\,(m)}}{B_1^{\,(m)}}\right)_{1,1}}{h_y^{\,2}} \qquad \text{(A-42)}$$

Similarly, if the top boundary condition is not Dirichlet (in which case, it must be assigned that value), then for $j=N_y$,

$$\left(u_y\right)_{N_y,1}^{(m)} \cong \left(D_{0,y}\right)_{N_y,1}^{(m)} = \frac{u_{N_y+1,1}^{(m)} - u_{N_y-1,1}^{(m)}}{2.h_y} = \left(\frac{f_T^{\,n+1} - T_2^{(m)}}{T_1^{(m)}}\right)_{N_y,1} \tag{A-43}$$

Therefore,

$$u_{N_y+1,1}^{(m)} = u_{N_y-1,1}^{(m)} + 2.h_y\left(\frac{f_T^{\,n+1} - T_2^{(m)}}{T_1^{(m)}}\right)_{N_y,1} \tag{A-44}$$

and,

$$\left(u_{yy}\right)_{N_y,1}^{(m)} \cong \left(D_{0,y}^{\,2}\right)_{N_y,1}^{(m)} = \frac{u_{N_y+1,1}^{(m)} - 2.u_{N_y,1}^{(m)} + u_{N_y-1,1}^{(m)}}{h_y^{\,2}} = \frac{2.h_y\left(\frac{f_T^{\,n+1} - T_2^{(m)}}{T_1^{(m)}}\right)_{N_y,1} - 2.u_{N_y,1}^{(m)} + 2.u_{N_y-1,1}^{(m)}}{h_y^{\,2}} \tag{A-45}$$

All derivatives at <u>time level *n*</u> can be obtained by replacing the iteration superscript *(m)* by the time level superscript, *n*, and then changing all $f^{\,n+1}$ to $f^{\,n}$, in Equations (A-36)-(A-45). In the linear non-Dirichlet cases, the following substitutions will make Equations (A-36)-(A-45) consistent: (a) <u>*Linear Neumann*</u> – $\boldsymbol{Lf_2 = 0}$ (zero), and $\boldsymbol{Lf_1 = 1}$, and (b) <u>*Linear Robin*</u> – $\boldsymbol{Lf_2 = \alpha_x.u^n}$, and $\boldsymbol{Lf_1 = 1}$. In addition, all RHS terms containing *u* are evaluated at time level *n,* for these cases. Therefore, ***f_L is evaluated at time level n*** (instead of at *n+1*). So, the <u>***left boundary and corner points***</u> are completely taken care of, for all three coordinate systems.

A-2.3.2 Right Boundary & Right Corner Points

NOTE: Since the coefficients of the spherical PDE are not analytic at $y = 0$ or $y = \pi$, the following analysis does not apply to the <u>*right corner points*</u> (both top & bottom) for a spherical coordinate system problem. Consider the general nonlinear BC presented in Equation (A-2) above:

$$R(u,u_x) = f_R(y,t_{n+1}) \tag{A-46}$$

If the BC is <u>non-Dirichlet</u>, it can be linearized by expanding the LHS functional about the previous iterate, to the third term in the Frechet-Taylor's series, to get:

$$R^{(m)} + R_u^{(m)}.\delta u^{(m)} + R_{u_x}^{(m)}.\delta u_x^{(m)} \cong f_R^{\,n+1} \tag{A-47}$$

Rearranging,

$$\left(R_u^{(m)} + R_{u_x}^{(m)} D_{0,x}\right)\delta u^{(m)} \cong f_R^{\,n+1} - R^{(m)} \tag{A-48}$$

Expanding the centered difference approximation, we can obtain an estimate for the value of the image point, $\delta u_{j,Nx+1}^{(m)}$, and thus, be able to solve the split step equations (A-16), at the right boundary. Substituting the expression for $D_{0,x}$ into Equation (A-48), we get:

$$\left[\left\{ R_u^{(m)} .\delta u \right\}_{j,N_x}^{(m)} + \left\{ R_{u_x}^{(m)} \right\}_{j,N_x} \left(\frac{\delta u_{j,N_x+1}^{(m)} - \delta u_{j,N_x-1}^{(m)}}{2.h_x} \right) \right] \cong f_{R,j}^{n+1} - R_{j,N_x}^{(m)} \qquad \text{(A-49)}$$

Rearranging (A-49), we get:

$$2h_x \left\{ R_u^{(m)} .\delta u \right\}_{j,N_x}^{(m)} + \left\{ R_{u_x}^{(m)} \right\}_{j,N_x} .\delta u_{j,N_x+1}^{(m)} - \left\{ R_{u_x}^{(m)} \right\}_{j,N_x} .\delta u_{j,N_x-1}^{(m)} \cong 2h_x \left(f_{R,j}^{n+1} - R_{j,N_x}^{(m)} \right) \qquad \text{(A-50)}$$

Adopting the same notation as above for the unknown variables at each stage: v for the first stage, and u for the second stage, we thus have:

$$\delta v_{j,N_x+1}^{(m)} \cong -\left(\frac{2h_x .R_u^{(m)}}{R_{u_x}^{(m)}} \right)_{j,N_x} \delta v_{j,N_x}^{(m)} + \delta v_{j,N_x-1}^{(m)} + 2h_x . \left(\frac{f_R^{n+1} - R^{(m)}}{R_{u_x}^{(m)}} \right)_{j,N_x} \qquad \text{(A-51)}$$

For the linear case, we get correspondingly:

$$\delta v_{j,N_x+1}^{(1)} = v_{j,N_x+1}^{(1)} - u_{j,N_x+1}^{n} \cong -2h_x .\alpha_x .\delta v_{j,N_x}^{(1)} + \delta v_{j,N_x+1}^{(1)} + 2h_x .(f_{R,j}^{n+1} - f_{R,j}^{n}) \qquad \text{(A-51')}$$

NOTE: Just as for the Left BC, α_x is the linear Robin BC parameter (as in: $u_x + \alpha_x.u$), and will be 0 (zero) for the linear Neumann BC. The linear Equation (A-*51'*) can also be obtained from the nonlinear Equation (A-51) as a special case, by setting $R^{(m)} = f_R^{\,n}$, $R_u^{(m)} = \alpha_x$,, and $R_{ux}^{(m)} = 1$.

Setting $i=N_x$ in both (A-16a and b), and substituting (A-51) into Equation (A-16a) we obtain for the right boundary:

$$2\delta v_{j,N_x-1}^{(m)} - \left\{ 2 + \left(\frac{4 - k.N_u^{(m)}}{4\rho_x .N_{u_{xx}}^{(m)}} \right)_{j,N_x} + \left(\frac{2h_x .R_u^{(m)}}{R_{u_x}^{(m)}} \right)_{j,N_x} . \left\{ 1 + \frac{h_x .N_{u_x}^{(m)}}{2.N_{u_{xx}}^{(m)}} \right\}_{j,N_x} \right\} \delta v_{j,N_x}^{(m)} = -\frac{\left\{ u^n + \frac{k}{2}\left(N^{(m)} + N^n \right) \right\}_{j,N_x} - u_{j,N_x}^{(m)}}{\rho_x .N_{u_{xx}\,j,N_x}^{(m)}} - 2h_x . \left(\frac{f_R - R^{(m)}}{R_{u_x}^{(m)}} \right)_{j,N_x} . \left\{ 1 + \frac{h_x .N_{u_x}^{(m)}}{2.N_{u_{xx}}^{(m)}} \right\}_{j,N_x} \qquad \text{(A-52a)}$$

and,

$$\left\{ 1 - \left(\frac{h_y .N_{u_y}^{(m)}}{2.N_{u_{yy}}^{(m)}} \right)_{j,N_x} \right\} \delta u_{j-1,N_x}^{(m)} - \left\{ 2 + \left(\frac{4 - k.N_u^{(m)}}{4\rho_x .N_{u_{yy}}^{(m)}} \right)_{j,N_x} \right\} \delta u_{j,N_x}^{(m)} + \left\{ 1 + \left(\frac{h_y .N_{u_y}^{(m)}}{2.N_{u_{yy}}^{(m)}} \right)_{j,N_x} \right\} \delta u_{j+1,N_x}^{(m)} = -\frac{\delta v_{j,N_x}^{(m)}}{\rho_y .N_{u_{yy}\,j,N_x}^{(m)}} \qquad \text{(A-52b)}$$

NOTE: Unless otherwise indicated, ALL nonlinear functionals (N & its derivatives) are evaluated at the advanced time step, $n+1$. For a **nonlinear problem with a <u>nonlinear</u> Dirichlet right boundary condition**, we consider the expansion in (A-47) to only the 2[nd] term:

$$\left(R_u^{(m)} . \right) \delta v^{(m)} \cong f_R^{n+1} - R^{(m)} \qquad \text{(A-53)}$$

and the right grid points are **<u>assigned</u>** as follows:

$$\delta v_{j,N_x}^{(m)} \cong \left(\frac{f_R^{n+1} - R^{(m)}}{R_u^{(m)}} \right)_{j,N_x} \qquad \text{(A-54a)}$$

For a **nonlinear problem with a <u>linear or nonlinear</u> Dirichlet right boundary condition**, this reduces to:

$\delta v_{j,Nx}{}^{(m)} = 0.$ for all $m > 0$ (A-54b)

Irrespective of the linearity of the boundary condition, if the PDE is nonlinear, all functional values for the first iteration ($m = 0$, according to the notation used here) have to be evaluated at the previous time level in order to take into account the time dependence of the Dirichlet condition. This also follows naturally from the fact that the first guess for the advanced time step is the converged value at the end of the last time step. If these were evaluated at the advanced time level $n+1$, then the boundary value will remain the same as at t = t_0. So, $v^{(0)} = u^n$, $R^{(0)} = R^n$, and $R_u^{(0)} = R_u^n$:

$$\delta v_{j,N_x}{}^{(0)} = v_{j,N_x}{}^{(1)} - v_{j,N_x}{}^{(0)} \cong \left(\frac{f_R{}^{n+1} - R^n}{R_u{}^n} \right)_{j,N_x} \tag{A-55}$$

It must be kept in mind that for the particular class of problems being considered, as shown in Equation (A-3), the boundary condition takes on the form of a generalized Robin BC:

$$R(u,u_x) = R_1(u) \,.\, u_x + R_2(u) \tag{A-56}$$

In this case, Equations (A-47) through (A-55) can be modified accordingly and everything expressed in terms of R_1 and R_2. For the special case of a linear problem with linear right boundary condition, the substitutions: $R^{(m)} = f_R^n$, $R_u^{(m)} = \alpha_x$, and $R_{ux}^{(m)} = 1$, can be made in Equations (A-52), just as for the left boundary condition, along with the appropriate linear functional substitutions.

Unlike the left boundary, the modified equations for the right boundary hold for all three coordinate systems (Cartesian and Cylindrical – all along the right boundary; <u>for **Spherical - all along the right boundary, except at $y = 0$ or $y = \pi$**</u>). In order to evaluate Equations (A-52), we need to evaluate the functional derivatives at the right boundary, and these in turn depend on the first and second derivatives of the dependent variable: u_x, u_{xx}, u_y, and u_{yy}, at all points on the right boundary. Again, these derivatives are evaluated at the right boundary only if the Right BC is *non-Dirichlet*. If the Right BC is Dirichlet, the values as assigned as per Equations (A-54) and (A-55) above. For calculating the derivatives at the boundaries, use can be made of the basic form of heat transfer boundary conditions [Equation (A-3)]. Thus, for non-Dirichlet BCs, for $j = 1, 2, 3, \ldots, N_y$, the derivatives can be expressed as:

$$(u_x)_{j,N_x}{}^{(m)} \cong (D_{0,x})_{j,N_x}{}^{(m)} = \frac{u_{j,N_x+1}{}^{(m)} - u_{j,N_x-1}{}^{(m)}}{2.h_x} = \left(\frac{f_R{}^{n+1} - R_2{}^{(m)}}{R_1{}^{(m)}} \right)_{j,N_x} \tag{A-57}$$

Therefore,

$$u_{j,N_x+1}{}^{(m)} = u_{j,N_x-1}{}^{(m)} + 2.h_x . \left(\frac{f_R{}^{n+1} - R_2{}^{(m)}}{R_1{}^{(m)}} \right)_{j,N_x} \tag{A-58}$$

$$(u_{xx})_{j,N_x}{}^{(m)} \cong (D_{0,x}{}^2)_{j,N_x}{}^{(m)} = \frac{u_{j,N_x+1}{}^{(m)} - 2.u_{j,N_x}{}^{(m)} + u_{j,N_x-1}{}^{(m)}}{h_x{}^2} = \frac{2.u_{j,N_x-1}{}^{(m)} - 2.u_{j,N_x}{}^{(m)} + 2.h_x . \left(\frac{f_R{}^{n+1} - R_2{}^{(m)}}{R_1{}^{(m)}} \right)_{j,N_x}}{h_x{}^2} \tag{A-59}$$

The y-derivatives at the right boundary can be computed as for the left boundary [Equations (A-39) - (A-45)], except at the corner points (j=1 and $j=N_y$). So, for j = 2, 3, …., (N_y-1):

$$\left(u_y\right)_{j,N_x}^{(m)} \cong \left(D_{0,y}\right)_{j,N_x}^{(m)} = \frac{u_{j+1,N_x}^{(m)} - u_{j-1,N_x}^{(m)}}{2.h_y} \tag{A-60}$$

For $j=1$,

$$\left(u_y\right)_{1,N_x}^{(m)} \cong \left(D_{0,y}\right)_{1,N_x}^{(m)} = \frac{u_{2,N_x}^{(m)} - u_{0,N_x}^{(m)}}{2.h_y} = \left(\frac{f_B^{\,n+1} - B_2^{(m)}}{B_1^{(m)}}\right)_{1,N_x} \tag{A-61}$$

Therefore,

$$u_{0,N_x}^{(m)} = u_{2,N_x}^{(m)} - 2.h_y\left(\frac{f_B^{\,n+1} - B_2^{(m)}}{B_1^{(m)}}\right)_{1,N_x} \tag{A-62}$$

and,

$$\left(u_{yy}\right)_{1,N_x}^{(m)} \cong \left(D_{0,y}^{\,2}\right)_{1,N_x}^{(m)} = \frac{u_{2,N_x}^{(m)} - 2.u_{1,N_x}^{(m)} + u_{0,N_x}^{(m)}}{h_y^{\,2}} = \frac{2.u_{2,N_x}^{(m)} - 2.u_{1,N_x}^{(m)} - 2.h_y\left(\frac{f_B^{\,n+1} - B_2^{(m)}}{B_1^{(m)}}\right)_{1,N_x}}{h_y^{\,2}} \tag{A-63}$$

Similarly, for $j=N_y$,

$$\left(u_y\right)_{N_y,N_x}^{(m)} \cong \left(D_{0,y}\right)_{N_y,N_x}^{(m)} = \frac{u_{N_y+1,N_x}^{(m)} - u_{N_y-1,N_x}^{(m)}}{2.h_y} = \left(\frac{f_T^{\,n+1} - T_2^{(m)}}{T_1^{(m)}}\right)_{N_y,N_x} \tag{A-64}$$

Therefore,

$$u_{N_y+1,N_x}^{(m)} = u_{N_y-1,N_x}^{(m)} + 2.h_y\left(\frac{f_T^{\,n+1} - T_2^{(m)}}{T_1^{(m)}}\right)_{N_y,N_x} \tag{A-65}$$

and,

$$\left(u_{yy}\right)_{N_y,N_x}^{(m)} \cong \left(D_{0,y}^{\,2}\right)_{N_y,N_x}^{(m)} = \frac{u_{N_y+1,N_x}^{(m)} - 2.u_{N_y,N_x}^{(m)} + u_{N_y-1,N_x}^{(m)}}{h_y^{\,2}} = \frac{2.h_y\left(\frac{f_T^{\,n+1} - T_2^{(m)}}{T_1^{(m)}}\right)_{N_y,N_x} - 2.u_{N_y,N_x}^{(m)} + 2.u_{N_y-1,N_x}^{(m)}}{h_y^{\,2}} \tag{A-66}$$

As with the left boundary, all derivatives at time level n can be obtained by replacing the iteration superscript *(m)* by the time level superscript, n, and then changing all f^{n+1} to f^n, in Equations (A-57)-(A-66). In the linear non-Dirichlet cases, the following substitutions will make Equations (A-57)-(A-66) consistent: (a) Linear Neumann – $\boldsymbol{R_2} = \mathbf{0}$ (zero), and $\boldsymbol{R_1} = \mathbf{1}$, and (b) Linear Robin – $\boldsymbol{R_2} = \boldsymbol{\alpha_x}.\boldsymbol{u^n}$, and $\boldsymbol{R_1} = \mathbf{1}$. In addition, for these cases, all RHS terms containing u are evaluated at time level n. Therefore, ***f_R is evaluated at time level n*** (instead of at $n+1$). So, the ***right boundary and corner points (except for spherical)*** are completely taken care of, for all three coordinate systems.

A-2.3.3 Bottom Boundary

NOTE: For the spherical system, the right bottom corner point will be considered here. For the other two coordinate systems, we do not consider the corner points here since they were considered under the left and right boundaries described above. Consider the general nonlinear BC presented in Equation (A-2) above:

$$B(u,u_y) = f_B(x,t_{n+1}) \tag{A-67}$$

If the BC is non-Dirichlet, it can be linearized by expanding the LHS functional about the previous iterate, to the third term in the Frechet-Taylor's series, to get:

$$B^{(m)} + B_u{}^{(m)}.\delta u^{(m)} + B_{u_y}{}^{(m)}.\delta u_y{}^{(m)} \cong f_B{}^{n+1} \tag{A-68}$$

Rearranging,

$$\left(B_u{}^{(m)} + B_{u_y}{}^{(m)} D_{0,y}\right).\delta u^{(m)} \cong f_B{}^{n+1} - B^{(m)} \tag{A-69}$$

Expanding the centered difference approximation, we can obtain an estimate for the value of the image point, $\delta u_{0,i}{}^{(m)}$, and thus, be able to solve the split step equations (A-16), at the bottom boundary. Substituting the expression for $D_{0,y}$ into Equation (A-69), we get:

$$\left[\left\{B_u{}^{(m)}.\delta u\right\}_{1,i}{}^{(m)} + \left\{B_{u_y}{}^{(m)}\right\}_{1,i}\left(\frac{\delta u_{2,i}{}^{(m)} - \delta u_{0,i}{}^{(m)}}{2.h_y}\right)\right] \cong f_{B,i}{}^{n+1} - B_{1,i}{}^{(m)} \tag{A-70}$$

Rearranging (A-70), we get:

$$2h_y\left\{B_u{}^{(m)}.\delta u\right\}_{1,i}{}^{(m)} + \left\{B_{u_y}{}^{(m)}\right\}_{1,i}.\delta u_{2,i}{}^{(m)} - \left\{B_{u_y}{}^{(m)}\right\}_{1,i}.\delta u_{0,i}{}^{(m)} \cong 2h_y\left(f_{B,i}{}^{n+1} - B_{1,i}{}^{(m)}\right) \tag{A-71}$$

We now use the same notation as in Equations (A-16) for the unknown variables at each stage: v for the first stage, and u for the second stage. We thus have, for $i = 2, 3, ..., N_x - 1$:

$$\delta u_{0,i}{}^{(m)} \cong \left(\frac{2h_y.B_u{}^{(m)}}{B_{u_y}{}^{(m)}}\right)_{1,i} \delta u_{1,i}{}^{(m)} + \delta u_{2,i}{}^{(m)} - 2h_y.\left(\frac{f_B{}^{n+1} - B^{(m)}}{B_{u_y}{}^{(m)}}\right)_{1,i} \tag{A-72}$$

For the linear case, we get correspondingly:

$$\delta u_{0,i}{}^{(m)} \cong 2h_y.\alpha_y.\delta u_{1,i}{}^{(m)} + \delta u_{2,i}{}^{(m)} - 2h_y.(f_{B,i}{}^{n+1} - f_{B,i}{}^{n}) \tag{A-72'}$$

NOTE: α_y is the linear Robin BC parameter (as in: $u_y + \alpha_y.u$), and will be 0 (zero) for the linear Neumann BC. The linear Equation (A-72') was obtained from the nonlinear Equation (A-72) as a special case, by setting $B^{(m)} = f_B{}^n$, $B_u{}^{(m)} = \alpha_y$,, and $B_{uy}{}^{(m)} = 1$. Setting j=*1* in both (A-16a and b), and substituting (A-72) into Equation (A-16b), we finally get, for the bottom boundary:

$$\left\{1-\left(\frac{h_x.N_{u_x}{}^{(m)}}{2.N_{u_{xx}}{}^{(m)}}\right)_{1,i}\right\}\delta v_{1,i-1}{}^{(m)} - \left\{2+\left(\frac{4-k.N_u{}^{(m)}}{4\rho_x.N_{u_{xx}}{}^{(m)}}\right)_{1,i}\right\}\delta v_{1,i}{}^{(m)} + \left\{1+\left(\frac{h_x.N_{u_x}{}^{(m)}}{2.N_{u_{xx}}{}^{(m)}}\right)_{1,i}\right\}\delta v_{1,i+1}{}^{(m)} = -\frac{\left\{u^n+\frac{k}{2}\left(N^{(m)}+N^n\right)\right\}_{1,i} - u_{1,i}{}^{(m)}}{\rho_x.N_{u_{xx}1,i}{}^{(m)}} \tag{A-73a}$$

$$-\left\{2+\left(\frac{4-k.N_u{}^{(m)}}{4\rho_y.N_{u_{yy}}{}^{(m)}}\right)_{1,i} - \left(\frac{2h_y.B_u{}^{(m)}}{B_{u_y}{}^{(m)}}\right)_{1,i}.\left\{1-\left(\frac{h_y.N_{u_y}{}^{(m)}}{2.N_{u_{yy}}{}^{(m)}}\right)_{1,i}\right\}\right\}\delta u_{1,i}{}^{(m)} + 2\delta u_{2,i}{}^{(m)} = -\frac{\delta v_{1,i}{}^{(m)}}{\rho_y.N_{u_{yy}1,i}{}^{(m)}} + 2h_y.\left(\frac{f_B-B^{(m)}}{B_{u_y}{}^{(m)}}\right)_{1,i}\left\{1-\left(\frac{h_y.N_{u_y}{}^{(m)}}{2.N_{u_{yy}}{}^{(m)}}\right)_{1,i}\right\} \tag{A-73b}$$

NOTE: Unless otherwise indicated, ALL nonlinear functionals (*N* & its derivatives) are evaluated at the advanced time step, *n+1*. For a **nonlinear problem with a <u>nonlinear</u> Dirichlet bottom boundary condition**, we consider the expansion in (A-68) to only the 2nd term:

$$B_u{}^{(m)}.\delta u^{(m)} \cong f_B{}^{n+1} - B^{(m)} \tag{A-74}$$

and the bottom grid points are **<u>assigned</u>** as follows:

$$\delta u_{1,i}{}^{(m)} \cong \left(\frac{f_B{}^{n+1} - B^{(m)}}{B_u{}^{(m)}}\right)_{1,i} \tag{A-75a}$$

For a **nonlinear problem with a <u>linear or nonlinear</u> Dirichlet bottom boundary condition**, this reduces to:

$$\delta u_{1,i}{}^{(m)} = 0. \qquad \text{for all } m > 0 \tag{A-75b}$$

Irrespective of the linearity of the boundary condition, if the PDE is nonlinear, all functional values for the first iteration ($m = 0$, according to the notation used here) have to be evaluated at the previous time level in order to take into account the time dependence of the Dirichlet condition. This also follows naturally from the fact that the first guess for the advanced time step is the converged value at the end of the last time step. If these were evaluated at the advanced time level $n+1$, then the boundary value will remain the same as at t = t_0. So, $u^{(0)} = u^n$, $B^{(0)} = B^n$, and $B_u^{(0)} = B_u^n$:

$$\delta u_{1,i}{}^{(0)} = u_{1,i}{}^{(1)} - u_{1,i}{}^{(0)} \cong \left(\frac{f_B{}^{n+1} - B^n}{B_u{}^n}\right)_{1,i} \tag{A-76}$$

It must be kept in mind that for the particular class of problems being considered, as shown in Equation (A-3), the boundary condition takes on the form of a generalized Robin BC:

$$B(u,u_y) = B_1(u) \,.\, u_y + B_2(u) \tag{A-77}$$

In this case, Equations (A-67)-(A-73) and (A-75)-(A-78) can be modified accordingly and everything expressed in terms of B_1 and B_2.

<u>Spherical coordinate system:</u> Now, the form of Equations (A-73) is identical for both Cartesian and Cylindrical coordinate systems. But for spherical coordinates, the PDE is not analytic as $y \to 0$, due to the presence of the function *Sin(y)* in the denominator of b_1. In this case, the PDE becomes (analogous to (A-34) above), after applying L'Hospital's rule to the y-component of Equation (A-15a) as $y \to 0$

$$\frac{\partial u}{\partial t} = N_S = \left[\frac{\left\{ k_t.\left(a_1 a_{2,x}.u_x + a_1 a_2.u_{xx} + 2b_1 b_2.u_{yy} \right) + k_{t,u}.\left(a_1 a_2.u_x{}^2 + b_1 b_2.u_y{}^2 \right) \right\} + f}{\rho_0 c_P} \right] \tag{A-78a}$$

Again, we have assumed the symmetry condition $\boldsymbol{u_y(\theta=0) = 0}$. So, Equation (A-73a) is still applicable in the x direction (since the x-derivative terms remain unchanged from Eq. (A-34a), except that N must be replaced by N_S), but not Equation (A-73b). In this case, the derivatives required in the indicial form of (A-78a) (equivalent to Equations (A-15)) are:

$$N_{S,u} = \left[\frac{\left\{ \left(k_{t,u}.c_P - k_t.c_{P,u} \right).\left(\frac{2.u_x}{x} + u_{xx} + \frac{2.u_{yy}}{x^2} \right) + \left(k_{t,uu}.c_P - k_{t,u}.c_{P,u} \right).\left(u_x{}^2 + \frac{u_y{}^2}{x^2} \right) \right\} + \left(f_u.c_P - f.c_{P,u} \right)}{\rho_0 c_P{}^2} \right] \tag{A-78b}$$

$$N_{S,u_x} = \frac{2.\left(\frac{k_t}{x} + k_{t,u}.u_x \right)}{\rho_0 c_P} = N_{u_x} \tag{A-78c}$$

$$N_{S,u_y} = \frac{2.k_{t,u} u_y}{\rho_0 c_P x^2} \tag{A-78d}$$

$$N_{S,u_{xx}} = \frac{k_t}{\rho_0 c_P} = N_{u_{xx}} \tag{A-78e}$$

$$N_{S,u_{yy}} = \frac{2.k_t}{\rho_0 c_P x^2} \tag{A-78f}$$

since a_1 x $a_2 = 1$, a_1 x $a_{2,x} = 2/x$, b_1 x $b_2 = 1/x^2$, in spherical coordinates. NOTE: We do not consider the case when $x = 0$ since it has already been considered under the left boundary condition. So, at $y = 0$, and for $x \neq 0$, the implementation of the PDE (A-73) becomes:

$$\left\{ 1 - \left(\frac{h_x.N_{S,u_x}{}^{(m)}}{2.N_{S,u_{xx}}{}^{(m)}} \right)_{1,i} \right\} \delta v_{1,i-1}{}^{(m)} - \left\{ 2 + \left(\frac{4 - k.N_{S,u}{}^{(m)}}{4\rho_x.N_{S,u_{xx}}{}^{(m)}} \right)_{1,i} \right\} \delta v_{1,i}{}^{(m)} + \left\{ 1 + \left(\frac{h_x.N_{S,u_x}{}^{(m)}}{2.N_{S,u_{xx}}{}^{(m)}} \right)_{1,i} \right\} \delta v_{1,i+1}{}^{(m)} = - \frac{\left\{ u^n + \frac{k}{2}\left(N_S{}^{(m)} + N_S{}^n \right) \right\}_{1,i} - u_{1,i}{}^{(m)}}{\rho_x.N_{S,u_{xx}\,1,i}{}^{(m)}} \tag{A-79a}$$

$$- \left\{ 2 + \left(\frac{4 - k.N_{S,u}{}^{(m)}}{4\rho_y.N_{S,u_{yy}}{}^{(m)}} \right)_{1,i} - \left(\frac{2h_y.B_u{}^{(m)}}{B_{u_y}{}^{(m)}} \right)_{1,i} . \left\{ 1 - \left(\frac{h_y.N_{S,u_y}{}^{(m)}}{2.N_{S,u_{yy}}{}^{(m)}} \right)_{1,i} \right\} \right\} \delta u_{1,i}{}^{(m)} + 2\delta u_{2,i}{}^{(m)} = - \frac{\delta v_{1,i}{}^{(m)}}{\rho_y.N_{u_{yy}\,1,i}{}^{(m)}} + 2h_y.\left(\frac{f_B - B^{(m)}}{B_{u_y}{}^{(m)}} \right)_{1,i} \left\{ 1 - \left(\frac{h_y.N_{S,u_y}{}^{(m)}}{2.N_{S,u_{yy}}{}^{(m)}} \right)_{1,i} \right\} \tag{A-79b}$$

The linear versions of Equations (A-79) can be deduced as a special case, by using the linear boundary conditions (Equation (A-72') and setting $N_{S,u} = 0$ *and* $N_{S,ux} /2.\ N_{S,uxx} = 1/x$, from Equations(A-78):

$$\left\{1-\frac{h_x}{x_i}\right\}\delta v_{1,i-1}{}^{(m)}-\left\{2+\left(\frac{1}{\rho_x.N_{S,u_{xx}}{}^{(m)}}\right)_{1,i}\right\}\delta v_{1,i}{}^{(m)}+\left\{1+\frac{h_x}{x_i}\right\}\delta v_{1,i+1}{}^{(m)}=-\frac{\left\{\frac{k}{2}\left(N_S{}^{(m)}+N_S{}^{n}\right)\right\}_{1,i}}{\rho_x.N_{S,u_{xx}\,1,i}{}^{(m)}}$$ (A-79a')

$$-\left\{2+\left(\frac{1}{\rho_y.N_{S,u_{yy}}{}^{(m)}}\right)_{1,i}-2h_y.\alpha_y\right\}\delta u_{1,i}{}^{(m)}+2\delta u_{2,i}{}^{(m)}=-\frac{\delta v_{1,i}{}^{(m)}}{\rho_y.N_{S,u_{yy}\,1,i}{}^{(m)}}+2h_y.(f_{B,i}{}^{n+1}-f_{B,i}{}^{n})$$ (A-79b')

Finally we can determine the values of the derivatives along the bottom boundary, excluding the corner points (corner points were considered separately under the left and right boundary conditions), *i.e.*, $i = 2, 3, \ldots, N_x - 1$:

$$(u_x)_{1,i}{}^{(m)}\cong(D_{0,x})_{1,i}{}^{(m)}=\frac{u_{1,i+1}{}^{(m)}-u_{1,i-1}{}^{(m)}}{2.h_x}\qquad\text{(A-80)}$$

$$(u_{xx})_{1,i}{}^{(m)}\cong(D_{0,x}{}^{2})_{1,i}{}^{(m)}=\frac{u_{1,i+1}{}^{(m)}-2.u_{1,i}{}^{(m)}+u_{1,i-1}{}^{(m)}}{h_x{}^{2}}\qquad\text{(A-81)}$$

$$(u_y)_{1,i}{}^{(m)}\cong(D_{0,y})_{1,i}{}^{(m)}=\frac{u_{2,i}{}^{(m)}-u_{0,i}{}^{(m)}}{2.h_y}=\left(\frac{f_B{}^{n+1}-B_2{}^{(m)}}{B_1{}^{(m)}}\right)_{1,i}\qquad\text{(A-82)}$$

Therefore,

$$u_{0,i}{}^{(m)}=u_{2,i}{}^{(m)}-2.h_y\left(\frac{f_B{}^{n+1}-B_2{}^{(m)}}{B_1{}^{(m)}}\right)_{1,i}\qquad\text{(A-83)}$$

and,

$$(u_{yy})_{1,i}{}^{(m)}\cong(D_{0,y}{}^{2})_{1,i}{}^{(m)}=\frac{u_{2,i}{}^{(m)}-2.u_{1,i}{}^{(m)}+u_{0,i}{}^{(m)}}{h_y{}^{2}}=\frac{2.u_{2,i}{}^{(m)}-2.u_{1,i}{}^{(m)}-2.h_y\left(\frac{f_B{}^{n+1}-B_2{}^{(m)}}{B_1{}^{(m)}}\right)_{1,i}}{h_y{}^{2}}\qquad\text{(A-84)}$$

As with the left and right boundaries, all derivatives at <u>time level *n*</u> can be obtained by replacing the iteration superscript *(m)* by the time level superscript, *n*, and then changing all f^{n+1} to f^n, in Equations (A-80)-(A-84). In the linear non-Dirichlet cases, the following substitutions will make Equations (A-80)-(A-84) consistent: (a) <u>*Linear Neumann*</u> – $\boldsymbol{B_2}$ **= 0** (zero), and $\boldsymbol{B_1}$ **= 1**, and (b) <u>*Linear Robin*</u> – $\boldsymbol{R_2 = \alpha_y.u^n}$, and $\boldsymbol{B_1}$ **= 1**. In addition, for these cases, all RHS terms containing u are evaluated at time level n. Therefore, ***f_B is evaluated at time level n*** (instead of at $n+1$). This completes the derivations for the <u>***bottom boundary***</u>.

A-2.3.4 Top Boundary

The derivations for the top boundary closely follow those for the bottom boundary in the previous section. Again, except for the spherical coordinate system, we <u>do not consider the corner points</u> here since they were considered under the left and right boundaries described above. Consider the general nonlinear BC presented in Equation (A-2) above:

$T(u,u_y) = f_T(x,t_{n+1})$ (A-85)

If the BC is non-Dirichlet, it can be linearized by expanding the LHS functional about the previous iterate, to the third term in the Frechet-Taylor's series, to get:

$$T^{(m)} + T_u{}^{(m)}.\delta u^{(m)} + T_{u_y}{}^{(m)}.\delta u_y{}^{(m)} \cong f_T{}^{n+1} \quad \text{(A-86)}$$

Rearranging,

$$\left(T_u{}^{(m)} + T_{u_y}{}^{(m)} D_{0,y}\right)\delta u^{(m)} \cong f_T{}^{n+1} - T^{(m)} \quad \text{(A-87)}$$

Expanding the centered difference approximation, we can obtain an estimate for the value of the image point, $\delta u_{Ny+1,i}{}^{(m)}$, and thus, be able to solve the split step equations (A-16), at the bottom boundary. Substituting the expression for $D_{0,y}$ into Equation (A-87), we get:

$$\left[\left\{T_u{}^{(m)}.\delta u\right\}_{N_y,i}{}^{(m)} + \left\{T_{u_y}{}^{(m)}\right\}_{N_y,i}\left(\frac{\delta u_{N_y+1,i}{}^{(m)} - \delta u_{N_y-1,i}{}^{(m)}}{2.h_y}\right)\right] \cong f_{T,i}{}^{n+1} - T_{N_y,i}{}^{(m)} \quad \text{(A-88)}$$

Rearranging (A-88), we get:

$$2h_y\left\{T_u{}^{(m)}.\delta u\right\}_{N_y,i}{}^{(m)} + \left\{T_{u_y}{}^{(m)}\right\}_{N_y,i}.\delta u_{N_y+1,i}{}^{(m)} - \left\{T_{u_y}{}^{(m)}\right\}_{N_y,i}.\delta u_{N_y-1,i}{}^{(m)} \cong 2h_y\left(f_{T,i}{}^{n+1} - T_{N_y,i}{}^{(m)}\right) \quad \text{(A-89)}$$

We now use the same notation as in Equations (A-16) for the unknown variables at each stage: v for the first stage, and u for the second stage. We thus have, for $i = 2, 3, ..., N_x-1$:

$$\delta u_{N_y+1,i}{}^{(m)} \cong -\left(\frac{2h_y.T_u{}^{(m)}}{T_{u_y}{}^{(m)}}\right)_{N_y,i} \delta u_{N_y,i}{}^{(m)} + \delta u_{N_y-1,i}{}^{(m)} + 2h_y.\left(\frac{f_T{}^{n+1} - T^{(m)}}{T_{u_y}{}^{(m)}}\right)_{N_y,i} \quad \text{(A-90)}$$

For the linear case, we get correspondingly:

$$\delta u_{N_y+1,i}{}^{(m)} \cong -2h_y.\alpha_y.\delta u_{N_y,i}{}^{(m)} + \delta u_{N_y-1,i}{}^{(m)} + 2h_y.(f_{T,i}{}^{n+1} - f_{T,i}{}^{n}) \quad \text{(A-90')}$$

NOTE: α_y is the linear Robin BC parameter (as in: $u_y + \alpha_y.u$), and will be 0 (zero) for the linear Neumann BC. The linear Equation (A-90') was obtained from the nonlinear Equation (A-90) as a special case, by setting $T^{(m)} = f_T^n$, $T_u{}^{(m)} = \alpha_y$,, and $T_{uy}{}^{(m)} = 1$. Setting $j=N_y$ in both (A-16a and b), and substituting (A-90) into Equation (A-16b), we finally get, for the top boundary:

$$\left\{1-\left(\frac{h_x.N_{u_x}{}^{(m)}}{2.N_{u_{xx}}{}^{(m)}}\right)_{N_y,i}\right\}\delta v_{N_y,i-1}{}^{(m)} - \left\{2+\left(\frac{4-k.N_u{}^{(m)}}{4\rho_x.N_{u_{xx}}{}^{(m)}}\right)_{N_y,i}\right\}\delta v_{N_y,i}{}^{(m)} + \left\{1+\left(\frac{h_x.N_{u_x}{}^{(m)}}{2.N_{u_{xx}}{}^{(m)}}\right)_{N_y,i}\right\}\delta v_{N_y,i+1}{}^{(m)} = -\frac{\left\{u^n + \frac{k}{2}\left(N^{(m)} + N^n\right)\right\}_{N_y,i} - u_{N_y,i}{}^{(m)}}{\rho_x.N_{u_{xx}\,N_y,i}{}^{(m)}}$$

(A-91a)

$$2\delta u_{N_y-1,i}^{(m)} - \left\{ 2 + \left(\frac{4 - k.N_u^{(m)}}{4\rho_y.N_{u_{yy}}^{(m)}} \right)_{N_y,i} + \left(\frac{2h_y.T_u^{(m)}}{T_{u_y}^{(m)}} \right)_{N_y,i} . \left\{ 1 + \left(\frac{h_y.N_{u_y}^{(m)}}{2.N_{u_{yy}}^{(m)}} \right)_{N_y,i} \right\} \right\} \delta u_{N_y,i}^{(m)} = -\frac{\delta v_{N_y,i}^{(m)}}{\rho_y.N_{u_{yy}\,N_y,i}^{(m)}} - 2h_y . \left(\frac{f_T - T^{(m)}}{T_{u_y}^{(m)}} \right)_{N_y,i} \left\{ 1 + \left(\frac{h_y.N_{u_y}^{(m)}}{2.N_{u_{yy}}^{(m)}} \right)_{N_y,i} \right\}$$
(A-91b)

For a **nonlinear problem with a <u>nonlinear</u> Dirichlet top boundary condition**, we consider the expansion in (A-86) to only the 2nd term:

$$T_u^{(m)}.\delta u^{(m)} \cong f_T^{\,n+1} - T^{(m)} \quad \text{(A-92)}$$

and the top grid points are **<u>assigned</u>** as follows:

$$\delta u_{N_y,i}^{(m)} \cong \left(\frac{f_T^{\,n+1} - T^{(m)}}{T_u^{(m)}} \right)_{N_y,i} \quad \text{(A-93a)}$$

For a **nonlinear problem with a <u>linear or nonlinear</u> Dirichlet top boundary condition**, this reduces to:

$\delta u_{Ny,i}^{(m)} = 0$ for all $m > 0$ (A-93b)

Irrespective of the linearity of the boundary condition, if the PDE is nonlinear, all functional values for the first iteration ($m = 0$, according to the notation used here) have to be evaluated at the previous time level in order to take into account the time dependence of the Dirichlet condition. This also follows naturally from the fact that the first guess for the advanced time step is the converged value at the end of the last time step. If these were evaluated at the advanced time level $n+1$, then the boundary value will remain the same as at $t = t_0$. So, $u^{(0)} = u^n$, $T^{(0)} = T^n$, and $T_u^{(0)} = T_u^n$:

$$\delta u_{N_y,i}^{(0)} = u_{N_y,i}^{(1)} - u_{N_y,i}^{(0)} \cong \left(\frac{f_T^{\,n+1} - T^n}{T_u^{\,n}} \right)_{N_y,i} \quad \text{(A-94)}$$

It must be kept in mind that for the particular class of problems being considered, as shown in Equation (A-3), the boundary condition takes on the form of a generalized Robin BC:

$T(u,u_y) = T_1(u) \,.\, u_y + T_2(u)$ (A-95)

In this case, Equations (A-85)-(A-94) can be modified accordingly and everything expressed in terms of T_1 and T_2.

<u>Spherical coordinate system:</u> Now, the form of Equations (A-92) is identical for both Cartesian and Cylindrical coordinate systems. But for spherical coordinates, the PDE is not analytic as $\boldsymbol{y \to \pi}$, due to the presence of the function *Sin(y)* in the denominator of b_1. The computation of the functional at the top boundary and deducing the resultant top boundary equations is identical to that for the bottom boundary, and Equations (A-78) and (A-79) can be used for the top boundary, after changing the y-index to N_y, instead of *1*.

Finally we can determine the values of the derivatives along the top boundary, excluding the corner points (corner points were considered separately under the left and right boundary conditions), *i.e.*, $i = 2, 3, .., N_x - 1$:

$$(u_x)_{N_y,i}^{(m)} \cong (D_{0,x})_{N_y,i}^{(m)} = \frac{u_{N_y,i+1}^{(m)} - u_{N_y,i-1}^{(m)}}{2.h_x} \tag{A-96}$$

$$(u_{xx})_{N_y,i}^{(m)} \cong (D_{0,x}^{\ 2})_{N_y,i}^{(m)} = \frac{u_{N_y,i+1}^{(m)} - 2.u_{N_y,i}^{(m)} + u_{N_y,i-1}^{(m)}}{h_x^{\ 2}} \tag{A-97}$$

$$(u_y)_{N_y,i}^{(m)} \cong (D_{0,y})_{N_y,i}^{(m)} = \frac{u_{N_y+1,i}^{(m)} - u_{N_y-1,i}^{(m)}}{2.h_y} = \left(\frac{f_T^{\ n+1} - T_2^{\ (m)}}{T_1^{\ (m)}}\right)_{N_y,i} \tag{A-98}$$

Therefore,

$$u_{N_y+1,i}^{(m)} = u_{N_y-1,i}^{(m)} + 2.h_y\left(\frac{f_T^{\ n+1} - T_2^{\ (m)}}{T_1^{\ (m)}}\right)_{N_y,i} \tag{A-99}$$

and,

$$(u_{yy})_{N_y,i}^{(m)} \cong (D_{0,y}^{\ 2})_{N_y,i}^{(m)} = \frac{u_{N_y+1,i}^{(m)} - 2.u_{N_y,i}^{(m)} + u_{N_y-1,i}^{(m)}}{h_y^{\ 2}} = \frac{-2.u_{N_y,i}^{(m)} + 2.u_{N_y-1,i}^{(m)} + 2.h_y\left(\frac{f_T^{\ n+1} - T_2^{\ (m)}}{T_1^{\ (m)}}\right)_{N_y,i}}{h_y^{\ 2}} \tag{A-100}$$

As with the bottom boundary, all derivatives at <u>time level *n*</u> can be obtained by replacing the iteration superscript *(m)* by the time level superscript, *n*, and then changing all f^{n+1} to f^n, in Equations (A-98)-(A-102). In the linear non-Dirichlet cases, the following substitutions will make Equations (A-98)-(A-102) consistent: (a) <u>*Linear Neumann*</u> – $\boldsymbol{T_2}$ **= 0** (zero), and $\boldsymbol{T_1}$ **= 1**, and (b) <u>*Linear Robin*</u> – $\boldsymbol{T_2 = \alpha_y . u^n}$, and $\boldsymbol{T_1}$ **= 1**. In addition, for these cases, all RHS terms containing *u* are evaluated at time level *n*. Therefore, ***f_T* is evaluated at time level n** (instead of at *n+1*). This completes the derivations for the ***<u>top boundary</u>***.

A-2.4 Computational procedure summary

At each time level, the coefficients of the tridiagonal systems (Equations (16), (26), (35), (52), (73), (79), (91)) are first computed using an initial guess for ***u*** (converged value at the previous time step or initial condition). The tri-diagonal system of equations involving both the interior and boundary points can be solved by an LU-Decomposition scheme, once in each of x- and y-directions, to get a new iterate. Then new coefficients based on the last iterate are computed to generate subsequent iterates. This process is continued until the difference in the norms of two successive iterates becomes smaller than a specified tolerance. Once convergence is achieved at a time level, the algorithm moves to the next one, taking this value as the initial guess for that level. Section A-2.5 below outlines the algorithm for implementing this procedure. A detailed explanation for the code is given in Chapter A-3.

A-2.4.1 Algorithm for Implementation

- **Load/Specify the following (INPUTS):**
  - Flags for problem specification: *linear_flag* (problem linearity specification), *coord_flag* (geometry specification), *smooth_flag* (type of smoothing – None/1D/2D), *exact_sol_flag* (whether exact solution is known); Boundary condition (BC) type flags – *left_bc_flag, right_bc_flag, bottom_bc_flag, top_bc_flag*; BC linearity flags – *left_lin_flag, right_lin_flag, bottom_lin_flag, top_lin_flag*;
  - PDE Specification: Initial Condition - u_0; Coefficients of adjoint form of PDE (for user specified problem geometry, other than standard Cartesian, Cylindrical or Spherical systems: *coord_flag = 0*) – a_1, a_2, b_1, b_2; Linear or nonlinear functionals, L or N, and their derivatives w.r.t. temperature and its derivatives - $u, u_x, u_y, u_{xx}, u_{yy}$. Expressions for BCs – f_L, f_R, f_B, f_T; BC functionals that define the Left Hand Side (LHS) of the BC – $Lf_1, Lf_2, R_1, R_2, B_1, B_2, T_1, T_2$, and their derivatives w.r.t u, u_x, u_y; linear/nonlinear Right Hand Side (RHS) function or Source function of PDE – f_{rhs}, and its derivative w.r.t. temperature, u.
  - Problem data: Values of thermal and elastic properties of rock and fault surfaces being modeled – Thermal conductivity, k_t, Specific Heat, C_p, Density, ρ, Young's Modulus of elasticity, E, Poisson's ratio, ν, Coefficient of friction, μ, Shear stress, τ, Asperity radius, r_0, Slip velocity, V_{slip}, Angular contact, θ_0, and Contact duration, t_0. Expressions for nonlinear variation of these properties with temperature (if variation is significant, and or relevant), and their derivatives (as required) – for instance, $k_t(u), k_{t,u}, k_{t,uu}, C_p(u), C_{p,u}$. Smoothing flag - *smooth_factor*, if smoothing flag was non-zero.
  - Spatio-temporal domain boundaries – x_l, x_r, y_t, y_b, t_i, and t_f
  - Resolution/Step sizes - h_x, h_y, and k (time step)
  - Newton-Kantorovich (N-K) nonlinear iterations convergence tolerance – *quasi_epsilon*
  - Max allowed N-K iterations – *quasi_iterations*.
  - Output File parameters (for convergence tests and validation plots): Format of each file, Header information, data sampling resolutions, times and locations, output data definition or calculation.

- Main Program – *nonlin_parabolic_pde* - Time Loop:

For t = 1, t_steps

If t > 1 - CALL quasilinear subroutine – *delta_qlin_dgts*

- store the previous time step value u, in u^n
- set the initial grid function guess to the converged value at the end of last time step: $u^{(0)} = u^n$
- Perform Newton-Kantorovich Iterations until convergence:

For iter = 1, quasi_iterations ! NEWTON-KANTOROVICH iteration loop

If iter > 1 - Check for Convergence:

If convergence occurs:

Store relevant data,

Return to Main Program: go to next time step.

Otherwise - Compute next iterate:

For stage = 1, 2 ! DOUGLAS-GUNN x- and y-direction passes

- Call Coeff_RHS routine *qldgts_coeff_rhs*, to compute Coefficients at time level (n+1), using grid function values at the previous iteration.
- Compute RHS vector using both time levels as well as the grid function values at previous time step as well as previous iteration.
- Call the routine *lud_trid* to compute estimate at current stage - δv at the end of stage 1, and δu at the end of stage 2.

Repeat stage

Update grid function values for current iteration: $u^{(m)} = \delta u + u^{(m-1)}$

Compute errors, if exact solution is not known, or error estimates

Store $u^{(m)}$ for use in the next iteration.

Repeat iter

Repeat t

- Print output once marching in time is completed.

A-3. COND2D – FORTRAN 90 CODE DESCRIPTION, SETUP & VALIDATION

A-3.1 Scope of COND2D: Current capabilities, their potential extension, and code limitations

The goal here was to develop a very general, reliable, and modular 2D diffusion code, that can be applied to either linear or nonlinear PDEs, with any combination of linear/nonlinear boundary conditions, and in any geometry, that can be extended without significant modifications to a 3D. *COND2D* is such a general code, and can be applied with minor modifications to any 2D parabolic partial differential equation (PDE). Its different loops and flow pathways have been thoroughly tested using over 35 different linear and nonlinear problems with known solutions of varying complexity and smoothness – with almost all possible combinations of coordinate systems, linear and nonlinear boundary conditions, and parameter ranges. This led to about 10 versions of the code that were successively "purged" (of numerous numerical, input/output, and formatting bugs) to produce this current reliable version. Some details of these validation tests are presented in Section A-3.4 below. It is the author's experience that if this version of the code did not work for a particular problem, more often than not, the issue was with the myriad inputs that the code requires in terms of flags, parameter values, and boundary conditions. Before using *COND2D*, it is recommended that this chapter be carefully read and the organization of the code be understood (Figure 1 and Section A-2.4 above), before trying to implement it for a problem of interest.

Minor modifications – like changing the values of any of a number of parameters and/or modifying the algebraic expressions for various linear/nonlinear functional subroutines in the code - have to be made implementing this code for a problem of interest. In addition, some advanced level (major) modifications that can be made to the code without significant rewriting of the *COND2D* source code are (roughly in increasing order of difficulty, and quantity of additional code to be appended):

- **General Boundary Conditions:** *COND2D* can be made to accept very general boundary conditions, instead of being restricted to only conductive Neumann/Robin conditions. This can be accomplished by a simple change in the expressions for the appropriate (a) boundary condition functionals (e.g., for the left boundary condition, subroutines *lbc1* & *lbc2* may have to be replaced by a single subroutine *lbc*, and appropriate modifications made to existing *lbc_u* and *lbc_ux* subroutines), and (b) boundary condition right hand side (RHS) functions (e.g., subroutine *f_left* for the left boundary). In addition, appropriate changes have to be made to the derivative subroutines, *u_x, u_y, u_xx,* and *u_yy*, as well as to the coefficients and RHS terms for boundary grid points in subroutine *qlindgts_coeff_rhs* (see Section A-3.2 below). The relevant theory for this was discussed in Section A-2.3 above. However, if the boundary conditions for a problem of interest can be cast in the form of Equation (3) (Chapter1), then no changes need to be made to *COND2D*.

- **User Defined Geometry/Coefficients:** *COND2D* can be applied to a geometry different from the three standard coordinate systems (Cartesian, Cylindrical, and Spherical). This can be accomplished by setting the coordinate system flag (*coord_flag*) to 0, and then specifying appropriate expressions for the coefficients of the PDE – a_1, a_2, b_1, b_2. If these expressions are not analytic at some point(s) in the spatial domain, then appropriate modifications need to be made to the subroutine *qlindgts_coeff_rhs* (see Section A-3.2 below). So, this code can be applied to a PDE in other "regular" coordinate systems like: conical, ellipsoidal, elliptic cylindrical, oblate spheroidal, parabolic, parabolic cylindrical, paraboloidal, and prolate spheroidal (in the order of increasing symmetry – see, for instance Moon and Spencer 1988).

- **Including Advection/Transport terms:** *COND2D* can be modified relatively easily to include advection terms in a Conduction/Advection equation. Again, suitable modifications need to be made to the subroutine *qlindgts_coeff_rhs* (see Section A-3.2 below).

- **Parallelizing *COND2D*:** This is an important issue with linear or nonlinear problems with non-smooth data (boundary conditions, source functions, coefficients, etc.) - the more non-smooth the data, the higher the required spatial and/or temporal resolution at which the problem has to be solved. That is, below a certain resolution, the numerical problem is under-resolved, and cannot accurately represent the smaller scale physics characterized by the non-smooth data. This critical resolution has to be determined on a case-by-case basis by testing for grid function convergence with increasing resolution. While the problem is under-resolved, the solution my not be stable and may vary widely with uniform resolution increases. But above the critical resolution, the solution starts converging with increasing resolution (and not necessarily to any of the under-resolved solutions). Parallelization of the code may be required to improve the odds of being able to compute the solution in reasonable time as well as stay within machine array size limits, parallelization is important.

- **Extension from 2D to 3D problems:** *COND2D* can be extended to a parallelizable 3D form, by considering a 3D spatial domain as a stack of 2D domain slices (McDonough and Dong 2001). In this case, each 2D slice can be solved independently of the others at every iteration, and the 2D Douglas-Gunn scheme itself can be parallelized. At each iteration, the original 3D solve is reduced to a 2D solve (which can be carried out with *COND2D*) and a 1D solve, which requires the addition of a loop that is very similar in structure and content to that for each stage of the two level scheme used here, in the subroutine *delta_qlin_dgts* (Section 3.2 below). As shown in the aforementioned reference, the whole process can be efficiently implemented on parallel architecture machines.

- **Extension to systems of 2D or 3D PDEs:** The most complex of adaptations for *COND2D*, involving significant code modifications, involves applying it to systems of PDEs. As shown in McDonough (2002), the underlying linear algebra is similar but more general, in that, at each grid point of the domain we have to solve for a system of variables, instead of a single variable.

After any of the modifications suggested above are made to the code, and compilation errors corrected, the code has to be re-validated using a problem with a known solution, to test the modified parts of the code, as illustrated below in Section A-3.4.

Needless to say, the algorithm used here, and therefore COND2D, has a number of limitations:

- **Irregular geometry:** One major limitation is that of the finite difference approach itself: it cannot easily accommodate irregular or complex geometries that cannot be mapped (one-to-one) to a rectangular grid. In this case, a number of tricks may be used. For instance, some form of domain splitting can be implemented to create a number of subdomains of simple geometry, and then applying the code to these different subdomains. Of course, when the problem domain is split into subdomains, another level of iterations has to be introduced to ensure compatibility of solutions at the boundaries of these subdomains, while satisfying the overall boundary conditions of the problem. This would definitely involve not only a complete rewriting of parts of the current code, but also adding additional modules and "book-keeping" subroutines.

- **Symmetry requirements:** Another limitation of this code is the symmetry requirements on the solution at $r = 0$, for both spherical and cylindrical coordinate systems, and at $\theta = 0$ or π, for the spherical system. If the symmetry requirements cannot be assumed, L'Hospital's rule approximations cannot be made to the PDE at these non-analytic points and no solution can be computed ***at*** those points.
- **Storage:** *COND2D* uses a number of storage variables, so that all relevant data sampled at different time levels can be output at one time, at the completion of the "time-marching". This was done to minimize file writes, which are very inefficient. However, this limits the resolution at which the code can be run – especially on a shared machine like the HP Superdome supercomputer cluster on the University of Kentucky campus - due to the overall memory allocation limits (cache limits) for each user.

A-3.1.1 Organization of the source code

As described above, *COND2D* was developed as a highly modular code to provide users with a lot of flexibility in defining and setting up 2D heat conduction problems. A self-explanatory organizational and data flow chart of the code appears in Figure A- 1. It is suggested that this figure be used in conjunction with the procedure description and algorithm outline presented in Section A-2.4 above, and the example run setup illustrated in Section A-3.3. A description of contents of the code appears in the following section.

A-3.2 Brief description of modules, subroutines and key variables

The *COND2D* source code contains a large number of comment statements and the user is referred to it for any specific details. The objective of this section is to provide a brief overview of each subroutine, define its input and output variables, and discuss the importance of certain key variables that require user input, within the subroutine where they are encountered first. Section A-3.3 actually goes through the process of setting up a run, compiling the code and running it. In the subsections that follow, all modules are briefly described, and key variables are discussed where appropriate. A table (or tables) listing and describing the key variables in that module (or each individual subroutine in that module) is (are) also presented, if needed. Use of this section in conjunction with Section A-2.4 and Figure A-1 is recommended.

A-3.2.1 MODULE const_params

This module specifies constants and sets the values of machine limit parameters needed by the rest of the subroutines, basic partial differential equation (PDE) flags, output file unit numbers and names, specifies output sampling point information, and defines global variables. It is important to check this module over carefully before running the code as it contains several key parameters for the run – from the very definition of the type of PDE, to whether it has an exact solution, to PDE domain definition and minimum run resolution, to output sampling points and output resolution – that have to be set by the user. Its the author's experience that in cases where *COND2D* does not work for a specific problem, more often than not the issue was misrepresentation/overlooking of a parameter value within this module. It is recommended that the user follow a suitable data checking procedure before attempting to run the code, given the number parameters that may need to be modified for a given problem. Key variables in this module are described in Table A-2.

Figure A- 1. Organizational chart for *COND2D*. Refer Section A-2.4 for an outline of the algorithm

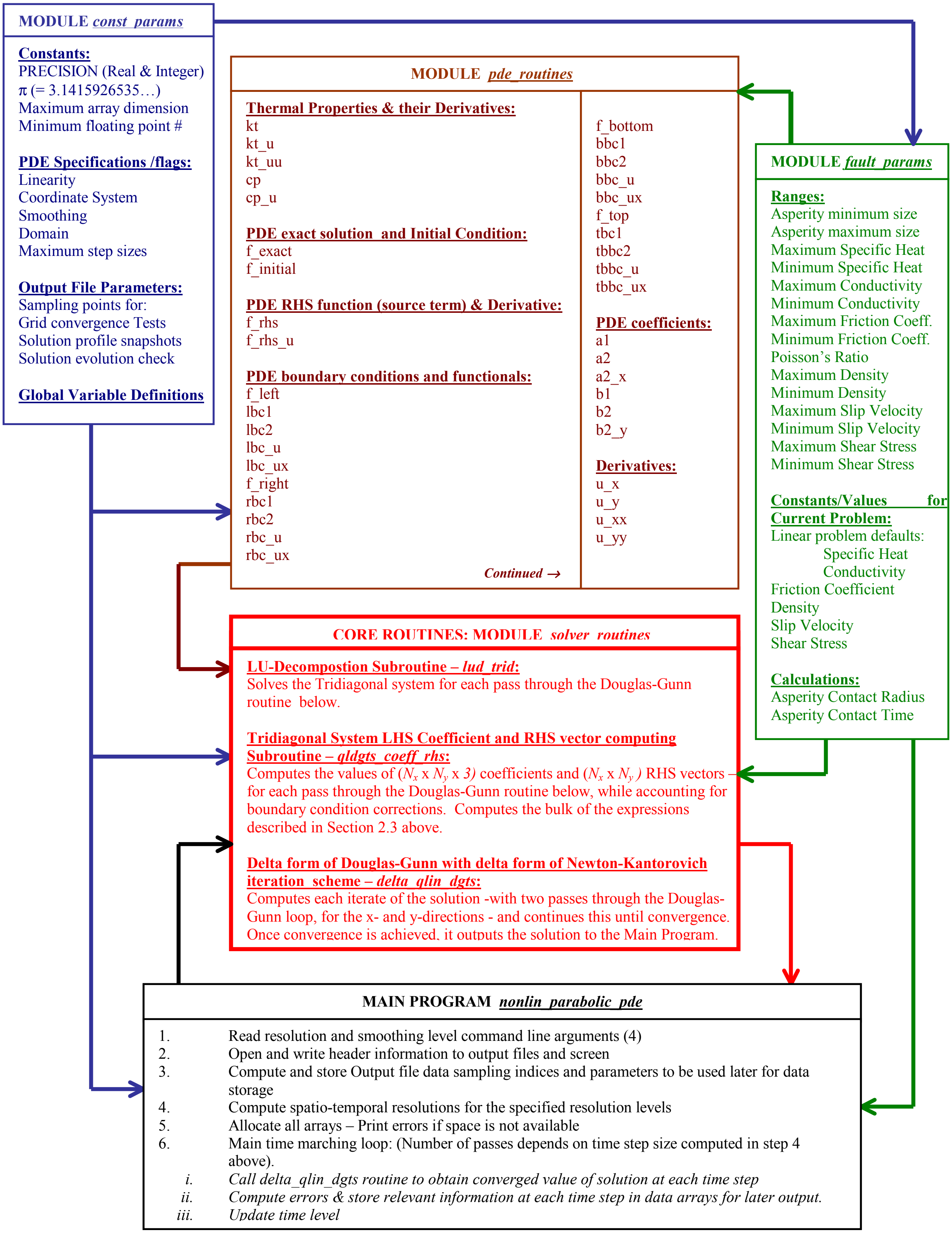


Table A- 2. Key variables in MODULE *const_params*.

Variable	Description or Comment
rp	Precision of all real variables and constants in the run
ip	Precision of all integer variables and constants in the run
out	Array containing output file unit numbers (5 for the present implementation)
outfile	Array containing the names of output files (See Section A-3.2.5 for description) (*GRID*, *ERROR*, *SNAP*, *EVOLUTION*, and *CONVERGENCE*)
max_points	Machine array dimension limit – This is the maximum number of grid points permitted in each spatial direction in *COND2D*.
epsilon	Smallest numerical approximation to zero – useful sometimes in avoiding floating point exceptions (or divide-by-zero errors).
linear_flag	= 1 if the PDE of interest is linear; = 0 if nonlinear.
coord_flag	= 0 if user defined system (see section 3.1 above for code modifications in this case); = 1 if the coordinate system of interest is Cartesian; = 2 if Cylindrical; = 3 if Spherical;
smooth_flag (DEFINED ONLY)	*COMMAND LINE ARGUMENT # 3.* = 0 if no smoothing of grid functions is required (in case of non-smooth data); = 1 for 1D smoothing; = 2 for 2D smoothing.
smooth_factor (DEFINED ONLY)	*COMMAND LINE ARGUMENT # 4.* Range 000000-999999. Degree of smoothing is non-zero if *smooth_flag* is non-zero (see under Main Program, Section A-3.2.5 for a description).
x_left	Domain left boundary coordinates.
x_right	Domain right boundary coordinates.
y_bottom	Domain bottom boundary coordinates.
y_top	Domain top boundary coordinates.
t_initial	Initial/start time of run.
t_final	Final/end time of run.
hx_max	Maximum x-step size (Minimum resolution in x-direction)
hy_max	Maximum y-step size (Minimum resolution in y-direction)
out_x_grid_spacing	x-direction resolution in the *GRID* and *ERROR* output files.
out_y_grid_spacing	y-direction resolution in the *GRID* and *ERROR* output files.
t_evol_spacing	Output temporal resolution in the temperature *EVOLUTION* output file.
t_snap	Array containing time levels at which *GRID* and *ERROR* data are output.
y_xsnap, t_xsnap	Y-coordinate and time level for snapshot of a solution profile parallel to the x-axis
x_ysnap, t_ysnap	X-coordinate and time level for snapshot of a solution profile parallel to the y-axis
x_time, y_time	X- and Y-coordinates for a single temperature plot data (output to *EVOLUTION* file)
grid_conv	2D Array containing X- and Y-coordinates as well as time levels at which grid convergence tests have to be performed (to be output to *CONVERGENCE* file)
verbose_flag	= 0 if no diagnostic screen output is needed; = 1 if diagnostic screen output - containing the number of nonlinear iterations to convergence, maximum and minimum temperatures, and maximum error (if computable), as well as their grid locations – is needed.
quasi_epsilon (DEFINED ONLY)	SPECIFIED IN *MAIN PROGRAM (Section 3.2.5).* Convergence tolerance for nonlinear iterations, chosen as the cube of time step size, k^3 (see McDonough 2002).
Quasi_iterations (DEFINED ONLY)	SPECIFIED IN *MAIN PROGRAM (Section 3.2.5).* Maximum number of nonlinear (Newton-Kantorovich) iterations allowed for the run – typically a low number (10-15 or lower).

A-3.2.2 MODULE fault_params

This module specifies fault and rock material parameters to be used in the run. Data in this module are derived from (or from fits to thermal property data in) Touloukian et al. (1981) or Byrelee (1978), Logan and Teufel (1986), and Nadeau and Johnson (1998). All the variables in this module are specified in Figure A-1, and the relevant data appears in Appendix C of Kanda (2003). Therefore, no data table appears in this section. This module can be modified by the user in accordance with problem requirements. It is, however, recommended that the information in the source code and the aforementioned appendix be reviewed before modifying the default data or ranges in this module. Just as a reminder, the angular area of contact, θ, is approximated by the expression for Hertzian (elastic) contact between two spheres (Timoshenko and Goodier 1970), and is given by:

$$\theta = TAN^{-1}(r_c /r_0) \cong (r_c /r_0) = \{3.\pi.(1-\nu^2).\tau\}/\{4.E_Y.\mu\}$$

where r_c is the radius of the asperity contact surface, r_0 is the asperity radius, ν is the Poisson's ratio, τ is the shear stress at the contact surface, E_Y is the Young's modulus for the rock material, and μ is the coefficient of friction. The duration of asperity contact is computed as: $t_0 = 4.r_c /V_{slip}$. Sources for the ranges of values for the above parameters are presented in Appendix C of Kanda (2003).

A-3.2.3 MODULE pde_routines

This module contains all the subroutines needed to define the PDE – nonlinear thermal properties, exact solution, initial condition, RHS or source function and its derivatives, all four boundary LHS functionals and RHS functions, PDE coefficients and their derivatives, and first and second derivatives of temperature. This is also a module that can be extensively modified to suit the user's needs. Extreme care must be taken, however, in making sure that all the parameters and expressions that the user modifies in this module, to implement a problem of interest, are accurately represented. Its the author's experience that in cases where *COND2D* does not work for a specific problem, more often than not the issue was misrepresentation of an expression or a sign in an expression within this module. It is recommended that the user follow a suitable quality control and data checking procedure before attempting to run the code, given the number of subroutines that may need to be modified for a given problem. Each of the subroutines in this module is briefly described below, along with any key variables that the user may need to modify. Since the number of variables in each routine is fairly small, no tables are included in this section.

A-3.2.3.1 Thermal conductivity & its derivatives: ***kt, kt_u, kt_uu***

The data and the final functional relationship chosen for the thermal dependence of thermal conductivity are presented in Appendix C of Kanda (2003). Since the coefficients of the tri-diagonal system - defined in Sections A-2.2 & A-2.3 above - are themselves dependent on the nonlinear functional *N* and its derivatives (and therefore, on the temperature, *u*), the stability of the scheme is strongly dependent on the type of thermal property temperature dependencies chosen, and has to be dealt with on a case-by-case basis. No amount of testing will guarantee the stability of the non-linear problem. However, as discussed in Chapter A-1 above, a "rule of thumb" criterion is to make sure that these temperature dependencies are Lipschitz continuous in the expected temperature range of the problem. It was with these considerations that an exponential relationship was chosen for the maximum temperature range of the problem (300 K to 3000 K, or above).

A-3.2.3.2 Specific Heat & its derivative: cp, cp_u

The data and the final functional relationship chosen for the thermal dependence of specific heat are presented in Appendix C of Kanda (2003). Since the coefficients of the tri-diagonal system - defined in Section A-2.2 & 2.3 above - are themselves dependent on the nonlinear functional *N* and its derivatives (and therefore, on the temperature, *u*), the stability of the scheme is strongly dependent on the type of thermal property temperature dependencies chosen, and has to be dealt with on a case-by-case basis. No amount of testing will guarantee the stability of the non-linear problem. However, as discussed in Chapter A-1 above, a "rule of thumb" criterion is to make sure that these temperature dependencies are Lipschitz continuous in the expected temperature range of the problem. It was with these considerations that an exponential relationship was chosen for the maximum temperature range of the problem (300 K to 3000 K, or above).

A-3.2.3.3 Exact solution: ***f_exact*** (Optional)

This routine is for test problems, in which case, a known exact solution can be input to *COND2D*, so it can compute exact errors. Exact errors are used to conduct convergence tests. The presence of an exact solution is indicated by setting the value of *exact_sol_flag* to 1 in the module *const_params* above. If its value is 0 (zero), then the program assumes that there is no exact solution, and does not call this routine. In case of nonlinear problems, it estimates an error, based on iteration errors.

A-3.2.3.4 PDE Initial Condition: ***f_initial***

This routine specifies the initial condition to a problem and is required for every problem. Note that the initial condition needs to be defined over the entire spatial domain of the problem.

A-3.2.3.5 PDE RHS or source function and its derivative: ***f_rhs***

These routines define the linear or nonlinear RHS or source function of the PDE, and its derivative. *f_rhs* can be easily computed for a test problem having a known solution – by direct substitution of that solution into the PDE. For problems of interest to scientists and engineers, when exact solutions are rarely known, it has to be based on the physics of the problem. For heat conduction problems, its units are energy per unit volume (for 3D problems) or energy per unit area (for 2D problems). An example is the radiogenic heat source in the lithosphere.

A-3.2.3.6 Left boundary condition (LBC): RHS function, and LHS functional & derivatives: ***f_left, lbc1, lbc2, lbc_u, lbc_ux***

Required for all problems, these routines help define the form of the left boundary condition. Each boundary condition consists of two components – an LHS functional and an RHS function, as illustrated below for the current implementation of *COND2D*:

$$Lf(u,u_x) \equiv Lf_1(u) + Lf_2(u) \,.\, u_x = f_{left}(y,t)$$

In the above standard form of the left boundary condition for general conduction problems (Equation (3), Chapter A-1), Lf, Lf_1 and Lf_2 are the LHS functionals, and f_{left} is the RHS function. For a general Dirichlet BC, $Lf_2 = 0$; for a general Neumann BC, $Lf_1 = 0$; and for a general Robin BC, both are non-zero functions of the temperature, *u*. By definition, the RHS function does not depend on the temperature, *u*, but only on the y coordinate and time level. While the form of the functionals are defined by the type of boundary heat source/sink – conductive, convective, radiative or assorted combinations – the RHS function is fixed/specified by the user, and takes the form of an analytical expression, or a single constant value. So, Lf_1 needs to be specified if the left boundary condition is radiative ($Lf_1 \propto u^4$) or convective ($Lf_1 \propto h(u).\ u$). Similarly, Lf_2 needs to be specified if the left boundary condition is conductive ($Lf_2 \propto k_t(u)$) or a combination of conductive and convective/radiative sources/sinks (Robin BC). The corresponding linear cases are: Dirichlet – $Lf_1 = u, Lf_2 = 0$; Neumann – $Lf_1 = 0, Lf_2 = 1$; and Robin – $Lf_1 = \alpha_x.u, Lf_2 = 1$. In order to incorporate a radiative BC in a linear problem, a nonlinear equation solver (Newton method) has to be used to compute the value of the temperature, thereby converting it to a Dirichlet condition. Once the form of the functional is fixed, computation of its derivatives w.r.t. temperature is straightforward, and these expressions have to be included in the appropriate subroutine.

For the current implementation of *COND2D*, $Lf_2 = k_t(u)$ for either nonlinear Neumann or nonlinear Robin left boundary condition, and $Lf_1 = u(1+u)/2$ (arbitrary function) for either nonlinear Dirichlet or nonlinear Robin left boundary condition.

A-3.2.3.7 All other boundary conditions (RBC, BBC, & TBC): RHS functions, and LHS functionals & derivatives: ***f_right, rbc1, rbc2, rbc_u, rbc_ux, f_bottom, bbc1, bbc2, bbc_u, bbc_uy, f_top, bbc1, bbc2, bbc_u, bbc_uy***

The treatment of the rest of the boundary conditions is identical to that for the left boundary condition described in Section A-3.2.3.6 above.

A-3.2.3.8 PDE coefficients and their derivatives: ***a1, a2, a2_x, b1, b2, b2_y***

These coefficients have been defined in Table A-1 above, for *coord_flag = 1-3*. If *coord_flag = 0*, then the user has to specify expressions for these coefficients in terms of the coordinate system and time. NOTE: These coefficients are not dependent on temperature, *u*, and therefore, cannot be nonlinear by definition. Expressions for the derivatives must be included in the subroutines *a2_x* and *b2_y*, if *coord_flag = 0.*

A-3.2.3.9 Temperature Derivatives: ***u_x, u_y, u_xx, u_yy***

The derivatives are all computed as discussed in Section A-2.3 above. They are used in the computation of the coefficients and RHS vector of the tridiagonal system to be solved at each pass of the Douglas-Gunn algorithm described in Sections A-2.2 and A-2.3. If the form of any boundary condition functional is changed (as when *coord_flag = 0*, or if more general boundary conditions are used), then expressions for these derivatives must be changed. The procedure outlined in Section A-2.3 can be used to compute these new expressions.

***A-3.2.4 MODULE solver_routines*: The core routines**

This module contains the main driver and 2 workhorse routines of *COND2D*, and is the main numerical computation kernel. Unless modifications listed in Section A-3.1 above are being made, no routine in this module needs user modifications. Thus, almost all 2D pure heat conduction problems can be solved with appropriate minor modifications to the modules *const_params*, *fault_params* and *pde_routines*. This structure minimizes the chances of accidental modification/deletion of any key core numerical components of *COND2D* (see also Figure A-1 above).

A-3.2.4.1 LU Decomposition for tridiagonal systems: lud_trid

This routine solves the tridiagonal system (see, for instance, McDonough 2001) generated by the discretization of the general nonlinear 2D diffusion/transport equation, discussed in Sections A-2.2 & A-2.3. It is called at every pass of the two stage Douglas-Gunn loop, which itself occurs twice per nonlinear iteration (for the 2D problem). *lud_trid* solves a system of linear equations (any number, up to machine memory limit):

$\boldsymbol{A}.\boldsymbol{x} = \boldsymbol{b}$

where $\boldsymbol{A}$ is a "compact" tri-diagonal coefficient matrix, of dimension N_x x N_y x 3 (N_x is the number of grid nodes in the x-direction, N_y is the number of grid nodes in the y-direction), and $\boldsymbol{b}$ is the RHS vector of dimension N_x x N_y. This routine gets arrays $\boldsymbol{A}$, and $\boldsymbol{b}$ as inputs. It returns the solution in vector $\boldsymbol{b}$, to conserve storage space. It uses the space allocated for the $\boldsymbol{A}$ to simultaneously store the elements of the lower ($\boldsymbol{L}$) and upper ($\boldsymbol{U}$) triangular matrices into which $\boldsymbol{A}$ is decomposed. It does this by not storing or using the diagonal elements of $\boldsymbol{U}$, which are all equal to 1.

A-3.2.4.2 Computing tridiagonal system coefficients and RHS vector: qlindgts_coeff_rhs

This routine computes all the coefficient and RHS vector elements in the arrays $\boldsymbol{A}$ and $\boldsymbol{b}$, respectively (discussed in the previous section). Essentially, it computes all of the expressions discussed in Sections A-2.2 & A-2.3 above. For most of the modifications discussed in Section A-3.1, it is here that all the linear or nonlinear PDE functionals have to be appropriately modified, and if necessary, to the boundary condition functionals that appear in calculating the elements of $\boldsymbol{A}$ and $\boldsymbol{b}$. The tridiagonal coefficient matrix, $\boldsymbol{A}$, is generated in the "compact" form described in the previous section.

A-3.2.4.3 Driver routine: delta_qlin_dgts

This is the driver routine for the numerical solution procedure adopted here – namely δ-form of "quazilinear" (Newton-Kantorovich) iterations coupled with the δ-form of the two level Douglas-Gunn (D-G) Scheme. The data flow within this subroutine is illustrated in the algorithm presented in Section A-2.4.1 above. Here, for each iteration of the quasilinearization process, the "imporved" iterate is constructed using two stages corresponding to the 2-step D-G scheme. Using an initial guess for temperature, $\boldsymbol{u}^{n-1}$, provided by the Main Program (Section 3.2.5) for EACH time step (Initial Condition, *f_initial*, for the 1st time step, and the converged value at the previous time step, for subsequent ones) to iterate to a converged value for that time step. It outputs the grid function values for the current time step, $\boldsymbol{u}^{n}$, to the main program. As discussed in detail in Sections A-2.2 & A-2.3, the grid functions at each Douglas-Gunn stage of a single nonlinear iteration, are related to those at the previous iteration by the compact time-split matrix formulae:

$$\boldsymbol{A}_x(\boldsymbol{u}^{(m)}, t_n) \,.\, \delta\boldsymbol{v}_1 = \boldsymbol{b}(\boldsymbol{u}^{(m)}, t_n, \boldsymbol{u}^{n-1}, t_{n-1}) \qquad and$$
$$\boldsymbol{A}_y(\boldsymbol{u}^{(m)}, t_n) \,.\, \delta\boldsymbol{v}_2 = \delta\boldsymbol{v}_1$$

where n denotes the time level index, m denotes the iteration counter, and $\boldsymbol{A}_x$ and $\boldsymbol{A}_y$ are the split coefficient arrays in the x- and y-direction, respectively, but having the same dimensions as $\boldsymbol{A}$ (N_x x N_y x 3). In each D-G stage, the routine first calls *qlindgts_coeff_rhs*, to obtain the coefficient and RHS vector arrays for that stage. The routine then calls the LU decomposition routine to compute δv_i at that stage.

After making both D-G passes, the routine updates the solution at the current iteration (in case of a nonlinear problem) or the current time step as follows:

$u^{(m)} = u^{(m-1)} + \delta v_2$ for the nonlinear problem, at iteration *m*, or
$u^n = u^{n-1} + \delta v_2$ for the linear problem, at time level *n*.

In the nonlinear case, it stores grid function values both at the last time step, $\boldsymbol{u}^{n-1}$, and for the last iteration, $\boldsymbol{u}^{(m-1)}$, as they are both required for every Newton-Kantorovich iteration.

A-3.2.5 MAIN PROGRAM nonlin_parabolic_pde

The main program contains the main time marching loop, and boundary condition flags, and performs almost all input/output (I/O) functions. The boundary condition flags were moved into the main program primarily to allow for changes in boundary condition (BC) types partway through a run. When this happens, the initial condition for the new set of BCs will be the same as the temperature, *u*, at the previous time step – but changes need to be made to this "initial" temperature, if any of the new BCs is Dirichlet. The chief functions of the Main Program are outlined in Figure A-1 above, and include:

- Read the four command line arguments (for the current version of *COND2D*).
- Compute the actual gird resolutions at which all calculations will be performed: use the minimum resolutions computed in the module *const_params* (Section 3.2 1 above), in conjunction with the spatial and temporal resolution flags from the command line (1st and 2nd, respectively). Then compute the grid node and time level indices for the problem domain defined in the module *const_params*.
- If the problem is nonlinear, define the convergence tolerance for nonlinear iterations, *quasi_epsilon*, and the maximum number of iterations allowed, *quasi_iterations*. If the problem is linear, set *quasi_epsilon* to a very large value and *quasi_iterations* to 1, so that the subroutine *delta_qlin_dgts* makes only the two required D-G passes at each time step.
- Compute any fault parameters that could not be computed in the module *fault_params* (due to Fortran 90 limitations – namely no expression containing a function call can appear in a parameter definition statement).
- Open all 5 output files outlined in Table A-2 above (and described below), and print out header information to all the output files and the screen.
- Compute all indicial information required for output data storage.
- Allocate all arrays needed in the run.
- <u>March through time:</u> AT THE FIRST TIME LEVEL, ASSIGN ALL BOUNDARY CONDITIONS. IF BOUNDARY CONDITION TYPE CHANGES AFTER A CERTAIN TIME, t_0, CHANGE IT THE FIRST TIME $t > t_0$. At each time level, *(a)* obtain the values of the solution at each time step, by calling the subroutine *delta_qlin_dgts* (described in the previous section); *(b)* Compute errors if exact solution is knows; *(c)* Store any relevant output data for later use; *(d)* smooth data if specified by the *smooth_flag*, using the given *smooth_factor* (3rd and 4th command line arguments, respectively); and *(e)* go to the next time level.
- At the end of the run, write all stored output data to the appropriate output files. Close output files and de-allocate all arrays before exiting.

In the following sections, three important features of the Main Program are described.

A-3.2.5.1 Command line arguments: Choosing optimal resolution

COND2D was designed to be run in batch mode, from a script file. Therefore, it accepts certain run specifications as command line arguments. Of course, the form and content of these arguments can be easily changed to the user's specifications. For the current version of *COND2D*, the program executable (created after compilation of source code and linking of object codes) must be followed by *THREE* 1-digit arguments, and *ONE* 6-digit argument, separated by spaces:

Argument # 1 – Spatial Resolution Flag (*res_flag_1*): A *ONE* character argument, it can have a value between 1 and 9. The actual spatial resolution of the run is determined as follows:

res_flag_1 = 1 implies x-step size, $hx = hx\_max/2^0$, y-step size, $hy = hy\_max/2^0$;
res_flag_1 = 2 implies x-step size, $hx = hx\_max/2^1$, y-step size, $hy = hy\_max/2^1$;
res_flag_1 = 3 implies x-step size, $hx = hx\_max/2^2$, y-step size, $hy = hy\_max/2^2$;
………………………………………………………………………………………
res_flag_1 = i implies x-step size, $hx = hx\_max/2^{i-1}$, y-step size, $hy = hy\_max/2^{i-1}$.

So, all else being equal, an increase in spatial resolution by 1 level results in a 4-fold increase in the number of grid-points, and a corresponding increase in the size of the coefficient, RHS, and solution arrays. Therefore, the arithmetic per D-G stage increases roughly 4 fold with each increase in spatial resolution level.

Argument # 2 – Temporal Resolution Flag (*res_flag_2*): A *ONE* character argument, it can have a value between 1 and 5 (due to machine size limitations, and huge time step increases with increasing resolution). The actual temporal resolution of the run is determined as follows:

res_flag_2 = 1 implies t-step size, $k = MIN(hx,hy)/10^0$;
res_flag_2 = 2 implies t-step size, $k = MIN(hx,hy)/10^1$;
res_flag_2 = 3 implies t-step size, $k = MIN(hx,hy)/10^2$;
………………………………………………………………………………………
res_flag_2 = j implies t-step size, $k = MIN(hx,hy)/10^{j-1}$;

So, all else being equal, an increase in temporal resolution by 1 level results in a 10-fold increase in the number of time steps at which the problem solution is computed, and so does the corresponding arithmetic for the entire run. Due to the coupling of the temporal resolution to the spatial resolution, each increase in temporal resolution level by 1 along with a spatial resolution level increase by 1, increases the arithmetic required for the run by a factor of 40! So, care has to be taken in determining the optimal resolution for the problem. One way to check this is to carry out convergence tests on the grid function values at successively smaller resolutions (keeping the ratio *hx:hy:k* constant) and then computing the rate of reduction in error. If this rate shows the expected 2nd order convergence of the solution, then no further increases in resolution are required. A useful strategy is to fix the temporal resolution at one level, then vary the spatial resolution as this strategy results in a smaller increase in arithmetic per change in level.

Another parameter to check for is the number of nonlinear iterations to convergence. Since the Newton-Kantorovich procedure converges quadratically (McDonough 2002), values for this number range between 3 and 5, typically. Of course, higher values may be reached for very non-smooth problems. This is also a good indicator of the stability of the run. If the maximum number of iterations is greater than about 10, and the nonlinear iterations do not converge within this limit, then it is possible that the problem my be under-resolved, and this requires an increase in the spatio-temporal resolution until quadratic convergence is observed. The number of iterations to convergence is output on the screen, if *verbose_flag = 1*, in the module *const_params* (Section 3.2.5.3).

A-3.2.5.2 Command line arguments: Smoothing and the under-resolution problem

The 3rd and the 4th command line arguments mentioned in the previous section pertain to smoothing, which may be required when a nonlinear problem possesses "extremely" steep gradients. Though it is optional to perform smoothing of the PDE problem data (including the solution) at each time step, the arguments are expected to be present. Therefore, there is an option to set the 3rd and 4th arguments to 0 (zero), each in their own format.

Argument # 3 – Smoothing Flag (*smooth_flag*): A *ONE* character argument, it can have values of 0, 1 or 2, with the following consequences:

smooth_flag = 0, implies NO smoothing
smooth_flag = 1, implies 1D smoothing
smooth_flag = 2, implies 2D smoothing

1D smoothing should be used when steep gradients exist *ONLY* along one of the principal directions of the problem domain. 2D smoothing should be used in the more general case, where the gradients are not aligned with only one of the principal directions. The actual smoothing procedure is outlined under the 4th argument below.

Argument # 4 – Smoothing Factor (*smooth_factor*): A *SIX*-character argument, it can have values ranging from 000000 to 999999 (~ 1 million). If *smooth_flag* (3rd command line argument) is 0 (zero), then this argument does not matter. For clarity, it should be set to 000000. If *smooth_flag* is non-zero, then either 1D or 2D smoothing needs to be performed on the solution at the end of every time step. Smoothing is essentially the application of a low-pass filter to the solution, to "smooth" out any steep gradients. The larger the value of this factor, the lesser the smoothing, and the lesser the solution deviates from its actual value at each time step. So, over the duration of the run, any such deviations can add up to give an erroneous result. Therefore, smoothing must be applied with caution. In the case of heat conduction in geologic settings, the thermal diffusivity is so low that the noise added by smoothing can erase the extremely slow conduction signature. So, for most geologic problems, smoothing might not be a good idea (actually, it is not recommended) – except as a desperate measure. Usually, the steep gradients do not cause problems if the resolution of the problem is sufficient. So, one way to get around smoothing in geologic problems is to try to use as small a domain size as practical, and then keep reducing the resolution until under-resolution problems (called the Reynolds cell problem in the computational fluid dynamics literature) vanish. This point is further illustrated in *Test Problem #32* discussed in Section A-3.4 below. In the rest of this Chapter, all tests and runs were carried out without employing any smoothing. The smoothing filters incorporated in *COND2D* are as follows:

Shuman filter for 1D smoothing: Applied to a user-defined range of rows *OR* columns. Here its application to a particular column is shown:

$$u(j, N_x-1) = [u(j, N_x-2) + (smooth\_factor).\ u(j, N_x-1) + u(j, Nx)]/(2 + smooth\_factor), \qquad j = 1{:}\ N_y$$

Shuman filter for 2D smoothing: Applied to a user-defined range of rows *AND* columns. Here its application to a specific problem subdomain is shown:

$$u(j, i) = [u(j, i-1) + u(j-1, i) + (smooth\_factor).\ u(j, i) + u(j, i+1) + u(j+1, i)]/(4 + smooth\_factor),$$
$$j = 1{:}\ 10,\ i = (N_x-3){:}N_x.$$

A-3.2.5.3 Output files and screen output

COND2D offers great flexibility in terms of the kind, and quantity of output that can be written to output files or screen. In addition to (optional) diagnostic screen output, the current version writes output to 5 different files. Illustrations of how data in these files can be used are presented under A-3.3.1. In what follows, these files are briefly described. **NOTE:** Grid functions or grid function values refer to the numerical solution (temperature) at specific grid nodes, corresponding to a specific grid resolution:

1. DGRID: This file contains grid function data in 2D, at the spatial resolution specified by the value of the variables *out_x_grid_spacing*, and *out_y_grid_spacing*, at time levels specified in the array *t_snap*, all of which are assigned in the module *const_params*. If either step size used for a run is at a lower resolution compared to the respective output resolution in that direction, the data are output at the step size resolution for that direction. Data in this file can be used to plot 2D surface plots using post-processing software such as MATLAB. A sample of this output file appears in Figure A-2.

2. DERRG: This file, formatted identically to the previous one, contains exact or estimated grid function errors depending on whether the exact solution is known or unknown.

3. DCONV: This file contains grid function values at the coordinates (in the problem domain) and time levels specified in the array *grid_conv*, which is assigned in the module *const_params*. The grid functions are output in a row, at the end of the file, for easy import into a spreadsheet software such as MS-EXCEL, for performing grid convergence tests at these spatio-temporal sampling points. A sample of this output file appears in Figure A-3.

4. DSNAP: This file contains grid function values along *TWO* profiles, each parallel to one of the principal axis. For the profile parallel to the x-axis, the y coordinate is set in the variable *y_xsnap*, and the time level is set in the variable *t_xsnap*, both assigned in the module *const_params*. Similarly, the corresponding variables for the profile parallel to the y-axis are: *x_ysnap*, and *t_ysnap*. A sample of this output file appears in Figure A-4.

5. DEVOL: This file contains four sets of data: (a) grid function values at the point *x_time* and *y_time*, as a function of time and for the duration of the run, at a temporal resolution specified by the variable *t_evol_spacing* (all these variables are assigned in the module *const_params*); (b) Evolution of the peak domain temperature, and its location, as a function of time - at “logarithmically” equidistant points (i.e., equidistant points on a logarithmic scale) - for the duration of the run; (c) Evolution of the maximum domain error (if available – exact or estimated), and its location, as a function of time - at “logarithmically” equidistant points (i.e., equidistant points on a logarithmic scale) - for the duration of the run; and (d) Evolution of the minimum domain temperature, and its location, as a function of time - at “logarithmically” equidistant points (i.e., equidistant points on a logarithmic scale) - for the duration of the run. A sample of this output file appears in Figure A-5.

6. SCREEN OUTPUT: Diagnostic messages can be output to the screen at each iteration and or time step of the entire run (this can quickly become a large amount of screen write data, and must be cautiously used – for instance, only for lower resolution runs for diagnostic purposes). The messages include the time level, maximum, and minimum domain temperature, and maximum domain error at the end of each time step, along with the corresponding errors or temperature, respectively; the number of nonlinear iterations till convergence at each time step, and the residual at the end of each iteration (this can be used to check if a problem is yielding the expected quadratic convergence). Usually the screen output can be redirected to another file for viewing later, especially when running the program in the background or in batch mode. So, if care is not taken, this file can exceed the storage capacity of a user’s account! A sample of the screen output appears in Figure A-6.

Figure A- 2. Sampling of output file DGRID

```
% Program to compute the solution evolution of a GENERALIZED NON-LINEAR, 2D
% HEAT CONDUCTION PDE, with GENERALIZED NON-LINEAR BCs, using the DELTA-FORM of
% QUASILINEARIZATION (NEWTON-KANTOROVICH PROCEDURE) WITH DOUGLAS-GUNN TIME SPLITTING SCHEME:
%  - by RAVI KANDA (July, 2002).
% Precision: KIND =  8 for FORTRAN90 Compiler v2.4 for HP-UX 11i on HP-SuperDome.
% ---------------------------------------------------------------------------
%
X-Limits: (x_left, x_right) = (0.00000000E+00,1.00000000E-01)
Y-Limits: (y_bottom, y_top) = (0.00000000E+00,3.14159265E+00)
t-Limits: (t_initial, t_final) = (0.00000000E+00,1.00000000E+00)
The value of x-step, hx = 1.00000000E-02
The value of y-step, hy = 1.00000000E-02
The value of t-step,  k = 1.00000000E-03
% ---------------------------------------------------------------------------
% This problem is indicated to be NON-LINEAR. Newton-Kantorovich
% iterations will be performed up to a convergence tolerance of 1.000000E-09.
% The maximum number of iterations, max_iter, was set to: 25.
% ---------------------------------------------------------------------
% SMOOTHING FLAG = 0: NO SMOOTHING WILL BE PERFORMED.
% ---------------------------------------------------
% COORDINATE SYSTEM: SPHERICAL.
% ------------------------------
Ambient Temperature,                                  U0 = 300 K.
Asperity Radius,                                      r0 =  0.100 m.
Young's Modulus,                                      E =  20.00 GPa.
Poisson's Ratio,                                     nu = 0.20 (dimensionless).
Coefficient of Friction,                              mu = 0.60 (dimensionless).
Density of asperity material,                        rho = 3000.00 kg/m**3.
Ambient average shear stress,                        TAU = 1.00E+08 Pa.
Asperity slip velocity,                                U =  1.000 m/sec.
The ratio,                                         rc/r0 = 1.88495559E-02 (dimensionless).
Maximum radius of circular asperity contact area,    rc = 1.885E-03 m.
Asperity slip duration,                               T0 = 7.540E-03 sec.
Maximum Asperity contact,                        THETA_0 = 0.01884732 Radians.
Specific Heat, Cp & Coeff. of Thermal Conductivity, k are NON-LINEAR FUNCTIONS OF TEMPERATURE.
% ---------------------------------------------------------------------------------------------
% For time <= To = 7.54E-03: LEFT BC = Linear Neumann; RIGHT BC = Non-Linear Neumann; BOTTOM BC = Linear Neumann; TOP BC = Linear Neumann;
% -------------------------------------------------------------------------------------------------------------------------------------
% For time > To = 7.54E-03: LEFT BC = Linear Neumann; RIGHT BC = Non-Linear Neumann; BOTTOM BC = Linear Neumann; TOP BC = Linear Neumann;
% -------------------------------------------------------------------------------------------------------------------------------------
TIME STEP     1: The solution u(x,y) at time =   0.000000
% ----------------------------------------------------------------
% x =   0.000000,  0.010000,  0.020000,  0.030000,  0.040000,  0.050000,  0.060000,  0.070000,  0.080000,  0.090000,  0.100000
y(    1)=  0.000000,     3.00000000E+02,     3.00000000E+02,     3.00000000E+02,     3.00000000E+02,     3.00000000E+02,     3.00000000E+02,     3.00000000E+02,     3.00000000E+02,     3.00000000E+02,     3.00000000E+02,     3.00000000E+02
y(    2)=  0.010000,     3.00000000E+02,     3.00000104E+02,     3.00000833E+02,     3.00002812E+02,     3.00006666E+02,     3.00013019E+02,     3.00022496E+02,     3.00035720E+02,     3.00053315E+02,     3.00075905E+02,     3.00104113E+02
y(    3)=  0.020000,     3.00000000E+02,     3.00000417E+02,     3.00003333E+02,     3.00011249E+02,     3.00026662E+02,     3.00052073E+02,     3.00089977E+02,     3.00142870E+02,     3.00213248E+02,     3.00303603E+02,     3.00416425E+02
y(    4)=  0.030000,     3.00000000E+02,     3.00000937E+02,     3.00007499E+02,     3.00025307E+02,     3.00059984E+02,     3.00117152E+02,     3.00202427E+02,     3.00321426E+02,     3.00479760E+02,     3.00683038E+02,     3.00936863E+02
y(    5)=  0.040000,     3.00000000E+02,     3.00001666E+02,     3.00013329E+02,     3.00044984E+02,     3.00106624E+02,     3.00208241E+02,     3.00359820E+02,     3.00571344E+02,     3.00852787E+02,     3.01214119E+02,     3.01665301E+02
y(    6)=  0.050000,     3.00000000E+02,     3.00002603E+02,     3.00020823E+02,     3.00070274E+02,     3.00166570E+02,     3.00325317E+02,     3.00562118E+02,     3.00892564E+02,     3.01332240E+02,     3.01896720E+02,     3.02601564E+02
y(    7)=  0.060000,     3.00000000E+02,     3.00003747E+02,     3.00029978E+02,     3.00101173E+02,     3.00239808E+02,     3.00468354E+02,     3.00809271E+02,     3.01285009E+02,     3.01918004E+02,     3.02730676E+02,     3.03745428E+02
y(    8)=  0.070000,     3.00000000E+02,     3.00005099E+02,     3.00040793E+02,     3.00137671E+02,     3.00326321E+02,     3.00637316E+02,     3.01101222E+02,     3.01748586E+02,     3.02609938E+02,     3.03715788E+02,     3.05096618E+02
y(    9)=  0.080000,     3.00000000E+02,     3.00006658E+02,     3.00053264E+02,     3.00179762E+02,     3.00426087E+02,     3.00832163E+02,     3.01437899E+02,     3.02283182E+02,     3.03407877E+02,     3.04851818E+02,     3.06654811E+02
y(   10)=  0.090000,     3.00000000E+02,     3.00008424E+02,     3.00067389E+02,     3.00227434E+02,     3.00539083E+02,     3.01052849E+02,     3.01819223E+02,     3.02888670E+02,     3.04311628E+02,     3.06138495E+02,     3.08419632E+02
y(   11)=  0.100000,     3.00000000E+02,     3.00010396E+02,     3.00083165E+02,     3.00280675E+02,     3.00665282E+02,     3.01299319E+02,     3.02245101E+02,     3.03564905E+02,     3.05320975E+02,     3.07575510E+02,     3.10390659E+02
y(   12)=  0.110000,     3.00000000E+02,     3.00012574E+02,     3.00100588E+02,     3.00339475E+02,     3.00804653E+02,     3.01571516E+02,     3.02715431E+02,     3.04311724E+02,     3.06435676E+02,     3.09162518E+02,     3.12567418E+02
y(   13)=  0.120000,     3.00000000E+02,     3.00014957E+02,     3.00119653E+02,     3.00403817E+02,     3.00957163E+02,     3.01869375E+02,     3.03230101E+02,     3.05128948E+02,     3.07655465E+02,     3.10899140E+02,     3.14949389E+02
y(   14)=  0.130000,     3.00000000E+02,     3.00017545E+02,     3.00140355E+02,     3.00473687E+02,     3.01122775E+02,     3.02192822E+02,     3.03788988E+02,     3.06016382E+02,     3.08980048E+02,     3.12784959E+02,     3.17536000E+02
y(   15)=  0.140000,     3.00000000E+02,     3.00020337E+02,     3.00162691E+02,     3.00549069E+02,     3.01301451E+02,     3.02541782E+02,     3.04391958E+02,     3.06973813E+02,     3.10409109E+02,     3.14819523E+02,     3.20326633E+02
y(   16)=  0.150000,     3.00000000E+02,     3.00023332E+02,     3.00186655E+02,     3.00629943E+02,     3.01493147E+02,     3.02916171E+02,     3.05038866E+02,     3.08001012E+02,     3.11942306E+02,     3.17002346E+02,     3.23320618E+02
y(   17)=  0.160000,     3.00000000E+02,     3.00026530E+02,     3.00212240E+02,     3.00716291E+02,     3.01697816E+02,     3.03315898E+02,     3.05729557E+02,     3.09097733E+02,     3.13579271E+02,     3.19332904E+02,     3.26517240E+02
y(   18)=  0.170000,     3.00000000E+02,     3.00029931E+02,     3.00239441E+02,     3.00808092E+02,     3.01915411E+02,     3.03740869E+02,     3.06463866E+02,     3.10263713E+02,     3.15319612E+02,     3.21810641E+02,     3.29915733E+02
y(   19)=  0.180000,     3.00000000E+02,     3.00033532E+02,     3.00268251E+02,     3.00905325E+02,     3.02145879E+02,     3.04190982E+02,     3.07241618E+02,     3.11498674E+02,     3.17162915E+02,     3.24434964E+02,     3.33515284E+02
y(   20)=  0.190000,     3.00000000E+02,     3.00037334E+02,     3.00298664E+02,     3.01007964E+02,     3.02389165E+02,     3.04666129E+02,     3.08062627E+02,     3.12802321E+02,     3.19108737E+02,     3.27205244E+02,     3.37315034E+02
y(   21)=  0.200000,     3.00000000E+02,     3.00041335E+02,     3.00330671E+02,     3.01115988E+02,     3.02645212E+02,     3.05166196E+02,     3.08926696E+02,     3.14174342E+02,     3.21156614E+02,     3.30120822E+02,     3.41314075E+02
y(   22)=  0.210000,     3.00000000E+02,     3.00045534E+02,     3.00364266E+02,     3.01229368E+02,     3.02913956E+02,     3.05691065E+02,     3.09833620E+02,     3.15614409E+02,     3.23306057E+02,     3.33180999E+02,     3.45511450E+02
y(   23)=  0.220000,     3.00000000E+02,     3.00049931E+02,     3.00399441E+02,     3.01348080E+02,     3.03195336E+02,     3.06240610E+02,     3.10783181E+02,     3.17122178E+02,     3.25556553E+02,     3.36385047E+02,     3.49906159E+02
y(   24)=  0.230000,     3.00000000E+02,     3.00054524E+02,     3.00436187E+02,     3.01472093E+02,     3.03489283E+02,     3.06814699E+02,     3.11775153E+02,     3.18697291E+02,     3.27907566E+02,     3.39732199E+02,     3.54497153E+02
y(   25)=  0.240000,     3.00000000E+02,     3.00059313E+02,     3.00474494E+02,     3.01601379E+02,     3.03795728E+02,     3.07413197E+02,     3.12809299E+02,     3.20339371E+02,     3.30358533E+02,     3.43221659E+02,     3.59283338E+02
y(   26)=  0.250000,     3.00000000E+02,     3.00064295E+02,     3.00514355E+02,     3.01735906E+02,     3.04114596E+02,     3.08035959E+02,     3.13885374E+02,     3.22048027E+02,     3.32908872E+02,     3.46852595E+02,     3.64263572E+02

.........................................................................................................

TIME STEP  1001: The solution u(x,y) at time =   1.000000
% ----------------------------------------------------------------
% x =   0.000000,  0.010000,  0.020000,  0.030000,  0.040000,  0.050000,  0.060000,  0.070000,  0.080000,  0.090000,  0.100000
y(    1)=  0.000000,     5.87123378E+02,     2.99648220E+02,     2.99999178E+02,     3.00000059E+02,     3.00000050E+02,     3.00000051E+02,     3.00000056E+02,     3.00000063E+02,     3.00000070E+02,     3.00000078E+02,     3.00000076E+02
y(    2)=  0.010000,     5.87123385E+02,     2.99648262E+02,     2.99999486E+02,     3.00001095E+02,     3.00002503E+02,     3.00004843E+02,     3.00008336E+02,     3.00013210E+02,     3.00019694E+02,     3.00028016E+02,     3.00038393E+02
y(    3)=  0.020000,     5.87123407E+02,     2.99648387E+02,     3.00000409E+02,     3.00004202E+02,     3.00009865E+02,     3.00019218E+02,     3.00033174E+02,     3.00052649E+02,     3.00078560E+02,     3.00111823E+02,     3.00153338E+02
y(    4)=  0.030000,     5.87123442E+02,     2.99648595E+02,     3.00001946E+02,     3.00009379E+02,     3.00022132E+02,     3.00043173E+02,     3.00074565E+02,     3.00118370E+02,     3.00176654E+02,     3.00251480E+02,     3.00344881E+02
y(    5)=  0.040000,     5.87123492E+02,     2.99648886E+02,     3.00004099E+02,     3.00016626E+02,     3.00039302E+02,     3.00076702E+02,     3.00132497E+02,     3.00210358E+02,     3.00313953E+02,     3.00446953E+02,     3.00612979E+02
y(    6)=  0.050000,     5.87123556E+02,     2.99649260E+02,     3.00006866E+02,     3.00025941E+02,     3.00061370E+02,     3.00119798E+02,     3.00206959E+02,     3.00328589E+02,     3.00490425E+02,     3.00698195E+02,     3.00957565E+02
y(    7)=  0.060000,     5.87123634E+02,     2.99649717E+02,     3.00010246E+02,     3.00037320E+02,     3.00088331E+02,     3.00172448E+02,     3.00297930E+02,     3.00473037E+02,     3.00706025E+02,     3.01005146E+02,     3.01378557E+02
y(    8)=  0.070000,     5.87123727E+02,     2.99650257E+02,     3.00014239E+02,     3.00050762E+02,     3.00120180E+02,     3.00234642E+02,     3.00405391E+02,     3.00643666E+02,     3.00960704E+02,     3.01367731E+02,     3.01875855E+02
y(    9)=  0.080000,     5.87123834E+02,     2.99650879E+02,     3.00018844E+02,     3.00066264E+02,     3.00156907E+02,     3.00306364E+02,     3.00529314E+02,     3.00840436E+02,     3.01254399E+02,     3.01785865E+02,     3.02449340E+02
y(   10)=  0.090000,     5.87123955E+02,     2.99651584E+02,     3.00024059E+02,     3.00083821E+02,     3.00198505E+02,     3.00387597E+02,     3.00669671E+02,     3.01063298E+02,     3.01587041E+02,     3.02259447E+02,     3.03098874E+02
y(   11)=  0.100000,     5.87124090E+02,     2.99652371E+02,     3.00029884E+02,     3.00103430E+02,     3.00244963E+02,     3.00478321E+02,     3.00826427E+02,     3.01312201E+02,     3.01958550E+02,     3.02788364E+02,     3.03824301E+02
y(   12)=  0.110000,     5.87124239E+02,     2.99653240E+02,     3.00036316E+02,     3.00125085E+02,     3.00296271E+02,     3.00578515E+02,     3.00999545E+02,     3.01587083E+02,     3.02368836E+02,     3.03372488E+02,     3.04625448E+02
y(   13)=  0.120000,     5.87124402E+02,     2.99654192E+02,     3.00043356E+02,     3.00148782E+02,     3.00352415E+02,     3.00688154E+02,     3.01188984E+02,     3.01887880E+02,     3.02817801E+02,     3.04011680E+02,     3.05502124E+02
y(   14)=  0.130000,     5.87124579E+02,     2.99655225E+02,     3.00050999E+02,     3.00174514E+02,     3.00413383E+02,     3.00807213E+02,     3.01394697E+02,     3.02214518E+02,     3.03305338E+02,     3.04705787E+02,     3.06454118E+02
y(   15)=  0.140000,     5.87124770E+02,     2.99656339E+02,     3.00059246E+02,     3.00202277E+02,     3.00479160E+02,     3.00935662E+02,     3.01616637E+02,     3.02566921E+02,     3.03831330E+02,     3.05454642E+02,     3.07481203E+02
y(   16)=  0.150000,     5.87124975E+02,     2.99657535E+02,     3.00068094E+02,     3.00232062E+02,     3.00549730E+02,     3.01073472E+02,     3.01854749E+02,     3.02945003E+02,     3.04395651E+02,     3.06258067E+02,     3.08583132E+02
y(   17)=  0.160000,     5.87125194E+02,     2.99658811E+02,     3.00077540E+02,     3.00263863E+02,     3.00625076E+02,     3.01220609E+02,     3.02108977E+02,     3.03348675E+02,     3.04998166E+02,     3.07115869E+02,     3.09759644E+02
y(   18)=  0.170000,     5.87125427E+02,     2.99660169E+02,     3.00087583E+02,     3.00297672E+02,     3.00705180E+02,     3.01377037E+02,     3.02379259E+02,     3.03777838E+02,     3.05638731E+02,     3.08027843E+02,     3.11010456E+02
y(   19)=  0.180000,     5.87125673E+02,     2.99661606E+02,     3.00098220E+02,     3.00333482E+02,     3.00790023E+02,     3.01542720E+02,     3.02665532E+02,     3.04232392E+02,     3.06317193E+02,     3.08993771E+02,     3.12335269E+02
y(   20)=  0.190000,     5.87125933E+02,     2.99663124E+02,     3.00109449E+02,     3.00371283E+02,     3.00879585E+02,     3.01717618E+02,     3.02967727E+02,     3.04712227E+02,     3.07033389E+02,     3.10013422E+02,     3.13733769E+02
y(   21)=  0.200000,     5.87126207E+02,     2.99664721E+02,     3.00121266E+02,     3.00411067E+02,     3.00973845E+02,     3.01901689E+02,     3.03285771E+02,     3.05217228E+02,     3.07787149E+02,     3.11086553E+02,     3.15205619E+02
.........................................................................................................
```

```
% Program to compute the solution evolution of a GENERALIZED NON-LINEAR, 2D
% HEAT CONDUCTION PDE, with GENERALIZED NON-LINEAR BCs, using the DELTA-FORM of
% QUASILINEARIZATION (NEWTON-KANTOROVICH PROCEDURE) WITH DOUGLAS-GUNN TIME SPLITTING SCHEME:
%  - by RAVI KANDA (July, 2002).
% Precision: KIND =  8 for FORTRAN90 Compiler v2.4 for HP-UX 11i on HP-SuperDome.
% -----------------------------------------------------------------------------
%
X-Limits: (x_left, x_right) = (0.00000000E+00,1.00000000E-01)
Y-Limits: (y_bottom, y_top) = (0.00000000E+00,3.14159265E+00)
t-Limits: (t_initial, t_final) = (0.00000000E+00,1.00000000E+00)
The value of x-step, hx = 1.00000000E-02
The value of y-step, hy = 1.00000000E-02
The value of t-step,  k = 1.00000000E-03
% -----------------------------------------------------------------------------
% This problem is indicated to be NON-LINEAR. Newton-Kantorovich
% iterations will be performed up to a convergence tolerance of 1.000000E-09.
% The maximum number of iterations, max_iter, was set to: 25.
% -----------------------------------------------------------------------
% SMOOTHING FLAG = 0: NO SMOOTHING WILL BE PERFORMED.
% -----------------------------------------------
% COORDINATE SYSTEM: SPHERICAL.
% -------------------------------
Ambient Temperature,                                 U0 = 300 K.
Asperity Radius,                                     r0 =  0.100 m.
Young's Modulus,                                     E =  20.00 GPa.
Poisson's Ratio,                                    nu = 0.20 (dimensionless).
Coefficient of Friction,                             mu = 0.60 (dimensionless).
Density of asperity material,                       rho = 3000.00 kg/m**3.
Ambient average shear stress,                       TAU = 1.00E+08 Pa.
Asperity slip velocity,                               U =  1.000 m/sec.
The ratio,                                        rc/r0 = 1.88495559E-02 (dimensionless).
Maximum radius of circular asperity contact area,    rc = 1.885E-03 m.
Asperity slip duration,                              T0 = 7.540E-03 sec.
Maximum Asperity contact,                       THETA_0 = 0.01884732 Radians.
Specific Heat, Cp & Coeff. of Thermal Conductivity, k are NON-LINEAR FUNCTIONS OF TEMPERATURE.
% -----------------------------------------------------------------------------------------------
 x = 0.06 y = 0.31 t = 0.15
 x = 0.06 y = 1.57 t = 0.15
 x = 0.07 y = 2.83 t = 0.15
 x = 0.07 y = 0.47 t = 0.15
 x = 0.08 y = 2.98 t = 0.20
 x = 0.08 y = 1.57 t = 0.20
 x = 0.09 y = 1.26 t = 0.20
 x = 0.07 y = 1.41 t = 0.20

-------------------------------------------------------------------------------
% For time <= To = 7.54E-03: LEFT BC = Linear Neumann; RIGHT BC = Non-Linear Neumann; BOTTOM BC = Linear Neumann; TOP BC = Linear Neumann;
% -------------------------------------------------------------------------------------------------------------------------------------------
% For time > To = 7.54E-03: LEFT BC = Linear Neumann; RIGHT BC = Non-Linear Neumann; BOTTOM BC = Linear Neumann; TOP BC = Linear Neumann;
% -------------------------------------------------------------------------------------------------------------------------------------------

Grid Function Convergence Data at the following grid points:
-------------------------------------------------------------------------------
 k hx hy U1 U2 U3 U4 U5 U6 U7 U8
 0.001000 0.010000 0.010000 3.1825474843E+02  5.9452271006E+02  6.8465079677E+02  3.6498895873E+02  1.1636252311E+03  7.4486672902E+02  1.0209691616E+03  6.9312879805E+02
```

Figure A- 3. Sampling of output file DCONV

Figure A- 4. Sampling of output file DSNAP

```
% Program to compute the solution evolution of a GENERALIZED NON-LINEAR, 2D
% HEAT CONDUCTION PDE, with GENERALIZED NON-LINEAR BCs, using the DELTA-FORM of
% QUASILINEARIZATION (NEWTON-KANTOROVICH PROCEDURE) WITH DOUGLAS-GUNN TIME SPLITTING SCHEME:
%  - by RAVI KANDA (July, 2002).
% Precision: KIND =  8 for FORTRAN90 Compiler v2.4 for HP-UX 11i on HP-SuperDome.
% ------------------------------------------------------------------------------
%
X-Limits: (x_left, x_right) = (0.00000000E+00,1.00000000E-01)
Y-Limits: (y_bottom, y_top) = (0.00000000E+00,3.14159265E+00)
t-Limits: (t_initial, t_final) = (0.00000000E+00,1.00000000E+00)
The value of x-step, hx = 1.00000000E-02
The value of y-step, hy = 1.00000000E-02
The value of t-step,  k = 1.00000000E-03
% ------------------------------------------------------------------------------
% This problem is indicated to be NON-LINEAR. Newton-Kantorovich
% iterations will be performed up to a convergence tolerance of 1.000000E-09.
% The maximum number of iterations, max_iter, was set to: 25.
% ------------------------------------------------------------------------
% SMOOTHING FLAG = 0: NO SMOOTHING WILL BE PERFORMED.
% ---------------------------------------------------
% COORDINATE SYSTEM: SPHERICAL.
% -------------------------------
Ambient Temperature,                                 U0 = 300 K.
Asperity Radius,                                     r0 =  0.100 m.
Young's Modulus,                                     E =  20.00 GPa.
Poisson's Ratio,                                    nu = 0.20 (dimensionless).
Coefficient of Friction,                             mu = 0.60 (dimensionless).
Density of asperity material,                       rho = 3000.00 kg/m**3.
Ambient average shear stress,                       TAU = 1.00E+08 Pa.
Asperity slip velocity,                               U =  1.000 m/sec.
The ratio,                                        rc/r0 = 1.88495559E-02 (dimensionless).
Maximum radius of circular asperity contact area,    rc = 1.885E-03 m.
Asperity slip duration,                              T0 = 7.540E-03 sec.
Maximum Asperity contact,                       THETA_0 = 0.01884732 Radians.
Specific Heat, Cp & Coeff. of Thermal Conductivity, k are NON-LINEAR FUNCTIONS OF TEMPERATURE.
% ------------------------------------------------------------------------------------------------
% For time <= To = 7.54E-03: LEFT BC = Linear Neumann; RIGHT BC = Non-Linear Neumann; BOTTOM BC = Linear Neumann; TOP BC = Linear Neumann;
% ------------------------------------------------------------------------------------------------------------------------------------
% For time > To = 7.54E-03: LEFT BC = Linear Neumann; RIGHT BC = Non-Linear Neumann; BOTTOM BC = Linear Neumann; TOP BC = Linear Neumann;
% ------------------------------------------------------------------------------------------------------------------------------------
SNAPSHOT at y =  0.200000 & t =  0.150000:
----------------------------------------------------------------------
     x        U_xsnap(x)
   0.00    3.4453633924E+02
   0.01    3.0003438239E+02
   0.02    3.0028477287E+02
   0.03    3.0096070256E+02
   0.04    3.0227699229E+02
   0.05    3.0444697231E+02
   0.06    3.0768389827E+02
   0.07    3.1220092251E+02
   0.08    3.1821107109E+02
   0.09    3.2592722156E+02
   0.10    3.3556182637E+02
----------------------------------------------------------------------
SNAPSHOT at x =  0.090000 & t =  0.200000:
----------------------------------------------------------------------
     y        U_ysnap(y)
   0.00    3.0000000382E+02
   0.01    3.0006215552E+02
   0.02    3.0024859726E+02
   0.03    3.0055928388E+02
   0.04    3.0099414080E+02
   0.05    3.0155306367E+02
   0.06    3.0223591840E+02
   0.07    3.0304254120E+02
   0.08    3.0397273861E+02
   0.09    3.0502628759E+02
   0.10    3.0620293555E+02
   0.11    3.0750240047E+02
   0.12    3.0892437096E+02
   0.13    3.1046850639E+02
   0.14    3.1213443696E+02
   0.15    3.1392176384E+02
   0.16    3.1583005930E+02
   0.17    3.1785886681E+02
   0.18    3.2000770125E+02
   0.19    3.2227604900E+02
   0.20    3.2466336812E+02
::::::::::::::::::………..
:::::::::::::::::……………………
   3.00    1.5289862861E+03
   3.01    1.5287236793E+03
   3.02    1.5284748321E+03
   3.03    1.5282406429E+03
   3.04    1.5280220017E+03
   3.05    1.5278197899E+03
   3.06    1.5276348799E+03
   3.07    1.5274681351E+03
   3.08    1.5273204101E+03
   3.09    1.5271925500E+03
   3.10    1.5270853907E+03
   3.11    1.5269997576E+03
   3.12    1.5269364242E+03
   3.13    1.5268938218E+03
   3.14    1.5268938218E+03
----------------------------------------------------------------------
```

Figure A- 5. Sampling of output file DEVOL

```
% Program to compute the solution evolution of a GENERALIZED NON-LINEAR, 2D
% HEAT CONDUCTION PDE, with GENERALIZED NON-LINEAR BCs, using the DELTA-FORM of
% QUASILINEARIZATION (NEWTON-KANTOROVICH PROCEDURE) WITH DOUGLAS-GUNN TIME SPLITTING SCHEME:
%  - by RAVI KANDA (July, 2002).
% Precision: KIND =  8 for FORTRAN90 Compiler v2.4 for HP-UX 11i on HP-SuperDome.
% ---------------------------------------------------------------------------
%
X-Limits: (x_left, x_right) = (0.00000000E+00,1.00000000E-01)
Y-Limits: (y_bottom, y_top) = (0.00000000E+00,3.14159265E+00)
t-Limits: (t_initial, t_final) = (0.00000000E+00,1.00000000E+00)
The value of x-step, hx = 1.00000000E-02
The value of y-step, hy = 1.00000000E-02
The value of t-step,  k = 1.00000000E-03
% ---------------------------------------------------------------------------
% This problem is indicated to be NON-LINEAR. Newton-Kantorovich
% iterations will be performed up to a convergence tolerance of 1.000000E-09.
% The maximum number of iterations, max_iter, was set to: 25.
% ---------------------------------------------------------------------
% SMOOTHING FLAG = 0: NO SMOOTHING WILL BE PERFORMED.
% --------------------------------------------------
% COORDINATE SYSTEM: SPHERICAL.
% -------------------------------
Ambient Temperature,                                U0 = 300 K.
Asperity Radius,                                    r0 =  0.100 m.
Young's Modulus,                                    E =  20.00 GPa.
Poisson's Ratio,                                   nu = 0.20 (dimensionless).
Coefficient of Friction,                            mu = 0.60 (dimensionless).
Density of asperity material,                      rho = 3000.00 kg/m**3.
Ambient average shear stress,                      TAU = 1.00E+08 Pa.
Asperity slip velocity,                              U =  1.000 m/sec.
The ratio,                                       rc/r0 = 1.88495559E-02 (dimensionless).
Maximum radius of circular asperity contact area,    rc = 1.885E-03 m.
Asperity slip duration,                             T0 = 7.540E-03 sec.
Maximum Asperity contact,                      THETA_0 = 0.01884732 Radians.
Specific Heat, Cp & Coeff. of Thermal Conductivity, k are NON-LINEAR FUNCTIONS OF TEMPERATURE.
% ------------------------------------------------------------------------------------------
% For time <= To = 7.54E-03: LEFT BC = Linear Neumann; RIGHT BC = Non-Linear Neumann; BOTTOM BC = Linear Neumann; TOP BC = Linear Neumann;
% -------------------------------------------------------------------------------------------------------------------------------------
% For time > To = 7.54E-03: LEFT BC = Linear Neumann; RIGHT BC = Non-Linear Neumann; BOTTOM BC = Linear Neumann; TOP BC = Linear Neumann;
% -------------------------------------------------------------------------------------------------------------------------------------
TIME LAG BETWEEN TIME CORRESPONDING TO U_max AND TIME AT ASPERITY SEPARATION = -7.539822E-03
RELATIVE TIME LAG (w.r.t. T0) BETWEEN TIME CORRESPONDING TO U_max AND TIME AT ASPERITY SEPARATION = -1.000000E+00
--------------------------------------------------------------------------------------------
Grid Function evolution at grid point: ( 0.000000,  3.141593).
----------------------------------------------------------------------------------
     t     U(x_time, y_time)
   0.00    3.00000000E+02
   0.05    3.14959791E+02
   0.10    3.29815126E+02
   0.15    3.44575958E+02
   0.20    3.59251274E+02
   0.25    3.73849201E+02
   0.30    3.88377106E+02
:::::::::::::::
   0.65    4.88603243E+02
   0.70    5.02767062E+02
   0.75    5.16903338E+02
   0.80    5.31014956E+02
   0.85    5.45104546E+02
   0.90    5.59174506E+02
   0.95    5.73227023E+02
   1.00    5.87264087E+02
-----------------------------------------------------------------------
Domain Maximum Temperature evolution:
---------------------------------------------------------------------------------------------
    Step #         t            j        i         U_max        Relative Error        U_norm
         1   0.000000E+00      261      11    2.37648033E+03    0.00000000E+00    4.99013634E+04
         2   1.000000E-03      261      11    2.37440606E+03    2.34352267E-08    4.98658172E+04
         3   2.000000E-03      261      11    2.37233386E+03    4.68591915E-08    4.98303108E+04
         4   3.000000E-03      261      11    2.37026373E+03    7.02718750E-08    4.97948442E+04
         5   4.000000E-03      261      11    2.36819566E+03    9.36732581E-08    4.97594174E+04
         6   5.000000E-03      261      11    2.36612967E+03    1.17063321E-07    4.97240303E+04
         7   6.000000E-03      261      11    2.36406573E+03    1.40442046E-07    4.96886828E+04
         8   7.000000E-03      261      11    2.36200386E+03    1.63809411E-07    4.96533751E+04
         9   8.000000E-03      261      11    2.35994405E+03    1.87165400E-07    4.96181070E+04
        10   9.000000E-03      261      11    2.35788629E+03    2.10509991E-07    4.95828784E+04
:::::::::::::::::::::::::::
       501   5.000000E-01      261      11    1.55981016E+03    9.91678708E-06    3.64205649E+04
       601   6.000000E-01      261      11    1.43998956E+03    1.13627838E-05    3.45856177E+04
       701   7.000000E-01      261      11    1.33156513E+03    1.25987185E-05    3.29806529E+04
       801   8.000000E-01      261      11    1.23345290E+03    1.36186910E-05    3.15850174E+04
       901   9.000000E-01      261      11    1.14467202E+03    1.44215454E-05    3.03795897E+04
      1001   1.000000E+00      261      11    1.06433495E+03    1.50110361E-05    2.93466368E+04
TEMPORAL GLOBAL TEMPERATURE MAXIMA:
------------------------------------
         1   0.00      261       11    2.37648033E+03    0.00000000E+00
--------------------------------------------------------------------------------------------
Domain Maximum Error evolution:
---------------------------------------------------------------------------------------------
    Step #         t            j        i      Max. Rel. Error          U              U_norm
         1   0.000000E+00        0        0    0.00000000E+00    0.00000000E+00    4.99013634E+04
         2   1.000000E-03      261        1    6.02200492E-06    3.00300292E+02    4.98658172E+04
         3   2.000000E-03      261        1    1.20516672E-05    3.00600538E+02    4.98303108E+04
:::::::::::::::::::::::::::::::::::
       901   9.000000E-01      261        1    8.53125075E-03    5.59175897E+02    3.03795897E+04
      1001   1.000000E+00      261        1    9.78870380E-03    5.87265536E+02    2.93466368E+04

TEMPORAL GLOBAL ABSOLUTE ERROR MAXIMA:
------------------------------
      1001   1.00      261        1    2.87265536E+02    5.87265536E+02
--------------------------------------------------------------------------------------------
Domain Minimum Temperature evolution:
---------------------------------------------------------------------------------------------
    Step #         t            j        i         U_min        Relative Error        U_norm
         1   0.000000E+00        1        1    3.00000000E+02    0.00000000E+00    4.99013634E+04
         2   1.000000E-03        1       11    3.00000000E+02    4.18345432E-13    4.98658172E+04
         3   2.000000E-03        1       11    3.00000000E+02    8.29096533E-13    4.98303108E+04
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

Figure A- 6. Sampling of SCREEN OUTPUT

```
Program to compute the solution of a GENERAL NON-LINEAR, 2D, HEAT CONDUCTION EQUATION (in Cartesian/
 Cylindrical/Spherical coordinates), with general NON-LINEAR BOUNDARY CONDITIONS USING THE DELTA-FORM
 OF QUASILINIARIZATION (NEWTON-KANTOROVICH PROCEDURE) IN CONJUNCTION WITH THE DELTA-FORM OF THE
 DOUGLAS-GUNN TIME SPLITTING SCHEME (2-STEP). THIS CODE CAN ALSO BE USED FOR LINEAR PROBLEMS WITHOUT
 ANY CHANGES TO THE CORE SUBROUTINES OF THIS IMPLEMENTATION. - by RAVI KANDA (November, 2002).
 ----------------------------------------------------------------------------------------------------

 WARNING: Grid output has been requested at a higher resolution than hx! Setting this to equal hx.
 X-Limits: (x_left, x_right)    = ( 0.0 , 0.1 )
 Y-Limits: (y_bottom, y_top)    = ( 0.0 , 3.14159265358979 )
 t-Limits: (t_initial, t_final) = ( 0.0 , 1.0 )
 The value of x-step, hx        =  1.000000000000000E-02
 The value of y-step, hy        =  1.000000000000000E-02
 The value of t-step,  k        =  1.000000000000000E-03
 Smoothing Flag                 =  0
 Smoothing Factor               =  0.0
 -------------------------------------------------------------------------------------
 Ambient Temperature,                             U0 =  300 K.
 Asperity Radius                                  r0 =  0.1  m.
 Young''s Modulus,                                 E =  20.0  GPa.
 Poisson's Ratio,                                 nu =  0.2  (dimensionless).
 Coefficient of Friction,                         mu =  0.6  (dimensionless).
 Density of asperity material,                   rho =  3000.0  kg/m**3.
 Ambient average shear stress,                   TAU =  100000000.0  Pa.
 Asperity slip velocity,                           U =  1.0  m/sec.
 The ratio,                                    rc/r0 =  1.884955592153876E-02  (dimensionless).
 Maximum radius of circular asperity contact area, rc =  1.884955592153876E-03  m.
 Asperity slip duration,                          T0 =  7.539822368615504E-03  sec.
 Maximum Asperity contact                    THETA_0 =  1.884723394541884E-02  Radians.
 Specific Heat, Cp & Coeff. of Thermal Conductivity, k are NON-LINEAR FUNCTIONS OF TEMPERATURE.
 -------------------------------------------------------------------------------------
 ALL grid ARRAYS SUCCESSFULLY ALLOCATED.
 ALL xsnap ARRAYS SUCCESSFULLY ALLOCATED.
 ALL ysnap ARRAYS SUCCESSFULLY ALLOCATED.
 ALL t_evol ARRAYS SUCCESSFULLY ALLOCATED.
 ALL temperature evolution ARRAYS SUCCESSFULLY ALLOCATED.
 ALL non-output-file ARRAYS SUCCESSFULLY ALLOCATED.
 -----------------------------------------------------------------------------------------------------------------
 t( 1 ) =  0.0 :
              row= 261 , col= 11 : DOMAIN MAXIMUM TEMPERATURE =  2376.480334151895
              row= 1 , col= 1 : DOMAIN MINIMUM TEMPERATURE =  300.0
              row= 0 , col= 0  : DOMAIN MAXIMUM ERROR      =  0.0 , TEMPERATURE =  0.0 .
TIME = 1.000E-03. Newton-Kantorovich Iterations Converged after  5 iterations. Final value of L2-norm of Dn: 5.274812E-12.
 t( 2 ) =  1.000000000000000E-03 :
              row= 261 , col= 11 : DOMAIN MAXIMUM TEMPERATURE =  2374.406060328647
              row= 1 , col= 11 : DOMAIN MINIMUM TEMPERATURE =  299.9999999791389
              row= 261 , col= 1  : DOMAIN MAXIMUM ERROR      =  6.022004915908457E-06 , TEMPERATURE =  300.3002921960209 .
TIME = 2.000E-03. Newton-Kantorovich Iterations Converged after  5 iterations. Final value of L2-norm of Dn: 5.169939E-12.
 t( 3 ) =  2.000000000000000E-03 :
              row= 261 , col= 11 : DOMAIN MAXIMUM TEMPERATURE =  2372.333858685076
              row= 1 , col= 11 : DOMAIN MINIMUM TEMPERATURE =  299.9999999586859
              row= 261 , col= 1  : DOMAIN MAXIMUM ERROR      =  1.205166717114302E-05 , TEMPERATURE =  300.600538320344 .
TIME = 3.000E-03. Newton-Kantorovich Iterations Converged after  5 iterations. Final value of L2-norm of Dn: 5.043072E-12.
 t( 4 ) =  3.000000000000000E-03 :
              row= 261 , col= 11 : DOMAIN MAXIMUM TEMPERATURE =  2370.263727150852
              row= 1 , col= 11 : DOMAIN MINIMUM TEMPERATURE =  299.999999938641
              row= 261 , col= 1  : DOMAIN MAXIMUM ERROR      =  1.808899054919657E-05 , TEMPERATURE =  300.9007384656672 .
TIME = 4.000E-03. Newton-Kantorovich Iterations Converged after  5 iterations. Final value of L2-norm of Dn: 5.536911E-12.
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

TIME = 9.970E-01. Newton-Kantorovich Iterations Converged after  5 iterations. Final value of L2-norm of Dn: 2.123085E-12.
 t( 998 ) =  0.997000000000001 :
              row= 261 , col= 11 : DOMAIN MAXIMUM TEMPERATURE =  1066.630072141645
              row= 1 , col= 2 : DOMAIN MINIMUM TEMPERATURE =  299.6511006798737
              row= 261 , col= 1  : DOMAIN MAXIMUM ERROR      =  286.4237120681518 , TEMPERATURE =  586.4237120681518 .
TIME = 9.980E-01. Newton-Kantorovich Iterations Converged after  5 iterations. Final value of L2-norm of Dn: 2.110874E-12.
 t( 999 ) =  0.998000000000001 :
              row= 261 , col= 11 : DOMAIN MAXIMUM TEMPERATURE =  1065.864266173153
              row= 1 , col= 2 : DOMAIN MINIMUM TEMPERATURE =  299.6501421966552
              row= 261 , col= 1  : DOMAIN MAXIMUM ERROR      =  286.7043253797016 , TEMPERATURE =  586.7043253797016 .
TIME = 9.990E-01. Newton-Kantorovich Iterations Converged after  5 iterations. Final value of L2-norm of Dn: 2.103206E-12.
 t( 1000 ) =  0.999000000000001 :
              row= 261 , col= 11 : DOMAIN MAXIMUM TEMPERATURE =  1065.099225166071
              row= 1 , col= 2 : DOMAIN MINIMUM TEMPERATURE =  299.6491821133234
              row= 261 , col= 1  : DOMAIN MAXIMUM ERROR      =  286.9849332115966 , TEMPERATURE =  586.9849332115966 .
TIME = 1.000E+00. Newton-Kantorovich Iterations Converged after  5 iterations. Final value of L2-norm of Dn: 2.036867E-12.
 t( 1001 ) =  1.0 :
              row= 261 , col= 11 : DOMAIN MAXIMUM TEMPERATURE =  1064.33494835625
              row= 1 , col= 2 : DOMAIN MINIMUM TEMPERATURE =  299.648220429031
              row= 261 , col= 1  : DOMAIN MAXIMUM ERROR      =  9.788703799522402E-03 , TEMPERATURE =  587.2655355776843 .

 -----------------------------------------------------------------------------------------------------------------------
 FINISHED DEALLOCATING ALL ARRAYS.
 OUTPUT FILE, Dgrid, CLOSED
 OUTPUT FILE, Derrg, CLOSED
 OUTPUT FILE, Dsnap, CLOSED
 OUTPUT FILE, Devol, CLOSED
 OUTPUT FILE, Dconv, CLOSED
 Program execution completed successfully. EXITING.
```

A-3.3 Implementing *COND2D*: An example run

Two examples are presented in this section, illustrating how to implement COND2D for a given problem. The process involves setting up the problem, compiling & linking the code, running the code, and processing the data in output files to check for convergence of grid functions and actual surface plots.

A-3.3.1 Example: Setting up multiple runs for a nonlinear test problem in the spherical coordinate system

Problem setup: The first stage of implementation is to setup the problem. In order to set up numerous types of test problems while minimum the scope for user error, a standard input format sheet was created for inputting the problem into *COND2D*. Such input sheets for the setup of three *Test Problems* are shown in Tables A-3, A-4 and A-5.

Compiling and linking: Once all relevant data and expressions have been introduced into the source code, the next step is to compile and link the code to create an executable file. This is a very platform and machine specific process, and users should contact their system administrator to obtain information regarding Fortran 90 compile options available at their facility. Compilers may differ widely in how strictly they interpret some fundamental Fortran 90 syntax rules. For instance, the syntax for defining the KIND parameter (that determines the precision of both real and integral variables for the run) - while some compilers accept the short or abridged form of definition, others will accept only the unabridged definition. All runs here were conducted on a HP-UX (HP Unix) platform. The runs were carried out in serial mode (since sufficient time was not available for parallelizing the code), on a single processor (with 2 Gigabytes of memory) of a 224 processor HP Superdome supercomputer cluster!

Several compiler optimization options were tested on *COND2D*, making sure that the accuracy of the program output was not compromised (this is sometimes an issue when using very high levels of optimization). The best compiler optimization option was found to be the at the highest possible on HP-UX – the *+Oall* option (see HP Fortran 90 Users Guide 1998), which reduced the program run time by about 70-80%, when compared with the non-optimized run! It was found using profiling software like *gprof* (available for use with HP-UX Fortran and C compilers) that this optimization option was inlining all subroutines into one big serial object code, and then applying parallelization to the code. It is to facilitate this maximum level of optimization that all diagnostic write statements from all subroutines were disabled (the optimizer ignores any subroutine that contains an I/O statement, thus reducing the effects of optimizing the entire code – see HP Fortran 90 Users Guide 1998). So, the optimal command line compilation & linking is obtained by using the following command:

```
$ f90 –o cond2d_test +Oall cond2d.f90
```

This generates an object file, *cond2d.o*, and an executable file, *cond2d_test*. Due to the high level of optimization being applied, compilation takes some time to be accomplished (~ a few minutes). After any reported compilation errors are corrected, and the code recompiled without further errors, it is ready to be used.

Table A- 3. Problem input sheet for *Test Problem #27*: Nonlinear spherical PDE with nonlinear Neumann/Robin boundary conditions. In all, over 30 different test problems were designed to validate *COND2D* (Table A-7). Input expressions for the code are in bold.

Solution: $\boldsymbol{u = \underline{e^{-t}}.\ \ \underline{\{x\text{-}Sin(x)\}}.\underline{\{y^2/2 + y.Sin(y) + Sin^2(y)\}}}$

$\boldsymbol{= h(t)\ .\ \ \ f(x).\ \ \ \ \ \ \ \ \ \ \ g(y)}$ **(Exact Sol Flag = 1)** (1)

$u_x\ =\ h(t)\ .\ g(y)\ .\ \{1\text{-}Cos(x)\} = h(t)\ .\ f\,'(x)\ .g(y)$ (1a)

$u_{xx} =\ h(t)\ .\ g(y)\ .\ \{Sin(x)\}\ \ \ = h(t)\ .\ f\,''(x)\ .g(y)$ (1b)

$u_y\ =\ h(t)\ .\ f(x)\ .\ \{\ y + Sin(y) + y.Cos(y) + Sin(2y)\ \}\ \ \ \ = h(t)\ .\ f(x)\ .g'(y)$ (1c)

$u_{yy} =\ h(t)\ .\ f(x)\ .\ \{\ 1 - y.Sin(y) + 2.[Cos(y) + Cos(2y)]\ \} = h(t)\ .\ f(x)\ .g''(y)$ (1d)

$u_t = -\ h(t)\ .\ f(x)\ .\ g(y) =\ -\ u$ (1e)

Initial Condition: u(t = 0) = $\boldsymbol{u_0 = \{x\text{-}Sin(x)\}.\{y^2/2 + y.Sin(y) + Sin^2(y)\}}$ (2)

$\boldsymbol{\rho = 1}$ (In module *fault_params*) (3a)

$k_t = 1+u,\ k_{t,u} = 1,\ k_{t,uu} = 0$ (**Linear Flag = 0**) (3b)

$C_p = 1+u,\ C_{p,u} = 1$ (**Linear Flag = 0**) (3c)

$a_1 = 1/x^2,\ a_2 = x^2,\ a_{2,x} = 2x$ (Spherical, **Coord. Flag = 3**) (3d)

$b_1 = 1/\{x^2.Sin(y)\},\ b_2 = Sin(y),\ b_{2,y} = Cos(y)$ (Spherical, **Coord. Flag = 3**) (3e)

<u>Boundary Conditions:</u> (**BC Flags: <u>L/R/ = 1, T/B = 2</u>**), (**BC Linearity Flags: <u>L/T/B/R = 0</u>**)

$L_1(u)\ .\ u_x + L_2(u) = k_t(u).u_x|_{(0,y)}$ = $\boldsymbol{0}$ $\boldsymbol{= f_L(y,t)}$ (4a)

$R_1(u)\ .\ u_x + R_2(u) = k_t(u).u_x|_{(1,y)}$ = $\boldsymbol{\{1+ h(t)\ .\ g(y)\ .\ (1\text{-}Sin(1)\}\ .\ h(t)\ .\ g(y)\ \ .\{1\text{-}Cos(1)\}}$ $\boldsymbol{= f_R(y,t)}$ (4b)

$B_1(u)\ .\ u_y + B_2(u) = \{k_t(u).u_y + u.(1+u)/2\}|_{(x,0)}$ = $\boldsymbol{0}$ $\boldsymbol{= f_B(x,t)}$ (4c)

$T_1(u)\ .\ u_y + T_2(u) = \{k_t(u).u_y + u.(1+u)/2)|_{(x,\pi)}$ = $\boldsymbol{.\{1+ (\pi^2/2).h(t)\ .\ f(x)\}\ .\ (\pi^2/4)\ .\ h(t)\ .\ f(x)}$ $\boldsymbol{= f_T(x,t)}$ (4d)

<u>Source Function, f_{rhs}:</u>

$$f = \rho.c_P.u_t - \left\{k_t.\left(a_1 a_{2,x}.u_x + a_1 a_2.u_{xx} + b_1 b_{2,y}.u_y + b_1 b_2.u_{yy}\right) + k_{t,u}.\left(a_1 a_2.u_x^{\ 2} + b_1 b_2.u_y^{\ 2}\right)\right\} \tag{5a}$$

OR

$$f = \rho.c_P.u_t - \left[k_t.\left\{\left(2\frac{u_x}{x} + u_{xx}\right) + \frac{1}{x^2}.\left(\frac{u_y}{Tan(y)} + u_{yy}\right)\right\} + k_{t,u}\left(u_x^{\ 2} + \frac{u_y^{\ 2}}{x^2}\right)\right] \tag{5b}$$

Therefore for **x ≠ 0, and y ≠ 0 or π,**

$$f = -\rho.(1+u)u - h(t).\left[(1+u).\left\{g(y)\left(2\frac{\{1-Cos(x)\}}{x} + Sin(x)\right) + \frac{f(x)}{x^2}.\left(\frac{g'(y)}{Tan(y)} + g''(y)\right)\right\} + h(t).\left((f'(x).g(y))^2 + \left(\frac{f(x)}{x}\right)^2.(g'(y))^2\right)\right] \tag{5c}$$

for **x ≠ 0, and y = 0 or π,**

$$f = -\rho.(1+u)u - h(t).\left[(1+u).\left\{g(y)\left(2\frac{\{1-Cos(x)\}}{x} + Sin(x)\right) + \frac{f(x)}{x^2}.(g''(y) + g''(y))\right\} + h(t).\left((f'(x).g(y))^2 + \left(\frac{f(x)}{x}\right)^2.(g'(y))^2\right)\right] \tag{5d}$$

and for **x = 0,**

$\boldsymbol{f = 0}$ (5e)

and, for **x ≠ 0, and y ≠ 0 or π,**

$$f = -\rho.(1+2u) - h(t).\left\{g(y)\left(2\frac{\{1-Cos(x)\}}{x} + Sin(x)\right) + \frac{f(x)}{x^2}.\left(\frac{g'(y)}{Tan(y)} + g''(y)\right)\right\} \tag{6a}$$

for **x ≠ 0, and y = 0 or π,**

$$f = -\rho.(1+2u) - h(t).\left\{g(y)\left(2\frac{\{1-Cos(x)\}}{x} + Sin(x)\right) + \frac{f(x)}{x^2}.(g''(y) + g''(y))\right\} \tag{6b}$$

and, for **x = 0,**

$\boldsymbol{f_u = -\rho}$ (6c)

Table A- 4. Problem input sheet for *Test Problem #32:* Nonlinear spherical PDE with linear/nonlinear Neumann boundary conditions. In all, over 30 different test problems were designed to validate *COND2D* (Table A-7). Input expressions for the code are in bold.

Solution: $\boldsymbol{u = 300 + \underline{(2.5\text{x}10^6).e^{-t}}.\underline{\{x\text{-}Sin(x)\}}.\underline{\{y^2/2 + y.Sin(y) + Sin^2(y)\}}}$

$= 300 + \quad h(t) . \quad f(x). \quad g(y)$ **(Exact Sol Flag = 1)** (1)

$u_x = h(t) . g(y) . \{1\text{-}Cos(x)\} = h(t) . f'(x) .g(y)$ (1a)

$u_{xx} = h(t) . g(y) . \{Sin(x)\} = h(t) . f''(x) .g(y)$ (1b)

$u_y = h(t) . f(x) . \{ y + Sin(y) + y.Cos(y) + Sin(2y) \} = h(t) . f(x) .g'(y)$ (1c)

$u_{yy} = h(t) . f(x) . \{ 1 - y.Sin(y) + 2.[Cos(y) + Cos(2y)] \} = h(t) . f(x) .g''(y)$ (1d)

$u_t = - h(t) . f(x) . g(y) = 300 - u$ (1e)

Initial Condition: u(t = 0) = $\boldsymbol{u_0 = 300 + (2.5\text{x}10^6).\{x\text{-}Sin(x)\}.\{y^2/2 + y.Sin(y) + Sin^2(y)\}}$ (2)

$\boldsymbol{\rho = \rho_{max}}$	(In module *fault_params)*	(3a)
$k_t = k_t(u), k_{t,u} = k_{t,u}(u), k_{t,uu} = k_{t,uu}(u)$	(Defaults for **Linear Flag = 0**)	(3b)
$C_p= C_p(u), C_{p,u}= C_{p,u}(u)$	(Defaults for **Linear Flag = 0**)	(3c)
$a_1 = 1/x^2, a_2 = x^2, a_{2,x} = 2x$	(Spherical, **Coord. Flag = 3**)	(3d)
$b_1 = 1/\{x^2.Sin(y)\}, b_2 = Sin(y), b_{2,y} = Cos(y)$	(Spherical, **Coord. Flag = 3**)	(3e)

<u>Boundary Conditions:</u> (**BC Flags: <u>L/R/T/B = 1</u>**), (**BC Linearity Flags: <u>L/T/B = 1, R = 0</u>**)

$L_1(u) . u_x + L_2(u) = 1.u_x(0,y) + 0 =$ $\boldsymbol{0}$ $\boldsymbol{= f_L(y,t)}$ (4a)

$R_1(u) . u_x + R_2(u) = k_t(u).u_x(1,y) + 0 =$ $\boldsymbol{h(t) . g(y) .\{1\text{-}Cos(1)\}}$ $\boldsymbol{= f_R(y,t)}$ (4b)

$B_1(u) . u_y + B_2(u) = 1.u_y(x,0) + 0 =$ $\boldsymbol{0}$ $\boldsymbol{= f_B(x,t)}$ (4c)

$T_1(u) . u_y + T_2(u) = 1.u_y(x,\pi) + 0 =$ $\boldsymbol{0}$ $\boldsymbol{= f_T(x,t)}$ (4d)

<u>Source Function, f_{rhs}:</u>

$$f = \rho.c_P.u_t - \left\{k_t.\left(a_1 a_{2,x}.u_x + a_1 a_2.u_{xx} + b_1 b_{2,y}.u_y + b_1 b_2.u_{yy}\right) + k_{t,u}.\left(a_1 a_2.u_x^{\ 2} + b_1 b_2.u_y^{\ 2}\right)\right\} \quad (5a)$$

OR

$$f = \rho.c_P.u_t - \left[k_t.\left\{\left(2\frac{u_x}{x} + u_{xx}\right) + \frac{1}{x^2}.\left(\frac{u_y}{Tan(y)} + u_{yy}\right)\right\} + k_{t,u}\left(u_x^{\ 2} + \frac{u_y^{\ 2}}{x^2}\right)\right] \quad (5b)$$

Therefore for **x ≠ 0, and y ≠ 0 or π,**

$$f = \rho.c_P.(300-u) - h(t).\left[k_t.\left\{g(y)\left(2\frac{\{1-Cos(x)\}}{x} + Sin(x)\right) + \frac{f(x)}{x^2}.\left(\frac{g'(y)}{Tan(y)} + g''(y)\right)\right\} + k_{t,u}.h(t).\left((f'(x).g(y))^2 + \left(\frac{f(x)}{x}\right)^2.(g'(y))^2\right)\right] \quad (5c)$$

for **x ≠ 0, and y = 0 or π,**

$$f = \rho.c_P.(300-u) - h(t).\left[k_t.\left\{g(y)\left(2\frac{\{1-Cos(x)\}}{x} + Sin(x)\right) + \frac{f(x)}{x^2}.(g''(y) + g''(y))\right\} + k_{t,u}.h(t).\left((f'(x).g(y))^2 + \left(\frac{f(x)}{x}\right)^2.(g'(y))^2\right)\right] \quad (5d)$$

and for **x = 0,**

$\boldsymbol{f = 300 .\rho.C_P}$ (5e)

and, for **x ≠ 0, and y ≠ 0 or π,**

$$f_u = \rho.c_{P,u}.(300-u) - \rho.c_P - h(t).\left[k_{t,u}.\left\{g(y)\left(2\frac{\{1-Cos(x)\}}{x} + Sin(x)\right) + \frac{f(x)}{x^2}.\left(\frac{g'(y)}{Tan(y)} + g''(y)\right)\right\} + k_{t,uu}.h(t).\left((f'(x).g(y))^2 + \left(\frac{f(x)}{x}\right)^2.(g'(y))^2\right)\right] \quad (6a)$$

for **x ≠ 0, and y = 0 or π,**

$$f_u = \rho.c_{P,u}.(300-u) - \rho.c_P - h(t).\left[k_{t,u}.\left\{g(y)\left(2\frac{\{1-Cos(x)\}}{x} + Sin(x)\right) + \frac{f(x)}{x^2}.(g''(y) + g''(y))\right\} + k_{t,uu}.h(t).\left((f'(x).g(y))^2 + \left(\frac{f(x)}{x}\right)^2.(g'(y))^2\right)\right] \quad (6b)$$

and, for **x = 0,**

$\boldsymbol{f_u = -\rho.C_P}$ (6c)

Table A- 5. Problem input sheet for Thesis problem: Nonlinear spherical PDE with linear/nonlinear Neumann boundary conditions. Input expressions for the code are in bold.

Solution: $\boldsymbol{u = UNKNOWN}$	**(Exact Sol Flag = 0)**	(1)
Initial Condition: u(t = 0) = $\boldsymbol{u_0 = 300}$		(2)
$\boldsymbol{\rho = \rho_{max}}$	(In module *fault_params)*	(3a)
$k_t = k_t(u)$, $k_{t,u} = k_{t,u}(u)$, $k_{t,uu} = k_{t,uu}(u)$	(Defaults for **Linear Flag = 0**)	(3b)
$C_p = C_p(u)$, $C_{p,u} = C_{p,u}(u)$	(Defaults for **Linear Flag = 0**)	(3c)
$a_1 = 1/x^2$, $a_2 = x^2$, $a_{2,x} = 2x$	(Spherical, **Coord. Flag = 3**)	(3d)
$b_1 = 1/\{x^2.Sin(y)\}$, $b_2 = Sin(y)$, $b_{2,y} = Cos(y)$	(Spherical, **Coord. Flag = 3**)	(3e)

<u>Boundary Conditions:</u> (**BC Flags: <u>L/R/T/B = 1</u>**), (**BC Linearity Flags: <u>L/T/B = 1, R = 0</u>**)

$L_1(u) \,.\, u_x + L_2(u) = 1.u_x(0,y) + 0 =$ $\boldsymbol{0}$ $\boldsymbol{= f_L(y,t)}$ (4a)

$R_1(u) \,.\, u_x + R_2(u) = k_t(u).u_x(0.1,y) + 0$

$= (\tau.V_{slip}/4)\boldsymbol{.[Tanh\{1000.(y - y_0)\} - Tanh(1000.y)].[Tanh\{1000.(t - t_0)\} - Tanh(1000.t)]}$ $\boldsymbol{= f_R(y,t)}$ (4b)

$B_1(u) \,.\, u_y + B_2(u) = 1.u_y(x,0) + 0 =$ $\boldsymbol{0}$ $\boldsymbol{= f_B(x,t)}$ (4c)

$T_1(u) \,.\, u_y + T_2(u) = 1.u_y(x,\pi) + 0 =$ $\boldsymbol{0}$ $\boldsymbol{= f_T(x,t)}$ (4d)

<u>Source Function, f_{rhs}:</u>

$\boldsymbol{f = 0}$ (5a)

and,

$\boldsymbol{f_u = 0}$ (5b)

<u>Running the executable file:</u> To run the code for short duration runs (reasonably small domain size, short temporal range, and coarse resolutions – typically, taking less than 2 hours to run to completion) type the name of the executable generated above, followed by the four arguments as discussed in Sections A-3.2.5.1 and A-3.2.5.2. It may be better to run the code in the background, piping the screen output to another file for later use, and leaving the terminal free for other things. In addition, the command itself can be timed, by using the UNIX *time* command. So, a very general run command for background execution is:

```
$ time cond2d_test 1 2 0 000000 > OUT_1_2_0_000000 &
```

The above command is useful when the run time for the code is small. Most large shared systems place a limit on the length of time a program job can be run from a user terminal. So, most extensive runs of the code (especially higher resolutions runs that can take days in serial mode) need to be submitted through a utility called Load Sharing Facility or LSF. This allows multiple user jobs of any length to be submitted to a serial or parallel queue, and the jobs are executed in the background even when the user is not logged on. In addition, LSF has options for emailing the user when the job starts and ends. When submitting a job via LSF, the first step is to create a script file containing commands to move the code, and relevant files to the scratch space (a common workspace allocated for all users for temporary storage of job output) and then execute the code from there. Commands to copy all relevant output back to the user directory can be included to automate the whole process. This is especially useful when a large number of runs are submitted in multiple jobs, and the script file can be automated to do all the bookkeeping, sorting and moving the files, saving a significant amount of time.

A typical UNIX script file, *T27script*, for submitting a multi-run job via LSF is presented in Figure A-7. This script was created to run *Test Problem #27*, presented in Table A-3 above, for testing convergence at successive higher resolution runs. Once the file is created and is given execute permission by the user (by using the *chmod* UNIX command), the following command submits the script file to the LSF serial queue, and sends email at the start and end of the job:

```
$ bsub –B –N –o RUN_OUTPUT –q serial T27script &
```

The file *RUN_OUTPUT* will contain all error messages, and code runtime information (if specified using the *time* UNIX command).

<u>Processing and analyzing code output:</u> The code output can be processed and analyzed in a number of ways. Analysis of grid function convergence involves checking how fast the errors - at each one of successively higher resolutions - are getting smaller (see McDonough 2001 for a description of grid function convergence tests). Convergence, evolution, and profile data can be imported to a spreadsheet program like MS-EXCEL and more advanced post-processing programs like MATLAB. The most rigorous convergence test is shown in Table A-6, for *Test Problem #27* (presented in Table A-3), at grid nodes specified in the array *grid_conv* in the module *const_params* (Sections 3.2.1 and 3.2.5.3 above). This data is output to the *<u>Dconv</u>* file. This table also shows typical convergence metrics (headings for the last four columns). Once it has been determined from such a test that COND2D is converging for the problem of interest, other visual aids can be used to record this convergence for different parts of the problem domain. Figures A-8, A-9, and A-10 are profile and evolution plots that can be used to check convergence of grid functions along different "slices" of the spatio-temporal domain of the problem. Figure A-8 was generated using output data in the *<u>Dsnap</u>* file. Figures A-9 and A-10 were generated from data in the *<u>Devol</u>* file. Data in the *<u>Dgrid</u>* and *<u>Derrg</u>* files can be used to generate surface/contour plots - as shown for grid functions in Figure A-11 (using MATLAB, here). **NOTE:** All the above tests can be carried out even when the exact solution is <u>not known</u>. In that case, it must be checked that successively higher resolutions produce converging sequences of grid functions (numerical solutions).

Figure A- 7. UNIX script, *T27script*, for submitting multiple runs for *Test Problem #27* (Table 3) as an LSF job:

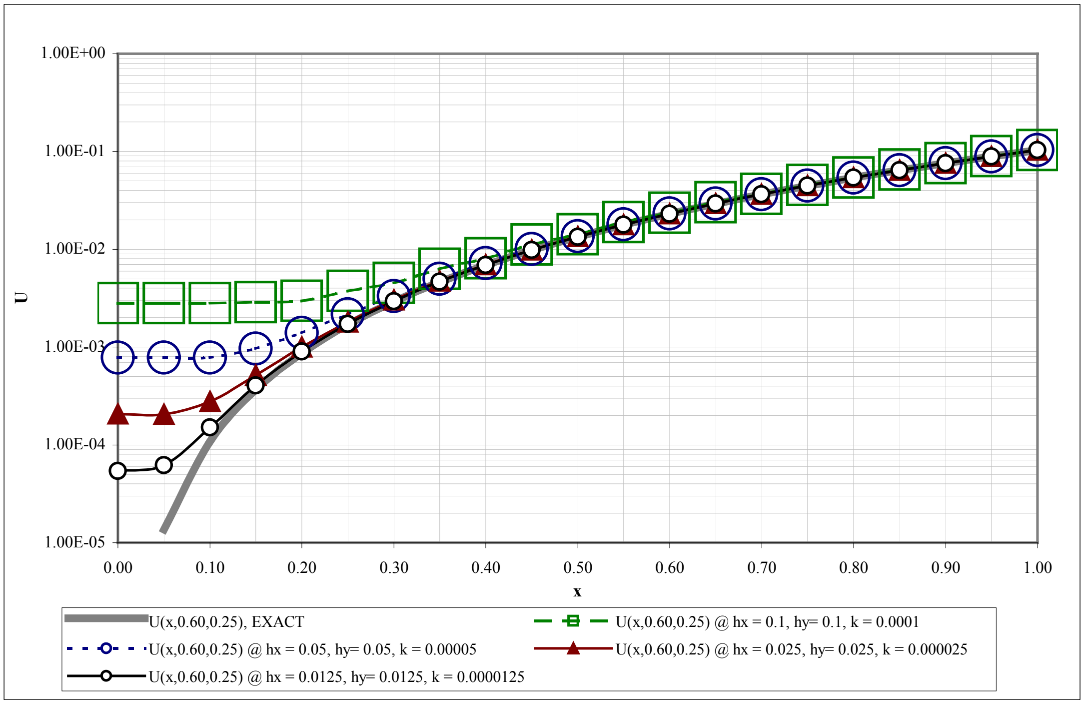
```
mkdir /scratch/rvkand2/T27_SmtF_05
cp ./qldgts /scratch/rvkand2/T27_SmtF_05
cp ./*.mod /scratch/rvkand2/T27_SmtF_05
cp ./qlindgts.o /scratch/rvkand2/T27_SmtF_05

cd /scratch/rvkand2/T27_SmtF_05
time ./qldgts 2 4 > ScreenOutput_05_4

cd
/u/home1/rvkand2/algorithms/PDE_SOLVERS/PARABOLIC_PDE/2D_Parabolic/QLin_DGTS_Lin_NonLin/TESTS/T27_SmtF_NonlinSp
hNeuRob
cp /scratch/rvkand2/T27_SmtF_05/ScreenOutput_05_4 ./
cp /scratch/rvkand2/T27_SmtF_05/Dconv ./Dconv_0.05_4
cp /scratch/rvkand2/T27_SmtF_05/Devol ./Devol_0.05_4
cp /scratch/rvkand2/T27_SmtF_05/Dgrid ./Dgrid_0.05_4
cp /scratch/rvkand2/T27_SmtF_05/Dplot ./Dplot_0.05_4
cp /scratch/rvkand2/T27_SmtF_05/Dsnap ./Dsnap_0.05_4

rm -rf /scratch/rvkand2/T27_SmtF_05
echo " "
echo "******************************** Run QLDGTS_05 Completed. ***********************************"
echo " "

#-----------------------------------------------------------------------------------------------
mkdir /scratch/rvkand2/T27_SmtF_025
cp ./qldgts /scratch/rvkand2/T27_SmtF_025
cp ./*.mod /scratch/rvkand2/T27_SmtF_025
cp ./qlindgts.o /scratch/rvkand2/T27_SmtF_025

cd /scratch/rvkand2/T27_SmtF_025
time ./qldgts 3 4 > ScreenOutput_025_4

cd
/u/home1/rvkand2/algorithms/PDE_SOLVERS/PARABOLIC_PDE/2D_Parabolic/QLin_DGTS_Lin_NonLin/TESTS/T27_SmtF_NonlinSp
hNeuRob
cp /scratch/rvkand2/T27_SmtF_025/ScreenOutput_025_4 ./
cp /scratch/rvkand2/T27_SmtF_025/Dconv ./Dconv_0.025_4
cp /scratch/rvkand2/T27_SmtF_025/Devol ./Devol_0.025_4
cp /scratch/rvkand2/T27_SmtF_025/Dgrid ./Dgrid_0.025_4
cp /scratch/rvkand2/T27_SmtF_025/Dplot ./Dplot_0.025_4
cp /scratch/rvkand2/T27_SmtF_025/Dsnap ./Dsnap_0.025_4

rm -rf /scratch/rvkand2/T27_SmtF_025
echo " "
echo "******************************** Run QLDGTS_025 Completed. ***********************************"
echo " "

#-----------------------------------------------------------------------------------------------
mkdir /scratch/rvkand2/T27_SmtF_0125
cp ./qldgts /scratch/rvkand2/T27_SmtF_0125
cp ./*.mod /scratch/rvkand2/T27_SmtF_0125
cp ./qlindgts.o /scratch/rvkand2/T27_SmtF_0125

cd /scratch/rvkand2/T27_SmtF_0125
time ./qldgts 4 4 > ScreenOutput_0125_4

cd
/u/home1/rvkand2/algorithms/PDE_SOLVERS/PARABOLIC_PDE/2D_Parabolic/QLin_DGTS_Lin_NonLin/TESTS/T27_SmtF_NonlinSp
hNeuRob
cp /scratch/rvkand2/T27_SmtF_0125/ScreenOutput_0125_4 ./
cp /scratch/rvkand2/T27_SmtF_0125/Dconv ./Dconv_0.0125_4
cp /scratch/rvkand2/T27_SmtF_0125/Devol ./Devol_0.0125_4
cp /scratch/rvkand2/T27_SmtF_0125/Dgrid ./Dgrid_0.0125_4
cp /scratch/rvkand2/T27_SmtF_0125/Dplot ./Dplot_0.0125_4
cp /scratch/rvkand2/T27_SmtF_0125/Dsnap ./Dsnap_0.0125_4

rm -rf /scratch/rvkand2/T27_SmtF_0125
echo " "
echo "******************************** Run QLDGTS_0125 Completed. ***********************************"
echo " "
```

Table A- 6. Grid function convergence tests for the nonlinear problem in Spherical system, *Test Problem #27*, generated from the output of the *Dconv* files produced after executing the script file *T27script* (Figure 7).

Newton-Kantorovich with Douglass-Gunn Time Splitting: Grid Convergence Tests for T27 NonlinSphNeuRob: *U(x,y,t) = (e^{-t}).{x - SIN(x)}.{ $(y^2/2)$ + y.SIN(y) + $SIN^2(y)$ }*						
Grid Resolution Relationships	Grid function Resolutions & Coordinates	U(x,y)	Absolute Grid Function Errors (W.R.T Exact Solution)	Absolute "Cauchy" Grid Function Errors (W.R.T Next Lower H Value)	Theoretical (Based on Absolute Errors) $R = E_h/E_{h/2}$	Computationally Observed (Based on "Cauchy" errors): $R' = e_h/e_{h/2}$
			E= ABS{U_{exact}(i,j) - u(i,j)}	e = ABS{$u^{(m+1)}$(i,j) - $u^{(m)}$(i,j)}	$R = 2^{N\_theo}$	$R' = 2^{N\_comp}$
x =	**0.20**					
y =	**0.10**					
t =	**0.25**					
k=0.001*hx=0.01*hy	**0.1**	2.05699284E-03	2.03114E-03	1.53474E-03	**4.09**	4.1
k=0.001*hx=0.01*hy	**0.05**	5.22251308E-04	4.96395E-04	3.73212E-04	**4.03**	4.1
k=0.001*hx=0.01*hy	**0.025**	1.49039017E-04	1.23183E-04	9.20911E-05	**3.96**	
k=0.001*hx=0.01*hy	**0.0125**	5.69478855E-05	3.10915E-05			
k=0.001*hx=0.01*hy	**0.00625**					
	EXACT SOLUTION	**2.58563938E-05**				
x =	**0.60**					
y =	**0.30**					
t =	**0.25**					
k=0.001*hx=0.01*hy	**0.1**	6.53711900E-03	4.51883E-04	3.38814E-04	**4.00**	4.0
k=0.001*hx=0.01*hy	**0.05**	6.19830516E-03	1.13069E-04	8.43551E-05	**3.94**	4.0
k=0.001*hx=0.01*hy	**0.025**	6.11395001E-03	2.87137E-05	2.12237E-05	**3.83**	
k=0.001*hx=0.01*hy	**0.0125**	6.09272634E-03	7.49003E-06			
k=0.001*hx=0.01*hy	**0.00625**					
	EXACT SOLUTION	**6.08523631E-03**				
x =	**0.50**					
y =	**0.50**					
t =	**0.25**					
k=0.001*hx=0.01*hy	**0.1**	1.03313012E-02	8.04399E-04	6.02856E-04	**3.99**	4.0
k=0.001*hx=0.01*hy	**0.05**	9.72844501E-03	2.01542E-04	1.50601E-04	**3.96**	4.0
k=0.001*hx=0.01*hy	**0.025**	9.57784421E-03	5.09417E-05	3.78312E-05	**3.89**	
k=0.001*hx=0.01*hy	**0.0125**	9.54001304E-03	1.31105E-05			
k=0.001*hx=0.01*hy	**0.00625**					
	EXACT SOLUTION	**9.52690256E-03**				
x =	**0.70**					
y =	**0.80**					
t =	**0.25**					
k=0.001*hx=0.01*hy	**0.1**	6.16307285E-02	4.41495E-04	3.29636E-04	**3.95**	4.0
k=0.001*hx=0.01*hy	**0.05**	6.13010921E-02	1.11859E-04	8.31279E-05	**3.89**	3.9
k=0.001*hx=0.01*hy	**0.025**	6.12179641E-02	2.87311E-05	2.11303E-05	**3.78**	
k=0.001*hx=0.01*hy	**0.0125**	6.11968338E-02	7.60086E-06			
k=0.001*hx=0.01*hy	**0.00625**					
	EXACT SOLUTION	**6.11892330E-02**				
x =	**0.40**					
y =	**0.60**					
t =	**0.50**					
k=0.001*hx=0.01*hy	**0.1**	6.62660127E-03	1.25076E-03	9.39176E-04	**4.01**	4.0
k=0.001*hx=0.01*hy	**0.05**	5.68742569E-03	3.11583E-04	2.32832E-04	**3.96**	4.0
k=0.001*hx=0.01*hy	**0.025**	5.45459387E-03	7.87511E-05	5.83731E-05	**3.86**	
k=0.001*hx=0.01*hy	**0.0125**	5.39622080E-03	2.03780E-05			
k=0.001*hx=0.01*hy	**0.00625**					
	EXACT SOLUTION	**5.37584281E-03**				
x =	**0.10**					
y =	**0.40**					
t =	**0.50**					
k=0.001*hx=0.01*hy	**0.1**	2.43444623E-03	2.39530E-03	1.80100E-03	**4.03**	4.0
k=0.001*hx=0.01*hy	**0.05**	6.33447862E-04	5.94304E-04	4.45441E-04	**3.99**	4.0
k=0.001*hx=0.01*hy	**0.025**	1.88006894E-04	1.48863E-04	1.10923E-04	**3.92**	
k=0.001*hx=0.01*hy	**0.0125**	7.70838936E-05	3.79404E-05			
k=0.001*hx=0.01*hy	**0.00625**					
	EXACT SOLUTION	**3.91434995E-05**				
x =	**0.90**					
y =	**0.90**					
t =	**0.50**					
k=0.001*hx=0.01*hy	**0.1**	1.22306655E-01	3.35046E-04	2.50567E-04	**3.97**	4.0
k=0.001*hx=0.01*hy	**0.05**	1.22056088E-01	8.44790E-05	6.23371E-05	**3.82**	3.9
k=0.001*hx=0.01*hy	**0.025**	1.21993751E-01	2.21419E-05	1.59770E-05	**3.59**	
k=0.001*hx=0.01*hy	**0.0125**	1.21977774E-01	6.16484E-06			
k=0.001*hx=0.01*hy	**0.00625**					
	EXACT SOLUTION	**1.21971609E-01**				
x =	**0.80**					
y =	**0.70**					
t =	**0.50**					
k=0.001*hx=0.01*hy	**0.1**	5.61258534E-02	4.37359E-04	3.27684E-04	**3.99**	4.0
k=0.001*hx=0.01*hy	**0.05**	5.57981690E-02	1.09675E-04	8.13212E-05	**3.87**	3.9
k=0.001*hx=0.01*hy	**0.025**	5.57168478E-02	2.83533E-05	2.06636E-05	**3.69**	
k=0.001*hx=0.01*hy	**0.0125**	5.56961841E-02	7.68971E-06			
k=0.001*hx=0.01*hy	**0.00625**					
	EXACT SOLUTION	**5.56884944E-02**				

Figure A- 8. Snapshots of profiles along the principal axes, for the nonlinear problem in Spherical system, *Test Problem #27* (Table 3). (a) Snapshot profile parallel to the x-axis, at y = 0.60, t = 0.25. (b) Snapshot profile parallel to the y-axis, at x = 0.30, t = 0.50. Data from *Dsnap* output file.

(a)



(b)

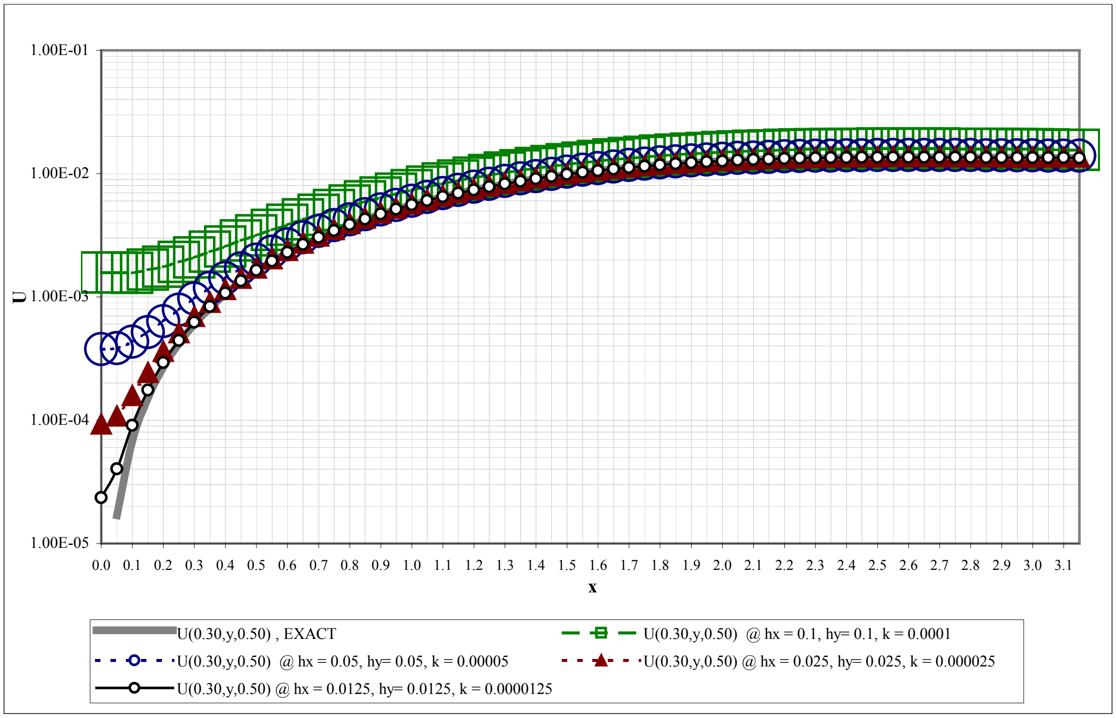


Figure A- 9. Evolution of grid functions with time, for the nonlinear problem in Spherical system, *Test Problem #27* (Table 3): x = 0.5, y = 0.5. Data from *Devol* output file.

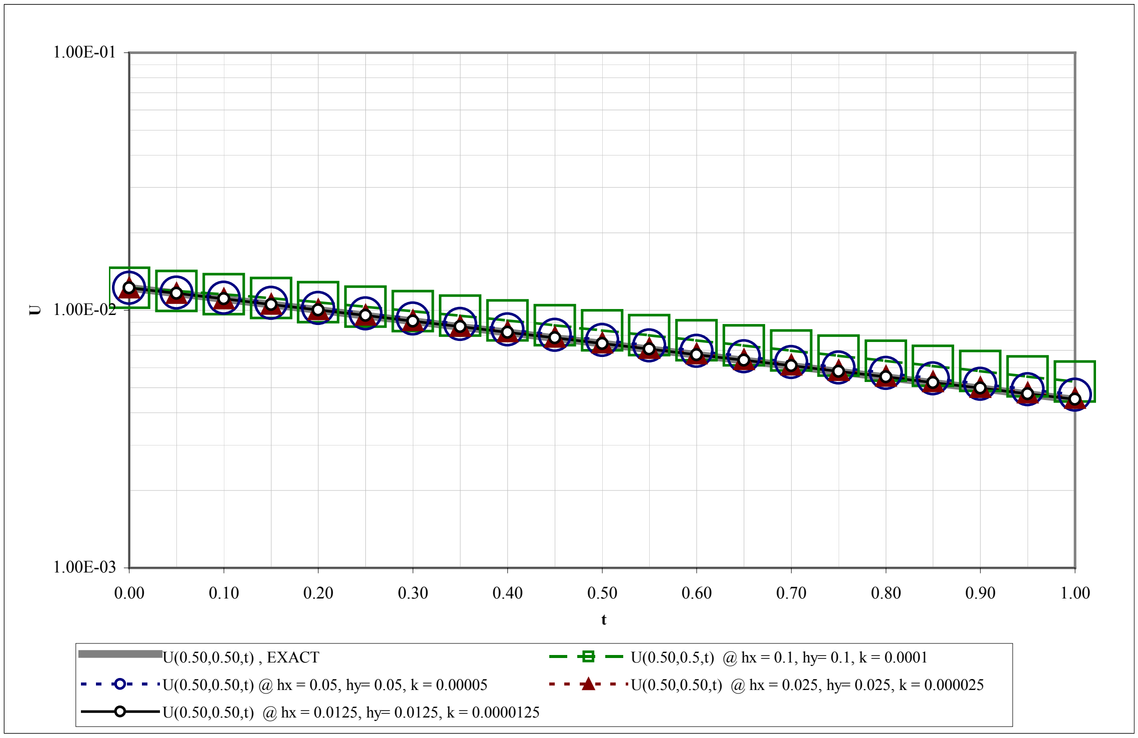
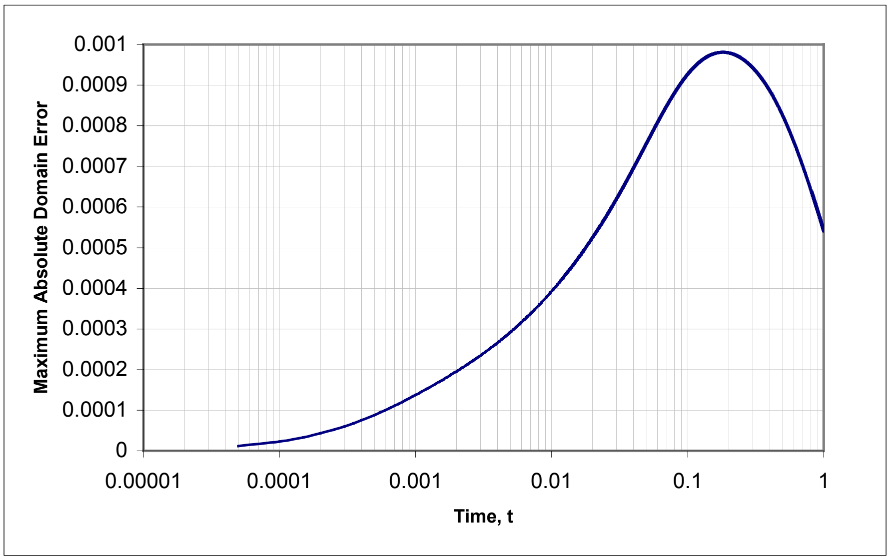


Figure A- 10. Evolution of maximum grid function error with time, for the nonlinear problem in Spherical system, *Test Problem #27* (Table 3), at a resolution of *hx = hy = 0.05*. (a) Peak Error (at the origin, *x = 0*): For this spherical system problem the peak error is primarily made up of truncation error at *x=0*, since the value of the solution here is *0* (zero). (b) Grid function maxima (at the boundary, *x = 1 & y = 2.6*): As a comparison, the temporal grid-function domain maximum occurs at *t = 0*, and has a magnitude of *~0.790433.. at (x,y) = (1.0, 2.6)*. The grid-function domain maximum at the time of peak error is *~0.615556.. at (x,y) = (1.0, 2.6)*. Thus, even though the maximum error and maximum grid-function value do not coincide in space, the former is still only *~0.16%* of this value. The maximum error at the peak grid function values is, however, much smaller, ~0.01% at its maximum. Thus, as expected, where the value of the grid function is comparable to the grid resolution, the accuracy of the numerical solution is affected. That is why, an optimal grid resolution is important for any problem. All data for these plots were obtained from the *Devol* output file.

(a)



(b)

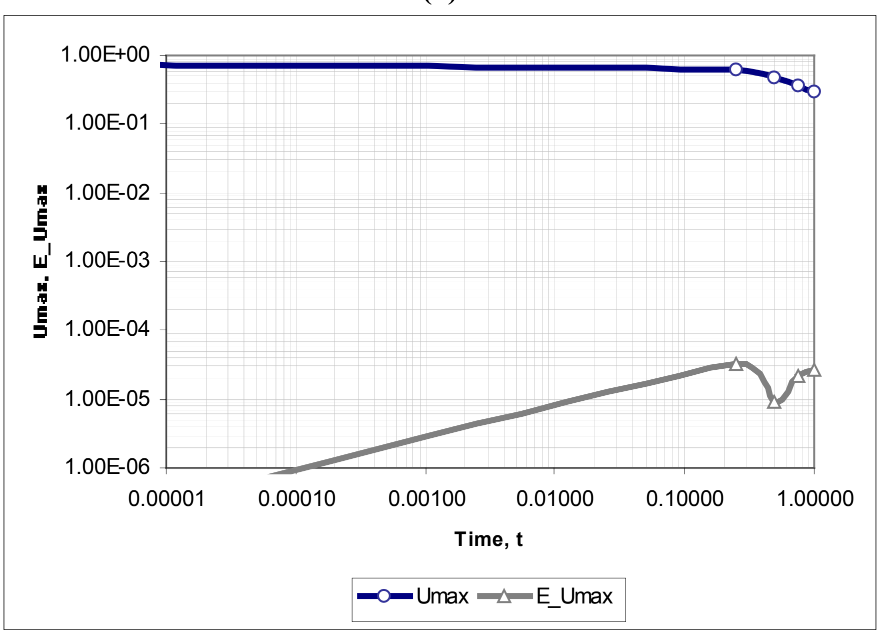
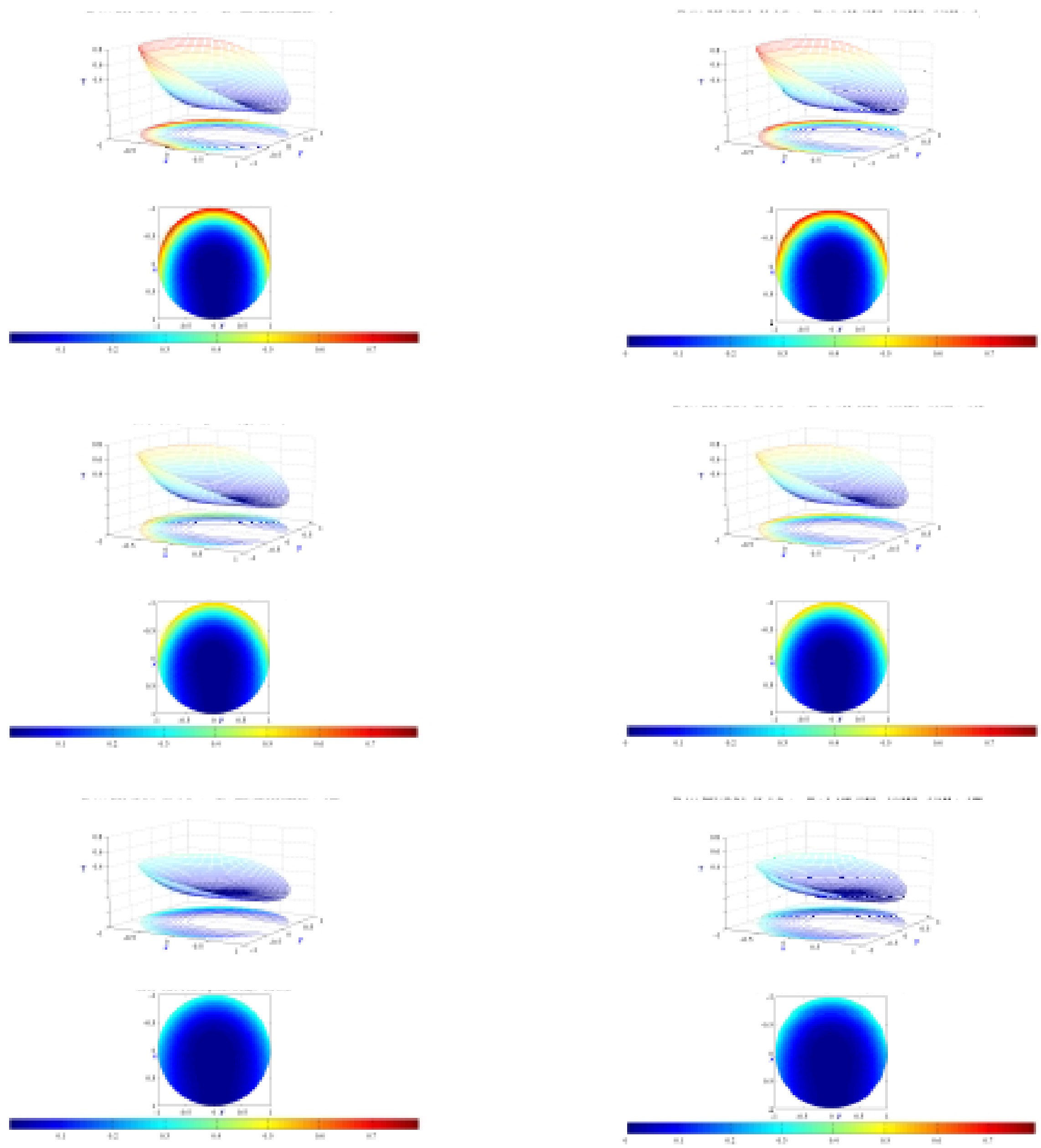


Figure A- 11. Surface contour plots comparing the analytical (exact) and numerical solutions at specific times, for the nonlinear problem in Spherical system, *Test Problem #27* (Table 3). As can be seen, at the resolution of these plots, the analytical and numerical solutions are identical at time = 0.0, 0.50 and 0.75.

ANALYTICAL **NUMERICAL**



A-3.4 *COND2D* validation tests

In order to test the validity of *COND2D*, three levels of test problems were designed – constant solution problems, linear solution problems, and "smooth" solution problems. So, using a limited set of solutions, a large number of different test problems can be generated by changing the boundary conditions, and the nonlinear thermal property functionals.

The variation of thermal conductivity, k_t, and specific heat, C_P, of quartz (the chief type of mineral in which frictional melts are found) with temperature were found to be well fitted by straight lines of different slopes in thermal zones corresponding to the two quartz phases. So, some of these tests used just a general linear function for these parameters. The one was added to make sure these functions do not become zero inside the domain, which is not realistic. For the nonlinear problem, such an abrupt profile for k_t and C_P meant using much higher resolutions. So, another alternative was sought that would vary both parameters smoothly over the temperature range of most geological conduction problems (300-3000 K), and at the same time, be Lipschitz continuous in this range (as discussed in Chapter A-1). This led, after some trial and error, to the two curves presented in Figures C-1 and C-2 of Appendix C.

The first two sets of tests – constant and linear solution problems - make it easier to identify any fundamental bugs in the code by, ensuring that either all the derivatives, or all but the first derivatives (respectively) of the solution are *0* (zero). The smooth solutions were designed to test convergence rates using "well-behaved" solutions. Different smooth solutions were generated to satisfy the symmetry conditions for cylindrical and spherical coordinate systems. In order to confirm that the code really works for the range of coordinate systems and BCs, for both linear and non-linear problems, a number of tests were conducted, that tested different loops in various *COND2D* subroutine. The total number of tests required was considerably reduced by taking advantage of the generality of the problem posed, {Equation (15a)}, and recognizing the following relationships between the three different parts of the code:

- The linear functional (PDE), Equation (15a') is a special case of the nonlinear functional (15a) – so attempting a linear problem with identical solution and boundary conditions (BCs) first can identify any basic problems with the code. The additional loops and subroutines for the nonlinear problem can then be tested "on top" of the linear test.
- Similarly, the Cartesian coordinate system yields the simplest PDE. Once the code has been tested for this system, for different linear/nonlinear BCs, most of the basic coordinate independent loops and most of the BC loops will have been tested.
- The boundary condition loops and subroutines are completely independent of the coordinate system specific loops and subroutines – so every combination of boundary condition and coordinate system need not be tested.
- The Robin and Neumann BC loops, as well as the cylindrical and spherical system loops have a lot in common – so as long as each one of them is tested once (or twice), only one of each pair need be tested thoroughly in subsequent runs. So, the tests shown in Table A-7 do not have as many Robin BC runs or Cylindrical system runs, but they do appear at least twice, to make sure that the code specific to these components does indeed work.

Tests conducted based on these very general rules are summarized in Table A-7. To limit the size of this document, only some key test results are shown here, as indicated by bolded rows in Table A-7. As the problem complexity increased, bugs were frequently detected in the new parts of the code that was being tested for the first time. When this happened, the bugs were rectified, and the code was re-run, or new problems generated – so a large number of tests had to be conducted in the end. Thus, in Table A-7, some runs share nearly identical problem data. The convergence tests appear in Tables A-8 to A-12, and relevant plots in Figures A-12 to A-23. Detailed results from *Test Problem #27* were presented in Section A-3.3 (and discussed in Figure A-10).

Table A- 7. Summary of validation tests conducted on *COND2D*. Second order convergence of Douglas-Gunn scheme, and quadratic convergence of the nonlinear iterations, were observed. In all cases. Rows in bold indicate tests for which convergence test data is presented in this document.

#	TP[$]	PDE Type	Coordinate System	Boundary Conditions	Boundary Condition Type	k_t & C_P	Exact Solution, u = f(x,y,t)
1	1	Linear	Cartesian	All: Dirichlet	Linear	k_t & C_P: Constants	
2	2	Nonlinear	Cartesian	All: Dirichlet	Nonlinear	$k_t = 1+u$, $C_P = 1+u$	
3	3	Linear	Cartesian	All: Neumann	Linear	k_t & C_P: Constants	
4	4	Nonlinear	Cartesian	All: Neumann	Nonlinear	$k_t = 1+u$, $C_P = 1+u$	
5	5	Linear	Cartesian	All: Robin	Linear	k_t & C_P: Constants	
6	6	Nonlinear	Cartesian	All: Robin	Nonlinear	$k_t = 1+u$, $C_P = 1+u$	1
7	7	Linear	Cylindrical	All: Neumann	Linear	k_t & C_P: Constants	
8	8	Nonlinear	Cylindrical	All: Neumann	Nonlinear	$k_t = 1+u$, $C_P = 1+u$	
9	9	Linear	Spherical	All: Dirichlet	Linear	k_t & C_P: Constants	
10	10	Nonlinear	Spherical	All: Dirichlet	Nonlinear	$k_t = 1+u$, $C_P = 1+u$	
11	11	Linear	Spherical	All: Neumann	Linear	k_t & C_P: Constants	
12	12	Nonlinear	Spherical	All: Neumann	Nonlinear	$k_t = 1+u$, $C_P = 1+u$	
13	13	Linear	Cartesian	All: Dirichlet	Linear	k_t & C_P: Constants	
14	14	Nonlinear	Cartesian	All: Dirichlet	Nonlinear	$k_t = 1+u$, $C_P = 1+u$	(x+y)t
15	15	Linear	Cartesian	All: Neumann	Linear	k_t & C_P: Constants	
16	16	Nonlinear	Cartesian	All: Neumann	Nonlinear	$k_t = 1+u$, $C_P = 1+u$	
17	**17**	**Nonlinear**	**Cartesian**	**L/R: Dirichlet T/B: Neumann**	**Nonlinear**	$\mathbf{k_t = 1+u, C_P = 1+u}$	
18	21			All: Dirichlet			***1 -*** $e^{-\pi^2 .t}$ ***.Sin(πx).Sin(πy)***
19	22	Linear	Cartesian	All: Neumann	Linear	k_t & C_P: Constants	
20	25	Nonlinear	Cartesian	All: Dirichlet	Nonlinear	$k_t = 1+u$, $C_P = 1+u$	
21	18	Nonlinear	Cylindrical	L/R: Neumann T/B: Dirichlet	Nonlinear	$k_t = 1+u$, $C_P = 1+u$	***1 –*** $e^{-t}.x^2.Cos(y)$
22	19	Nonlinear	Spherical	L/R: Dirichlet T/B: Neumann			
23	**23**	**Linear**	**Cylindrical**	**All: Neumann**	**Linear**	$\mathbf{k_t}$ **&** $\mathbf{C_P}$**: Constants**	***1 -*** e^{-t} ***.{x -Sin(x)}.Sin(y)***
24	24	Linear	Spherical				*1 -* e^{-t} *.{x -Sin(x)}.Cos(y)*
25	26	Nonlinear	Cylindrical	L/R:Neumann T/B:Dirichlet	Nonlinear	$k_t = 1+u$, $C_P = 1+u$	e^{-t} ***.{x -Sin(x)}. {y -Sin(y)}***
26	**27**	**Nonlinear**	**Spherical**	**L/R: Neumann T/B: Robin**	**Nonlinear**		e^{-t} ***.{x -Sin(x)}. {(*** y^2 ***/2) +y.Sin(y) +*** $Sin^2(y)$***}***
27	**28**	**Linear**	**Cartesian**	**All: Dirichlet**	**Linear**	$\mathbf{k_t}$ **&** $\mathbf{C_P}$**: Constants**	***Solution from Carslaw & Jaegar 1959: Sec. 5.6, p. 173.****
28	**29**	**Linear**	**Spherical**	**L/T/B: Neumann R: Dirichlet**	**Linear**		***Solution from Carslaw & Jaegar 1959: Sec. 9.11, p. 248-250.*****
29	30a				Nonlinear	$k_t = A/u^{\alpha}$, $C_P = B.Ln(u) + C$	
30	30c				Nonlinear	Figures C-1 & C-2: $k_t = 1 + A.e^{-Bu}$, $C_P = C.\{1 - D.e^{-Eu}\}$ *Non Differentiable at u= 300 K*	*300 + [(2.5x*10^6*).* e^{-t} *.{x -Sin(x)}. {(*y^2*/2) +y.Sin(y) +* $Sin^2(y)$*}]*
31	30b, 30g, 31			All: Neumann	L/T/B: Linear R: Nonlinear		
32	30d	Nonlinear	Spherical		Nonlinear		
33	30f				L/T/B: Linear R: Nonlinear	$k_t = 1+u$, $C_P = 1+u$	e^{-t} ***.{x -Sin(x)}. {(*** y^2 ***/2) +y.Sin(y) +*** $Sin^2(y)$***}***
34	30e			L/T/B: Neumann R: Dirichlet	Linear		
35	**32**			**All: Neumann**	**L/T/B: Linear R: Nonlinear**	**Figures C-1 & C-2:** $\mathbf{k_t = 1 + A.e^{-Bu}}$, $\mathbf{C_P = C.\{1 - D.e^{-Eu}\}}$ ***Differentiable at u = 300 K***	***300 + [(2.5x***10^6***).*** e^{-t} ***.{x -Sin(x)}. {(*** y^2 ***/2) +y.Sin(y) +*** $Sin^2(y)$***}]***

Table A-7. (Continued)

$ TP = Test Problem Number. This was the sequence in which actual tests were done.

* $\boxed{\sum_{0}^{\infty}\sum_{0}^{\infty}\frac{(-1)^{m+n}}{(2n+1).(2m+1)}.e^{-\left[\frac{\pi^2}{4}\left\{\left(\frac{2n+1}{l}\right)^2+\left(\frac{2m+1}{b}\right)^2\right\}.\kappa.t\right]}.Cos\left\{\frac{(2n+1).\pi.x}{2l}\right\}.Cos\left\{\frac{(2m+1).\pi.y}{2b}\right\}}$

** $\boxed{\sum_{n=0}^{\infty}\sum_{m=1}^{\infty}A_{nm}.e^{-\left[\frac{\alpha_{nm}^2.\kappa.t}{a^2}\right]}.j_n\left(\frac{\alpha_{nm}.r}{a}\right).P_n\{Cos(\theta)\},\text{ with } A_{nm}=\frac{(2n+1)}{a^3.j_{n+1}^2(\alpha_{nm})}\int_0^a\int_0^\pi\left\{f(r,\theta).j_n\left(\frac{\alpha_{nm}.r}{a}\right).P_n\{Cos(\theta)\}\right\}.r^2dr.Sin(\theta).d\theta}$

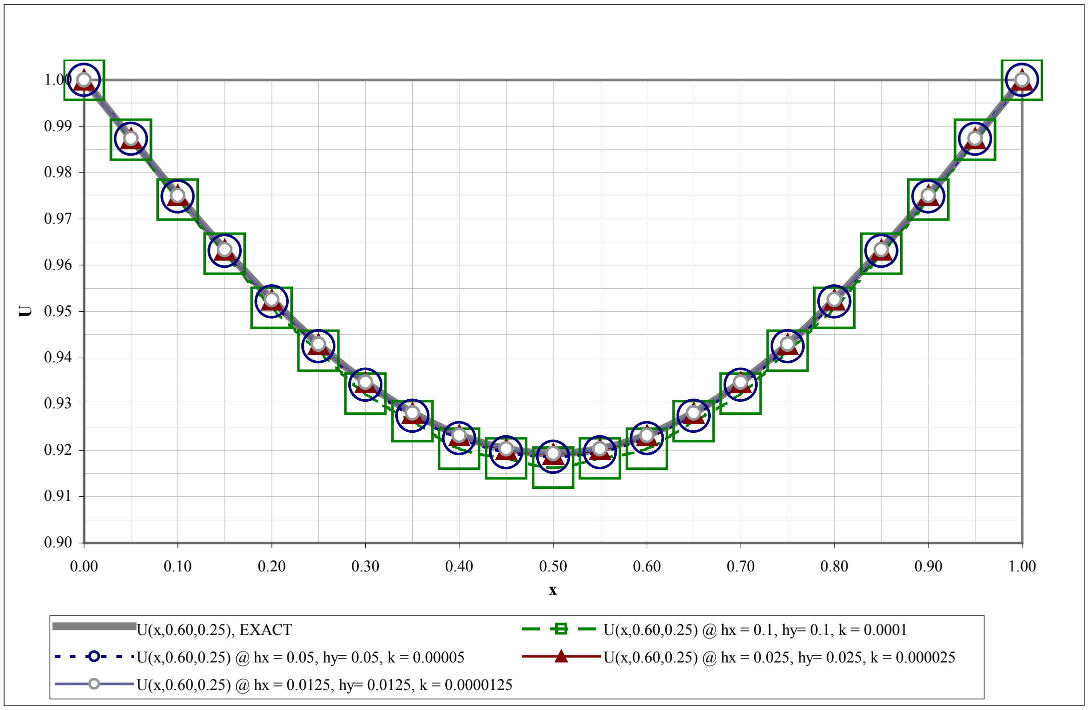
j_n = spherical Bessel function of order n, α_{nm} = m^{th} root of the n^{th} order spherical Bessel function, and P_n = Legendre function of order n.

Table A- 8. Grid function convergence tests for the nonlinear problem in Cartesian system, *Test Problem #17* (Table 7), generated from the output of the corresponding *Dconv* files.

Newton-Kantorovich with Douglass-Gunn Time Splitting: Grid Convergence Tests for T27_NonlinSphNeuRob: *U(x,y,t) = 1 - (e^{-π²t}). SIN(πx). SIN(πy)*						
Grid Resolution Relationships	**Grid function Resolutions & Coordinates**	**U(x,y)**	**Absolute Grid Function Errors**	**Absolute "Cauchy" Grid Function Errors (W.R.T Next Lower H**	**Theoretical (Based on Absolute Errors) $R = E_h/E_{h/2}$**	**Computationally Observed (Based on "Cauchy" errors): $R' = e_h/e_{h/2}$**
			$E = ABS\{U_{exact}(i,j) - u(i,j)\}$	**$e = ABS\{u^{(m+1)}(i,j) - u^{(m)}(i,j)\}$**	**$R = 2^{N\_theo}$**	**$R' = 2^{N\_comp}$**
x =	**0.20**					
y =	**0.10**					
t =	**0.25**					
k=0.001*hx=0.01*hy	**0.1**	0.9835007143	1.09568E-03	8.38531E-04	**4.26**	4.1
k=0.001*hx=0.01*hy	**0.05**	0.9843392449	2.57150E-04	2.04260E-04	**4.86**	4.5
k=0.001*hx=0.01*hy	**0.025**	0.9845435045	5.28907E-05	4.55489E-05	**7.20**	0.0
k=0.001*hx=0.01*hy	**0.0125**	0.9845890534	7.34182E-06	9.84589E-01	**0.00**	
k=0.001*hx=0.01*hy	**0.00625**	0.0000000000	9.84596E-01			
	EXACT SOLUTION	**0.9845963952**				
x =	**0.60**					
y =	**0.30**					
t =	**0.25**					
k=0.001*hx=0.01*hy	**0.1**	0.9320613431	2.68794E-03	2.04978E-03	**4.21**	4.1
k=0.001*hx=0.01*hy	**0.05**	0.9341111237	6.38159E-04	4.98404E-04	**4.57**	4.3
k=0.001*hx=0.01*hy	**0.025**	0.9346095281	1.39755E-04	1.14884E-04	**5.62**	0.0
k=0.001*hx=0.01*hy	**0.0125**	0.9347244117	2.48714E-05	9.34724E-01	**0.00**	
k=0.001*hx=0.01*hy	**0.00625**	0.0000000000	9.34749E-01			
	EXACT SOLUTION	**0.9347492831**				
x =	**0.50**					
y =	**0.50**					
t =	**0.25**					
k=0.001*hx=0.01*hy	**0.1**	0.9119633595	3.23167E-03	2.46252E-03	**4.20**	4.1
k=0.001*hx=0.01*hy	**0.05**	0.9144258823	7.69145E-04	5.98337E-04	**4.50**	4.3
k=0.001*hx=0.01*hy	**0.025**	0.9150242195	1.70808E-04	1.38973E-04	**5.37**	0.0
k=0.001*hx=0.01*hy	**0.0125**	0.9151631929	3.18346E-05	9.15163E-01		
k=0.001*hx=0.01*hy	**0.00625**	0.0000000000	9.15195E-01	9.15195E-01		
	EXACT SOLUTION	**0.9151950275**				
x =	**0.70**					
y =	**0.80**					
t =	**0.25**					
k=0.001*hx=0.01*hy	**0.1**	0.9577475607	1.92528E-03	1.46998E-03	**4.23**	4.1
k=0.001*hx=0.01*hy	**0.05**	0.9592175378	4.55301E-04	3.57727E-04	**4.67**	4.4
k=0.001*hx=0.01*hy	**0.025**	0.9595752647	9.75745E-05	8.15031E-05	**6.07**	0.0
k=0.001*hx=0.01*hy	**0.0125**	0.9596567677	1.60714E-05	9.59657E-01	**0.00**	
k=0.001*hx=0.01*hy	**0.00625**	0.0000000000	9.59673E-01			
	EXACT SOLUTION	**0.9596728392**				
x =	**0.40**					
y =	**0.60**					
t =	**0.50**					
k=0.001*hx=0.01*hy	**0.1**	0.9911336094	2.36127E-03	1.81400E-03	**4.31**	4.2
k=0.001*hx=0.01*hy	**0.05**	0.9929476081	5.47272E-04	4.32255E-04	**4.76**	4.4
k=0.001*hx=0.01*hy	**0.025**	0.9933798636	1.15017E-04	9.71800E-05	**6.45**	0.0
k=0.001*hx=0.01*hy	**0.0125**	0.9934770435	1.78369E-05	9.93477E-01	**0.00**	
k=0.001*hx=0.01*hy	**0.00625**	0.0000000000	9.93495E-01			
	EXACT SOLUTION	**0.9934948804**				
x =	**0.10**					
y =	**0.40**					
t =	**0.50**					
k=0.001*hx=0.01*hy	**0.1**	0.9971208136	7.65545E-04	5.88067E-04	**4.31**	4.2
k=0.001*hx=0.01*hy	**0.05**	0.9977088803	1.77478E-04	1.40176E-04	**4.76**	4.4
k=0.001*hx=0.01*hy	**0.025**	0.9978490560	3.73025E-05	3.15171E-05	**6.45**	0.0
k=0.001*hx=0.01*hy	**0.0125**	0.9978805731	5.78538E-06	9.97881E-01	**0.00**	
k=0.001*hx=0.01*hy	**0.00625**	0.0000000000	9.97886E-01			
	EXACT SOLUTION	**0.9978863585**				
x =	**0.90**					
y =	**0.90**					
t =	**0.50**					
k=0.001*hx=0.01*hy	**0.1**	0.9985802503	7.32986E-04	5.63404E-04	**4.32**	4.2
k=0.001*hx=0.01*hy	**0.05**	0.9991436543	1.69582E-04	1.34211E-04	**4.79**	4.5
k=0.001*hx=0.01*hy	**0.025**	0.9992778652	3.53711E-05	3.00520E-05	**6.65**	0.0
k=0.001*hx=0.01*hy	**0.0125**	0.9993079172	5.31909E-06	9.99308E-01	**0.00**	
k=0.001*hx=0.01*hy	**0.00625**	0.0000000000	9.99313E-01			
	EXACT SOLUTION	**0.9993132363**				
x =	**0.80**					
y =	**0.70**					
t =	**0.50**					
k=0.001*hx=0.01*hy	**0.1**	0.9951373375	1.44272E-03	1.10844E-03	**4.32**	4.2
k=0.001*hx=0.01*hy	**0.05**	0.9962457768	3.34279E-04	2.64143E-04	**4.77**	4.5
k=0.001*hx=0.01*hy	**0.025**	0.9965099194	7.01369E-05	5.93331E-05	**6.49**	0.0
k=0.001*hx=0.01*hy	**0.0125**	0.9965692524	1.08038E-05	9.96569E-01	**0.00**	
k=0.001*hx=0.01*hy	**0.00625**	0.0000000000	9.96580E-01			
	EXACT SOLUTION	**0.9965800562**				

Figure A- 12. Snapshots of profiles along the principal axes, for the nonlinear problem in Cartesian system, *Test Problem #17* (Table 7). (a) Snapshot profile parallel to the x-axis, at y = 0.60, t = 0.25. (b) Snapshot profile parallel to the y-axis, at x = 0.30, t = 0.50. Data from *Dsnap* output file.

(a)



(b)

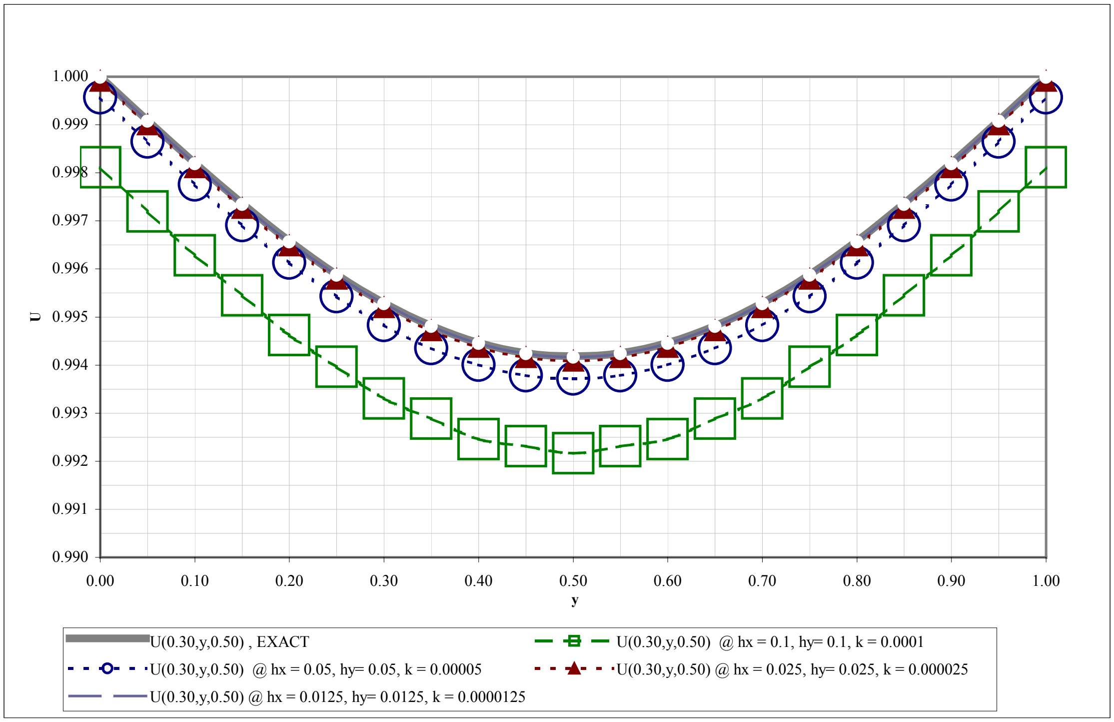


Figure A- 13. Evolution of grid functions with time, for the nonlinear problem in Cartesian system, *Test Problem #17* (Table 7): x = 0.5, y = 0.5. Data from *Devol* output file.

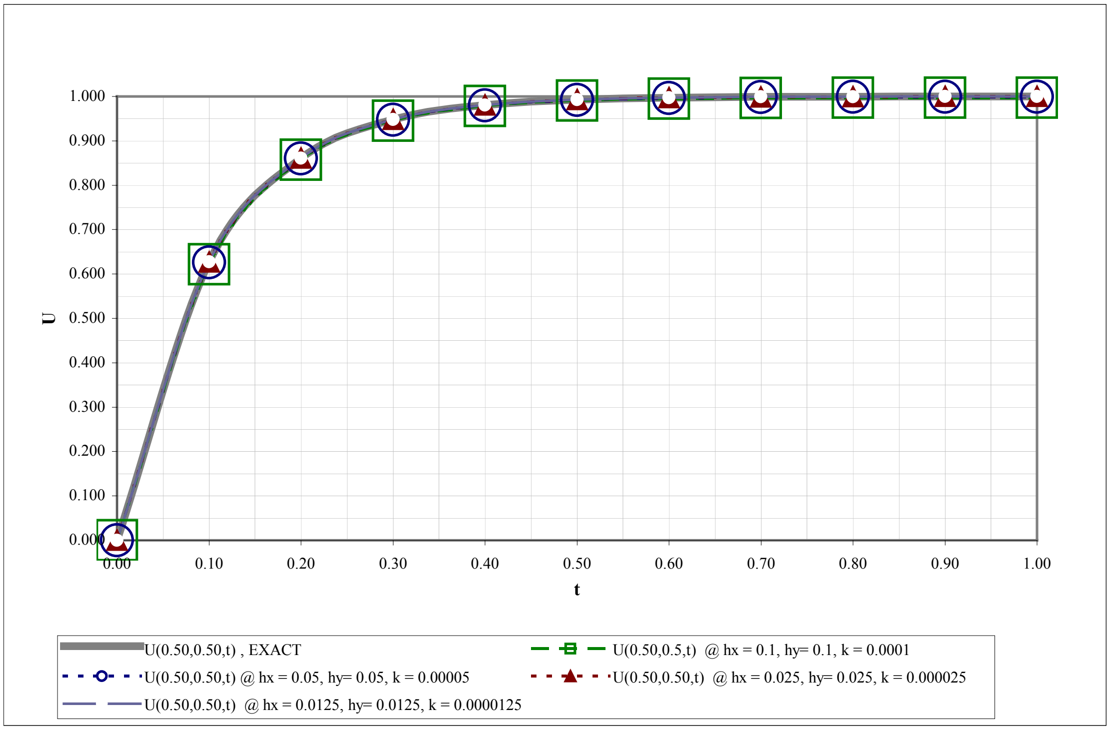
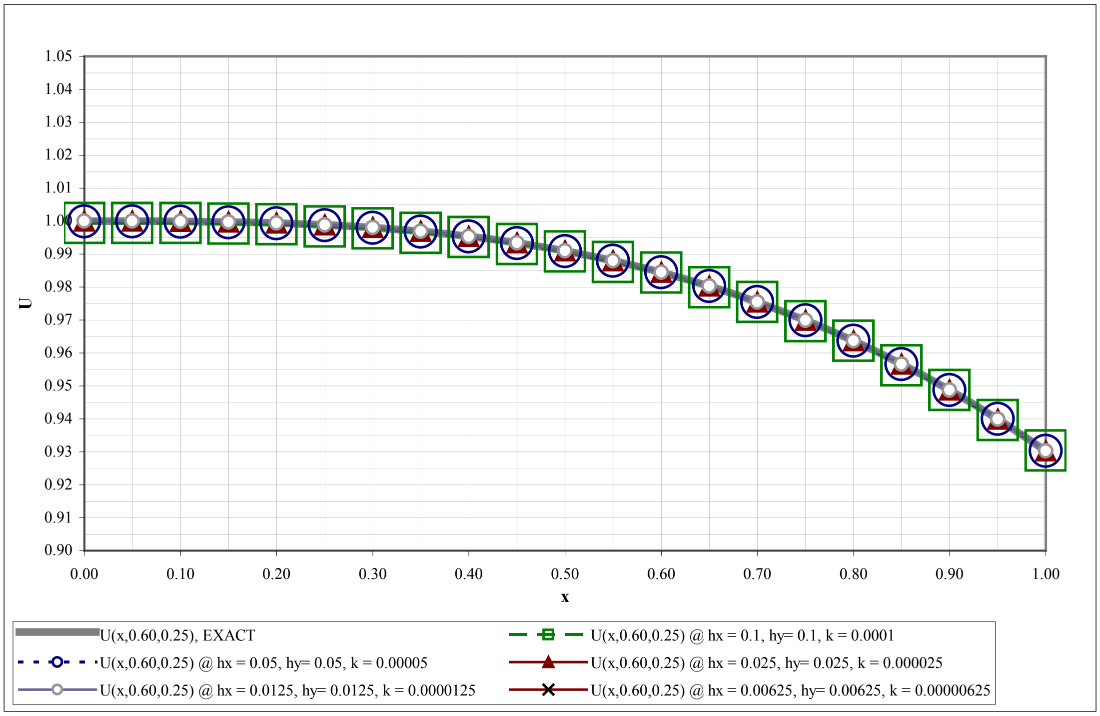


Table A- 9. Grid function convergence tests for the nonlinear problem in Cylindrical system, *Test Problem #23* (Table 7), generated from the output of the corresponding *Dconv* files.

Newton-Kantorovich with Douglass-Gunn Time Splitting: Grid Convergence Tests for T27 NonlinSphNeuRob: *U(x,y,t) = 1 - (e^-t). {x - SIN(x)}. SIN(y)*						
Grid Resolution Relationships	**Grid function Resolutions & Coordinates**	**U(x,y)**	**Absolute Grid Function Errors (W.R.T Exact Solution)**	**Absolute "Cauchy" Grid Function Errors (W.R.T Next Lower H**	**Theoretical (Based on Absolute Errors) $R = E_h/E_{h/2}$**	**Computationally Observed (Based on "Cauchy" errors): $R' = e_h/e_{h/2}$**
			E= ABS{U_{exact}(i,j) - u(i,j)}	**e = ABS{$u^{(m+1)}$(i,j) -**	**$R = 2^{N\_theo}$**	**$R' = 2^{N\_comp}$**
x =	**0.20**					
y =	**0.10**					
t =	**0.25**					
k=0.001*hx=0.01*hy	**0.1**	0.9995343341	3.62206E-04	2.72068E-04	**4.02**	4.0
k=0.001*hx=0.01*hy	**0.05**	0.9998064025	9.01375E-05	6.80253E-05	**4.08**	4.1
k=0.001*hx=0.01*hy	**0.025**	0.9998744278	2.21122E-05	1.67471E-05	**4.12**	4.1
k=0.001*hx=0.01*hy	**0.0125**	0.9998911749	5.36509E-06	4.11608E-06	**4.30**	
k=0.001*hx=0.01*hy	**0.00625**	0.9998952910	1.24901E-06			
	EXACT SOLUTION	**0.9998965400**				
x =	**0.60**					
y =	**0.30**					
t =	**0.25**					
k=0.001*hx=0.01*hy	**0.1**	0.9917860339	7.63830E-05	5.81611E-05	**4.19**	4.1
k=0.001*hx=0.01*hy	**0.05**	0.9918441950	1.82218E-05	1.41088E-05	**4.43**	4.3
k=0.001*hx=0.01*hy	**0.025**	0.9918583039	4.11302E-06	3.29129E-06	**5.01**	0.0
k=0.001*hx=0.01*hy	**0.0125**	0.9918615951	8.21730E-07	4.56014E-04	**0.00**	
k=0.001*hx=0.01*hy	**0.00625**	0.9923176090	4.55192E-04			
	EXACT SOLUTION	**0.9918624169**				
x =	**0.50**					
y =	**0.50**					
t =	**0.25**					
k=0.001*hx=0.01*hy	**0.1**	0.9921702579	1.47712E-04	1.11731E-04	**4.11**	4.1
k=0.001*hx=0.01*hy	**0.05**	0.9922819885	3.59811E-05	2.74741E-05	**4.23**	4.2
k=0.001*hx=0.01*hy	**0.025**	0.9923094627	8.50698E-06	6.60197E-06	**4.47**	4.3
k=0.001*hx=0.01*hy	**0.0125**	0.9923160646	1.90501E-06	1.54437E-06		
k=0.001*hx=0.01*hy	**0.00625**	0.9923176090	3.60638E-07	3.60638E-07		
	EXACT SOLUTION	**0.9923179696**				
x =	**0.70**					
y =	**0.80**					
t =	**0.25**					
k=0.001*hx=0.01*hy	**0.1**	0.9688150881	2.05897E-05	1.69568E-05	**5.67**	4.9
k=0.001*hx=0.01*hy	**0.05**	0.9688320448	3.63297E-06	3.49279E-06	**25.92**	7.4
k=0.001*hx=0.01*hy	**0.025**	0.9688355376	1.40180E-07	4.73090E-07	**0.42**	7.5
k=0.001*hx=0.01*hy	**0.0125**	0.9688360107	3.32910E-07	6.26900E-08	**1.23**	
k=0.001*hx=0.01*hy	**0.00625**	0.9688359480	2.70220E-07			
	EXACT SOLUTION	**0.9688356778**				
x =	**0.40**					
y =	**0.60**					
t =	**0.50**					
k=0.001*hx=0.01*hy	**0.1**	0.9961467876	2.29281E-04	1.73527E-04	**4.11**	4.0
k=0.001*hx=0.01*hy	**0.05**	0.9963203150	5.57533E-05	4.30297E-05	**4.38**	4.3
k=0.001*hx=0.01*hy	**0.025**	0.9963633447	1.27235E-05	1.00516E-05	**4.76**	4.4
k=0.001*hx=0.01*hy	**0.0125**	0.9963733963	2.67190E-06	2.26866E-06	**6.63**	
k=0.001*hx=0.01*hy	**0.00625**	0.9963756650	4.03243E-07			
	EXACT SOLUTION	**0.9963760682**				
x =	**0.10**					
y =	**0.40**					
t =	**0.50**					
k=0.001*hx=0.01*hy	**0.1**	0.9995462942	4.14360E-04	3.11325E-04	**4.02**	4.0
k=0.001*hx=0.01*hy	**0.05**	0.9998576191	1.03035E-04	7.85002E-05	**4.20**	4.2
k=0.001*hx=0.01*hy	**0.025**	0.9999361192	2.45348E-05	1.88692E-05	**4.33**	4.2
k=0.001*hx=0.01*hy	**0.0125**	0.9999549884	5.66558E-06	4.50060E-06	**4.86**	
k=0.001*hx=0.01*hy	**0.00625**	0.9999594890	1.16498E-06			
	EXACT SOLUTION	**0.9999606540**				
x =	**0.90**					
y =	**0.90**					
t =	**0.50**					
k=0.001*hx=0.01*hy	**0.1**	0.9446106557	4.34162E-05	2.98951E-05	**3.21**	3.5
k=0.001*hx=0.01*hy	**0.05**	0.9445807606	1.35211E-05	8.44768E-06	**2.67**	2.8
k=0.001*hx=0.01*hy	**0.025**	0.9445723129	5.07340E-06	3.02779E-06	**2.48**	2.7
k=0.001*hx=0.01*hy	**0.0125**	0.9445692851	2.04561E-06	1.13912E-06	**2.26**	
k=0.001*hx=0.01*hy	**0.00625**	0.9445681460	9.06493E-07			
	EXACT SOLUTION	**0.9445672395**				
x =	**0.80**					
y =	**0.70**					
t =	**0.50**					
k=0.001*hx=0.01*hy	**0.1**	0.9676890410	1.88615E-05	1.64452E-05	**7.81**	5.1
k=0.001*hx=0.01*hy	**0.05**	0.9677054862	2.41630E-06	3.25299E-06	**2.89**	114.8
k=0.001*hx=0.01*hy	**0.025**	0.9677087392	8.36686E-07	2.83300E-08	**0.97**	0.1
k=0.001*hx=0.01*hy	**0.0125**	0.9677087675	8.65016E-07	3.14540E-07	**1.57**	
k=0.001*hx=0.01*hy	**0.00625**	0.9677084530	5.50476E-07			
	EXACT SOLUTION	**0.9677079025**				

Figure A- 14. Snapshots of profiles along the principal axes, for the nonlinear problem in Cylindrical system, *Test Problem #23* (Table 7). (a) Snapshot profile parallel to the x-axis, at y = 0.60, t = 0.25. (b) Snapshot profile parallel to the y-axis, at x = 0.30, t = 0.50. Data from *Dsnap* output file.

(a)



(b)

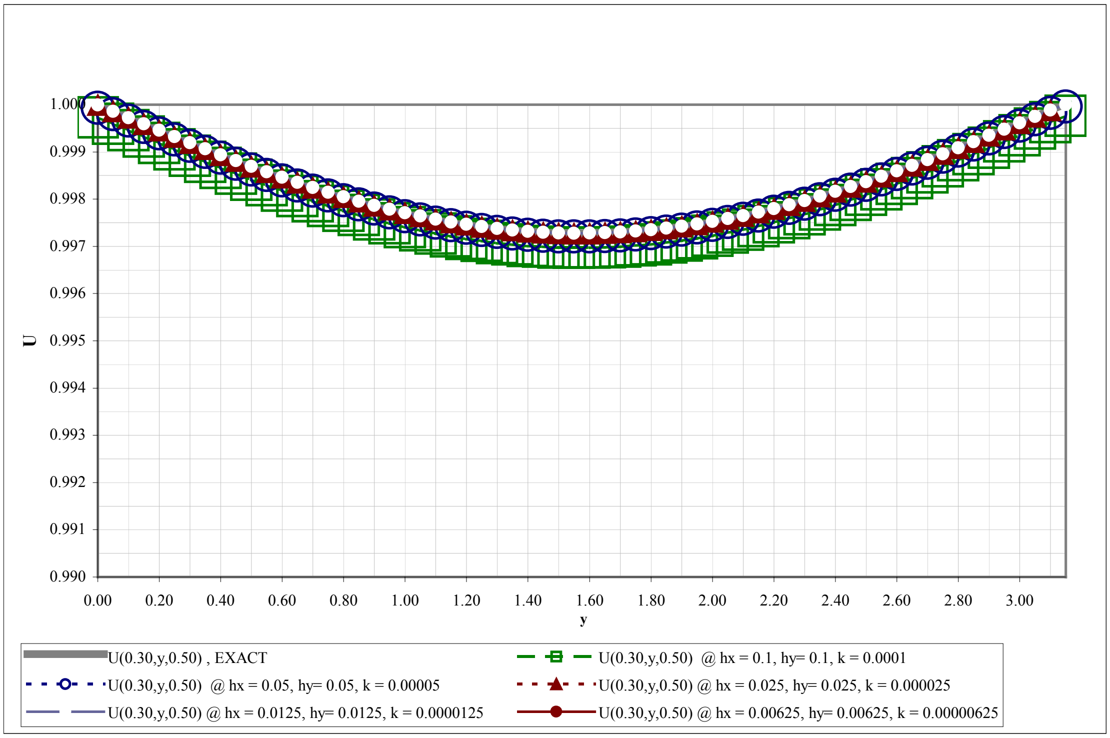


Figure A- 15. Evolution of grid functions with time, for the nonlinear problem in Cylindrical system, *Test Problem #23* (Table 7): x = 0.5, y = 0.5. Data from *Devol* output file.

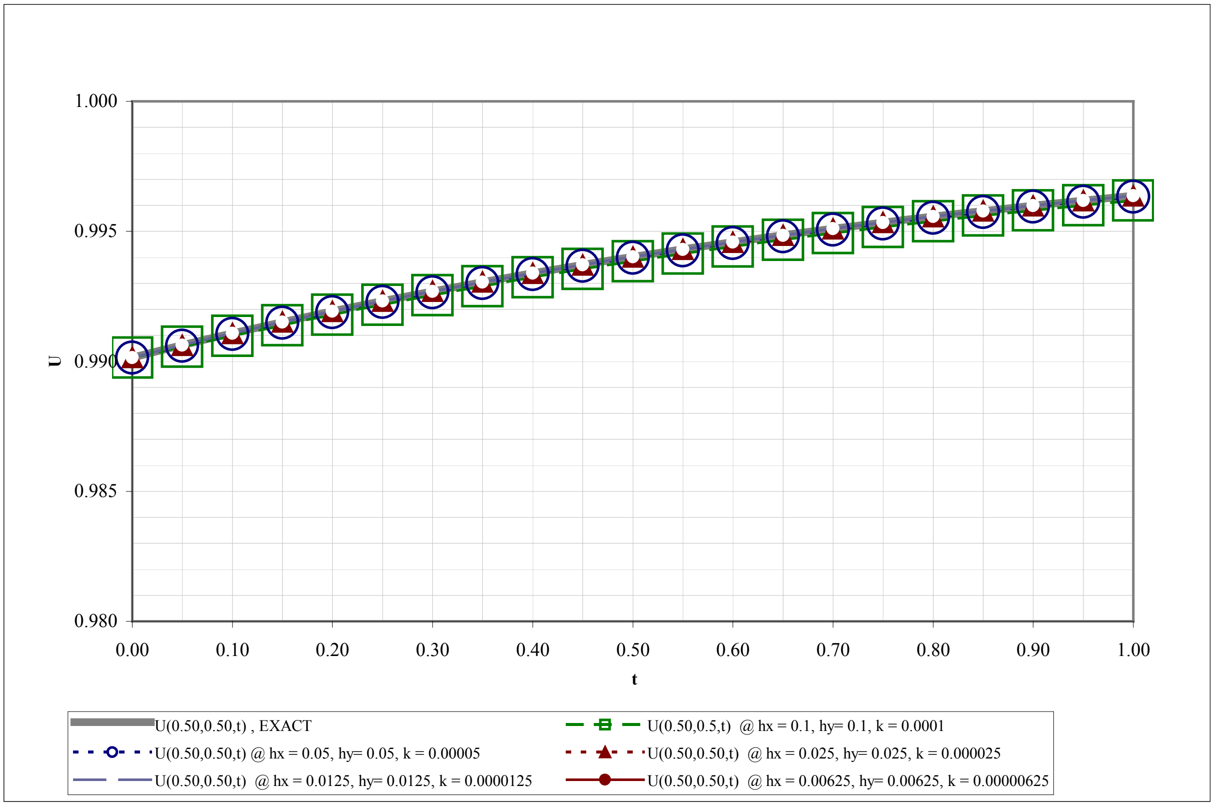
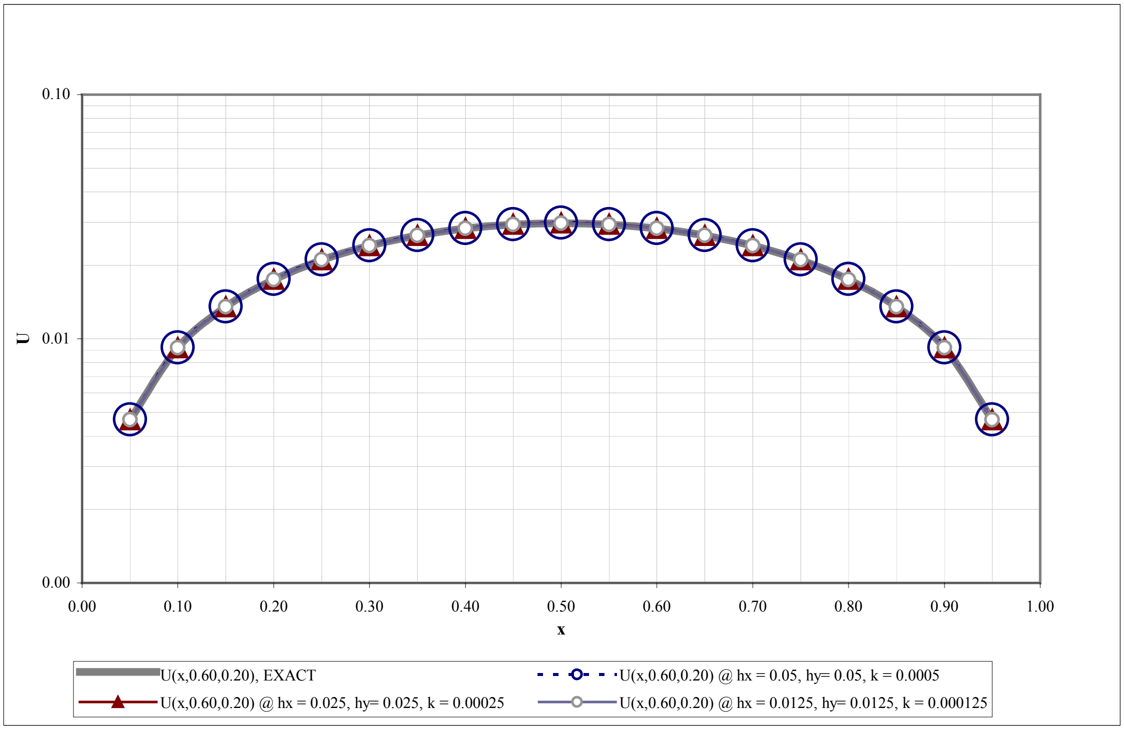


Table A- 10. Grid function convergence tests for the linear problem in Cartesian system, *Test Problem #28* (Table 7), generated from the output of the corresponding *Dconv* files.

Newton-Kantorovich with Douglass-Gunn Time Splitting: Grid Convergence Tests for T27 NonlinSphNeuRob: *U(x,y,t) from Carslaw & Jaegar 1959*						
Grid Resolution Relationships	Grid function Resolutions & Coordinates	U(x,y)	Absolute Grid Function Errors	Absolute "Cauchy" Grid Function Errors (W.R.T Next Lower H Value)	Theoretical (Based on Absolute Errors) $R = E_h/E_{h/2}$	Computationally Observed (Based on "Cauchy" errors): $R' = e_h/e_{h/2}$
			E= ABS{U_{exact}(i,j) - u(i,j)}	e = ABS{$u^{(m+1)}$(i,j) - $u^{(m)}$(i,j)}	$R = 2^{N\_theo}$	$R' = 2^{N\_comp}$
x =	0.20					
y =	0.10					
t =	0.20					
k=0.01*hx=0.01*hy	0.1	0.0000000000				
k=0.01*hx=0.01*hy	0.05	0.0057046101	2.26853E-05	1.70091E-05	4.00	4.0
k=0.01*hx=0.01*hy	0.025	0.0056876010	5.67623E-06	4.25688E-06	4.00	
k=0.01*hx=0.01*hy	0.0125	0.0056833441	1.41936E-06			
k=0.01*hx=0.01*hy	0.00625	0.0000000000				
	EXACT SOLUTION	0.0056819248				
x =	0.60					
y =	0.30					
t =	0.20					
k=0.01*hx=0.01*hy	0.1	0.0000000000	2.40690E-02	2.41651E-02		
k=0.01*hx=0.01*hy	0.05	0.0241651094	9.60948E-05	7.20503E-05	4.00	4.0
k=0.01*hx=0.01*hy	0.025	0.0240930592	2.40446E-05	1.80321E-05	4.00	
k=0.01*hx=0.01*hy	0.0125	0.0240750270	6.01244E-06	2.40750E-02		
k=0.01*hx=0.01*hy	0.00625	0.0000000000	2.40690E-02			
	EXACT SOLUTION	0.0240690146				
x =	0.50					
y =	0.50					
t =	0.20					
k=0.01*hx=0.01*hy	0.1	0.0000000000	3.12820E-02	3.14069E-02		
k=0.01*hx=0.01*hy	0.05	0.0314068766	1.24891E-04	9.36415E-05	4.00	4.0
k=0.01*hx=0.01*hy	0.025	0.0313132351	3.12500E-05	2.34358E-05	4.00	
k=0.01*hx=0.01*hy	0.0125	0.0312897993	7.81419E-06	3.12898E-02		
k=0.01*hx=0.01*hy	0.00625	0.0000000000	3.12820E-02	3.12820E-02		
	EXACT SOLUTION	0.0312819851				
x =	0.70					
y =	0.80					
t =	0.20					
k=0.01*hx=0.01*hy	0.1	0.0000000000	1.48755E-02	1.49349E-02		
k=0.01*hx=0.01*hy	0.05	0.0149348610	5.93904E-05	4.45299E-05	4.00	4.0
k=0.01*hx=0.01*hy	0.025	0.0148903311	1.48605E-05	1.11446E-05	4.00	
k=0.01*hx=0.01*hy	0.0125	0.0148791866	3.71596E-06	1.48792E-02		
k=0.01*hx=0.01*hy	0.00625	0.0000000000	1.48755E-02			
	EXACT SOLUTION	0.0148754706				
x =	0.40					
y =	0.60					
t =	0.40					
k=0.01*hx=0.01*hy	0.1	0.0000000000	5.45985E-04	5.52625E-04		
k=0.01*hx=0.01*hy	0.05	0.0005526250	6.63954E-06	4.98520E-06	4.01	4.0
k=0.01*hx=0.01*hy	0.025	0.0005476398	1.65434E-06	1.24110E-06	4.00	
k=0.01*hx=0.01*hy	0.0125	0.0005463987	4.13239E-07	5.46399E-04		
k=0.01*hx=0.01*hy	0.00625	0.0000000000	5.45985E-04			
	EXACT SOLUTION	0.0005459855				
x =	0.10					
y =	0.40					
t =	0.40					
k=0.01*hx=0.01*hy	0.1	0.0000000000	1.77401E-04	1.79559E-04		
k=0.01*hx=0.01*hy	0.05	0.0001795588	2.15732E-06	1.61979E-06	4.01	4.0
k=0.01*hx=0.01*hy	0.025	0.0001779390	5.37527E-07	4.03258E-07	4.00	
k=0.01*hx=0.01*hy	0.0125	0.0001775357	1.34269E-07	1.77536E-04		
k=0.01*hx=0.01*hy	0.00625	0.0000000000	1.77401E-04			
	EXACT SOLUTION	0.0001774014				
x =	0.90					
y =	0.90					
t =	0.40					
k=0.01*hx=0.01*hy	0.1	0.0000000000	5.76412E-05	5.83422E-05		
k=0.01*hx=0.01*hy	0.05	0.0000583422	7.00955E-07	5.26302E-07	4.01	4.0
k=0.01*hx=0.01*hy	0.025	0.0000578159	1.74653E-07	1.31026E-07	4.00	0.0
k=0.01*hx=0.01*hy	0.0125	0.0000576848	4.36267E-08	5.76848E-05		
k=0.01*hx=0.01*hy	0.00625	0.0000000000	5.76412E-05			
	EXACT SOLUTION	0.0000576412				
x =	0.80					
y =	0.70					
t =	0.40					
k=0.01*hx=0.01*hy	0.1	0.0000000000	2.87042E-04	2.90532E-04		
k=0.01*hx=0.01*hy	0.05	0.0002905322	3.49061E-06	2.62088E-06	4.01	4.0
k=0.01*hx=0.01*hy	0.025	0.0002879113	8.69737E-07	6.52485E-07	4.00	0.0
k=0.01*hx=0.01*hy	0.0125	0.0002872588	2.17252E-07	2.87259E-04		
k=0.01*hx=0.01*hy	0.00625	0.0000000000	2.87042E-04			
	EXACT SOLUTION	0.0002870416				

Figure A- 16. Snapshots of profiles along the principal axes, for the linear problem in Cartesian system, *Test Problem #28* (Table 7). (a) Snapshot profile parallel to the x-axis, at y = 0.60, t = 0.20. (b) Snapshot profile parallel to the y-axis, at x = 0.30, t = 0.40. Data from *Dsnap* output file.

(a)



(b)

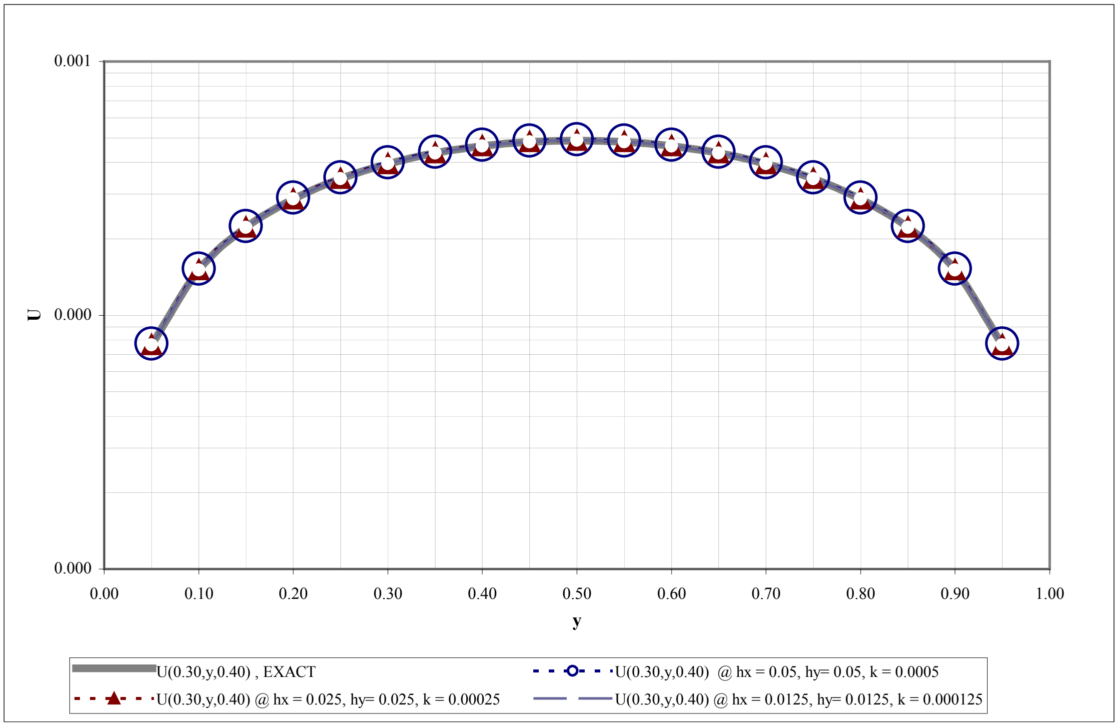


Figure A- 17. Evolution of grid functions with time, for the linear problem in Cartesian system, *Test Problem #28* (Table 7): x = 0.5, y = 0.5. Data from *Devol* output file.

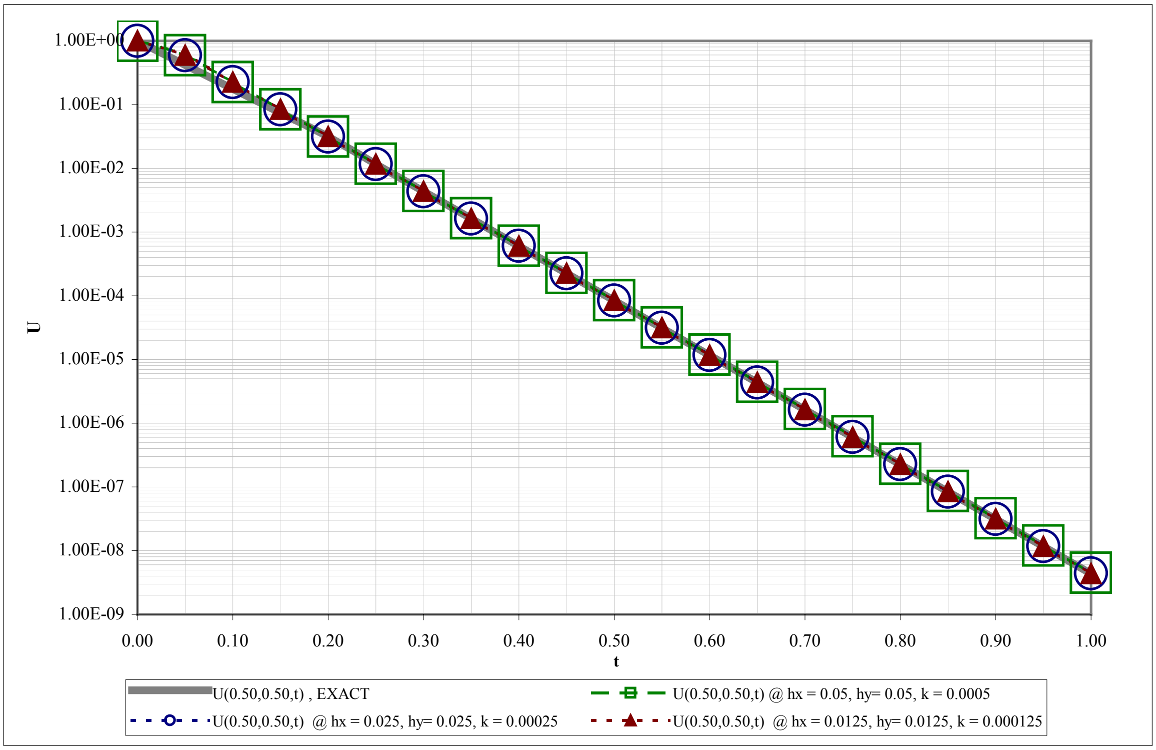


Figure A- 18. Surface contour plots comparing the analytical (exact) and numerical solutions at specific times, for the linear problem in Cartesian system, *Test Problem #28* (Table 7). As can be seen, at the resolution of these plots, the analytical and numerical solutions are identical for times 0.0, 0.4, and 0.8.

ANALYTICAL **NUMERICAL**

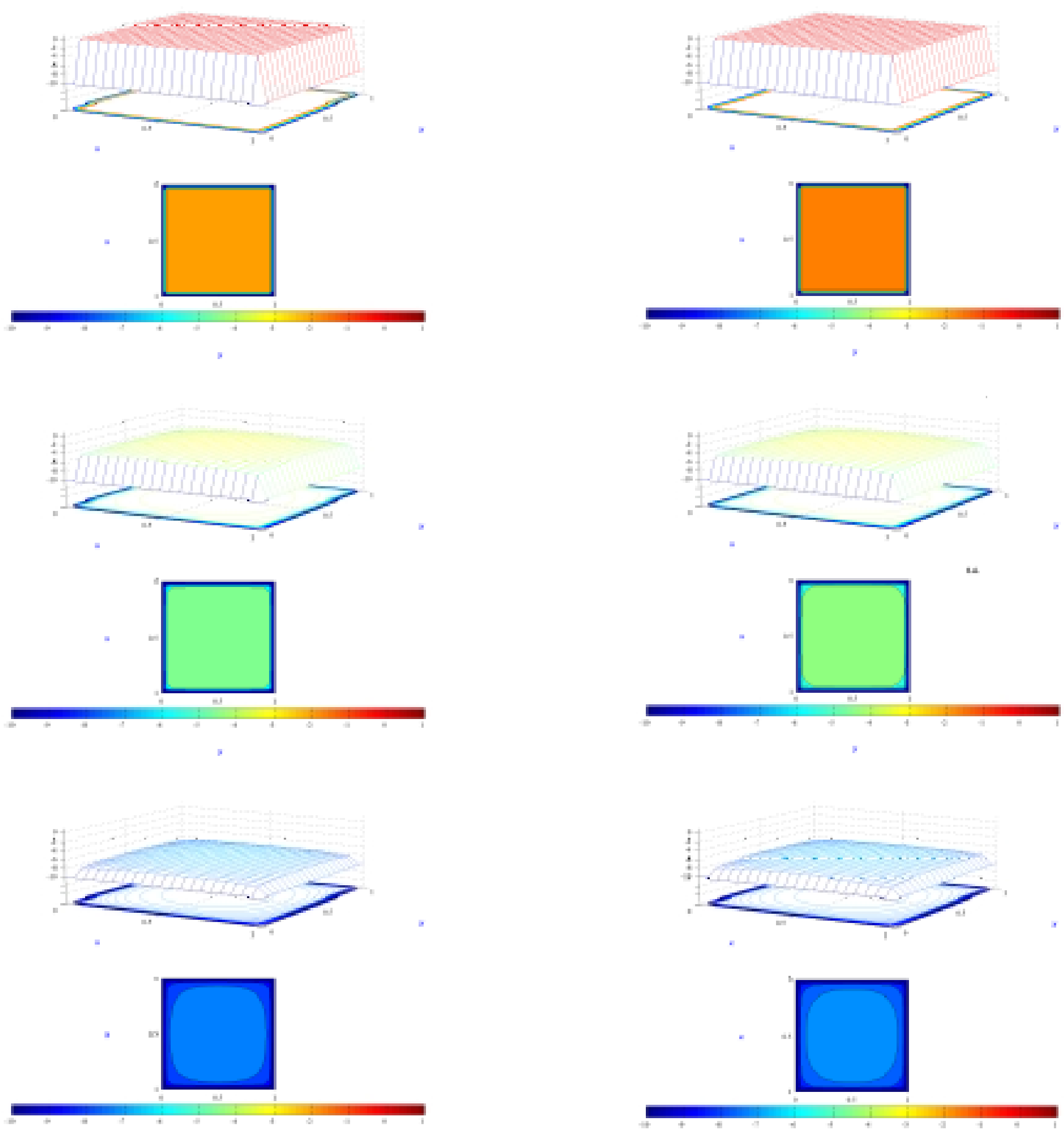
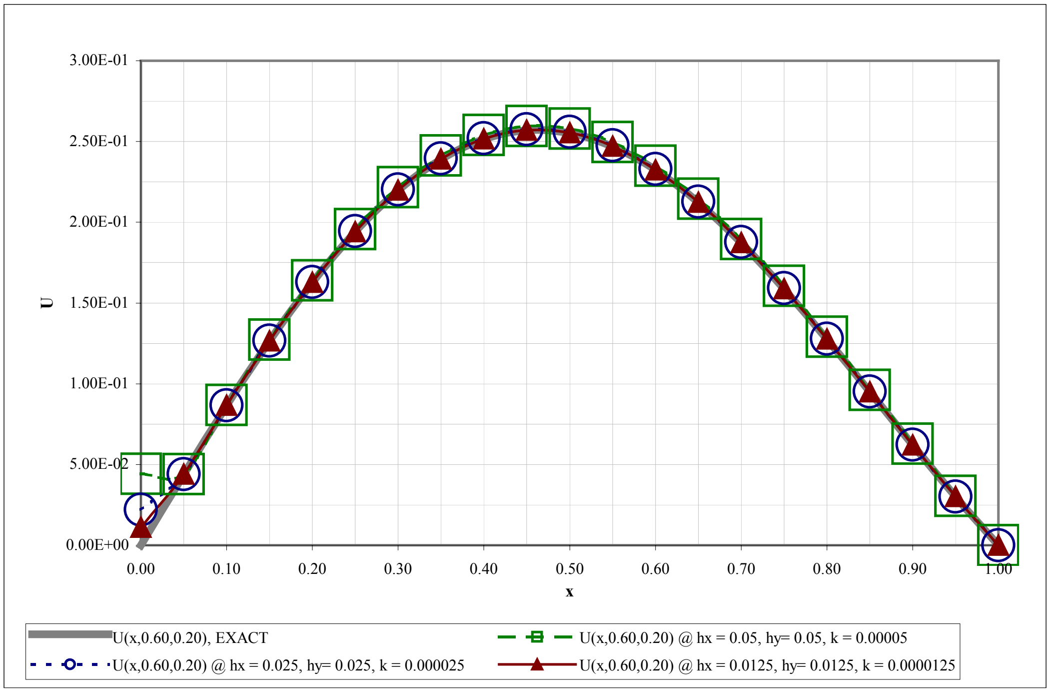


Table A- 11. Grid function convergence tests for the linear problem in Spherical system, *Test Problem #29* (Table 7), generated from the output of the corresponding *Dconv* files.

Newton-Kantorovich with Douglass-Gunn Time Splitting: Grid Convergence Tests for T27_NonlinSphNeuRob: *U(x,y,t) from Carslaw & Jaegar 1959*						
Grid Resolution Relationships	**Grid function Resolutions & Coordinates**	**U(x,y)**	**Absolute Grid Function Errors**	**Absolute "Cauchy" Grid Function Errors (W.R.T Next Lower H Value)**	**Theoretical (Based on Absolute Errors) $R = E_h/E_{h/2}$**	**Computationally Observed (Based on "Cauchy" errors): $R' = e_h/e_{h/2}$**
			E= ABS{U_{exact}(i,j) - u(i,j)}	**e = ABS{$u^{(m+1)}$(i,j) - $u^{(m)}$(i,j)}**	**$R = 2^{N\_theo}$**	**$R' = 2^{N\_comp}$**
x =	**0.20**					
y =	**0.10**					
t =	**0.20**					
k=0.01*hx=0.01*hy	**0.1**	0.0000000000				
k=0.01*hx=0.01*hy	**0.05**	0.1976762349	1.55057E-03	1.16117E-03	**3.98**	4.3
k=0.01*hx=0.01*hy	**0.025**	0.1965150662	3.89406E-04	2.67851E-04	**3.20**	
k=0.01*hx=0.01*hy	**0.0125**	0.1962472151	1.21555E-04			
	EXACT SOLUTION	**0.1961256600**				
x =	**0.60**					
y =	**0.30**					
t =	**0.20**					
k=0.01*hx=0.01*hy	**0.1**	0.0000000000				
k=0.01*hx=0.01*hy	**0.05**	0.2719819163	3.08096E-03	2.31175E-03	**4.01**	4.1
k=0.01*hx=0.01*hy	**0.025**	0.2696701658	7.69214E-04	5.69816E-04	**3.86**	
k=0.01*hx=0.01*hy	**0.0125**	0.2691003495	1.99398E-04			
	EXACT SOLUTION	**0.2689009520**				
x =	**0.50**					
y =	**0.50**					
t =	**0.20**					
k=0.01*hx=0.01*hy	**0.1**	0.0000000000				
k=0.01*hx=0.01*hy	**0.05**	0.2744022733	2.96548E-03	2.22464E-03	**4.00**	4.1
k=0.01*hx=0.01*hy	**0.025**	0.2721776344	7.40838E-04	5.44935E-04	**3.78**	
k=0.01*hx=0.01*hy	**0.0125**	0.2716326990	1.95903E-04			
	EXACT SOLUTION	**0.2714367960**				
x =	**0.70**					
y =	**0.80**					
t =	**0.20**					
k=0.01*hx=0.01*hy	**0.1**	0.0000000000				
k=0.01*hx=0.01*hy	**0.05**	0.1599398572	1.86779E-03	1.40168E-03	**4.01**	4.1
k=0.01*hx=0.01*hy	**0.025**	0.1585381806	4.66115E-04	3.43987E-04	**3.82**	
k=0.01*hx=0.01*hy	**0.0125**	0.1581941935	1.22127E-04			
	EXACT SOLUTION	**0.1580720660**				
x =	**0.40**					
y =	**0.60**					
t =	**0.40**					
k=0.01*hx=0.01*hy	**0.1**	0.0000000000				
k=0.01*hx=0.01*hy	**0.05**	0.0045194926	8.48274E-05	6.37044E-05	**4.02**	5.3
k=0.01*hx=0.01*hy	**0.025**	0.0044557883	2.11230E-05	1.20638E-05	**2.33**	
k=0.01*hx=0.01*hy	**0.0125**	0.0044437245	9.05927E-06			
	EXACT SOLUTION	**0.0044346652**				
x =	**0.10**					
y =	**0.40**					
t =	**0.40**					
k=0.01*hx=0.01*hy	**0.1**	0.0000000000				
k=0.01*hx=0.01*hy	**0.05**	0.0017170299	1.43174E-05	1.06705E-05	**3.93**	4.1
k=0.01*hx=0.01*hy	**0.025**	0.0017063595	3.64694E-06	2.63327E-06	**0.58**	
k=0.01*hx=0.01*hy	**0.0125**	0.0017089927	6.28021E-06			
	EXACT SOLUTION	**0.0017027125**				
x =	**0.90**					
y =	**0.90**					
t =	**0.40**					
k=0.01*hx=0.01*hy	**0.1**	0.0000000000				
k=0.01*hx=0.01*hy	**0.05**	0.0008418041	1.78854E-05	1.34411E-05	**4.02**	4.8
k=0.01*hx=0.01*hy	**0.025**	0.0008283630	4.44429E-06	2.81037E-06	**2.72**	
k=0.01*hx=0.01*hy	**0.0125**	0.0008255526	1.63393E-06			
	EXACT SOLUTION	**0.0008239187**				
x =	**0.80**					
y =	**0.70**					
t =	**0.40**					
k=0.01*hx=0.01*hy	**0.1**	0.0000000000				
k=0.01*hx=0.01*hy	**0.05**	0.0021283932	4.54215E-05	3.41352E-05	**4.02**	4.6
k=0.01*hx=0.01*hy	**0.025**	0.0020942580	1.12863E-05	7.36746E-06	**2.88**	
k=0.01*hx=0.01*hy	**0.0125**	0.0020868906	3.91889E-06			
	EXACT SOLUTION	**0.0020829717**				

Figure A- 19. Snapshots of profiles along the principal axes, for the linear problem in Spherical system, *Test Problem #29* (Table 7). (a) Snapshot profile parallel to the x-axis, at y = 0.60, t = 0.20. (b) Snapshot profile parallel to the y-axis, at x = 0.30, t = 0.40. Data from *Dsnap* output file.

(a)



(b)

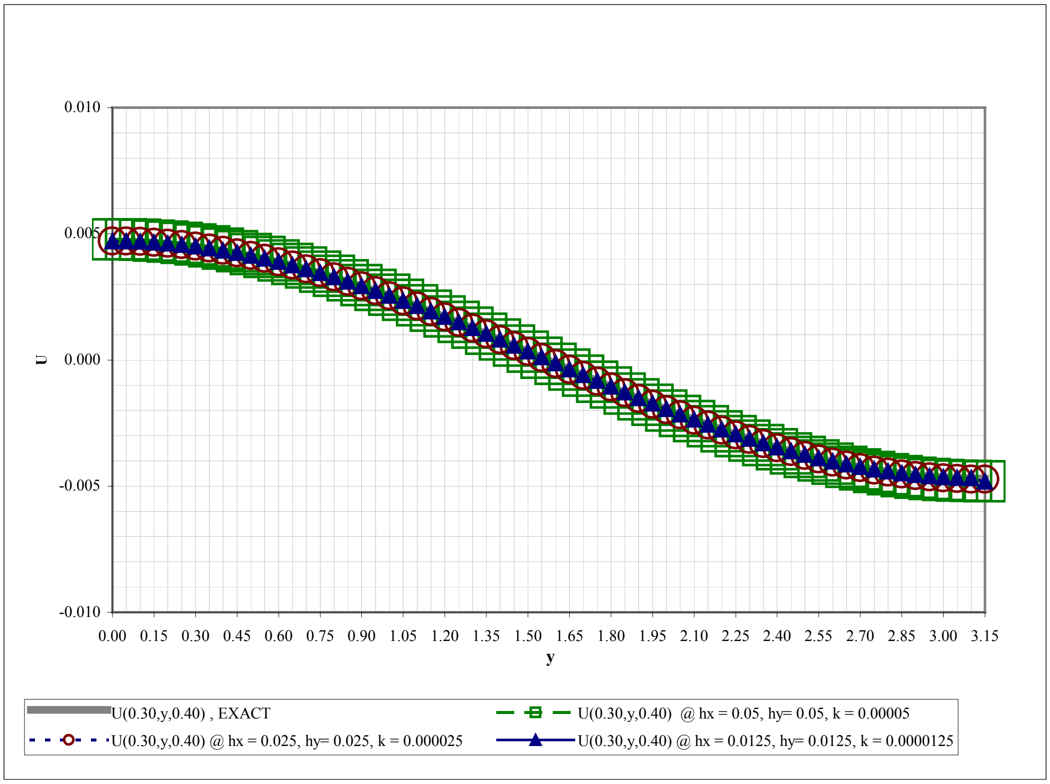


Figure A- 20. Evolution of grid functions with time, for the linear problem in Spherical system, *Test Problem #29* (Table 7): x = 0.5, y = 0.5. Data from *Devol* output file.

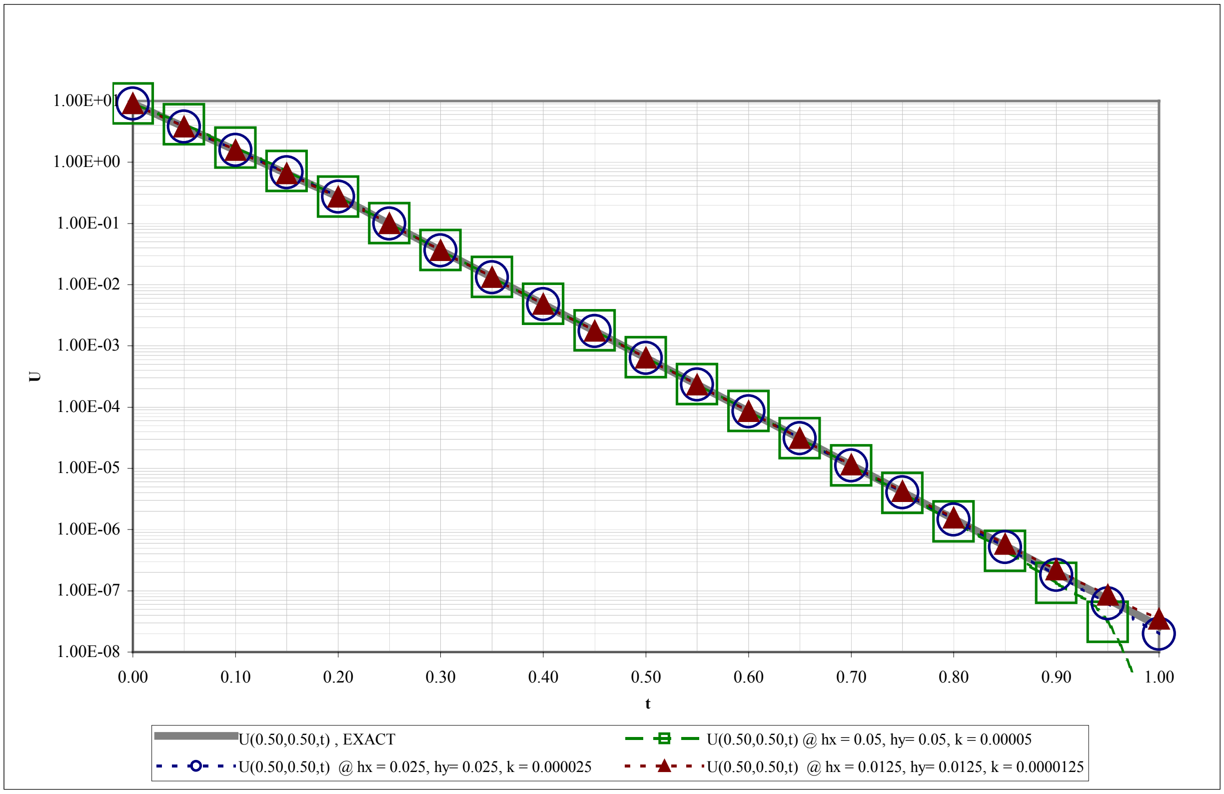


Figure A- 21. Surface contour plots comparing the analytical (exact) and numerical solutions at specific times, for the linear problem in Spherical system, *Test Problem #29* (Table 7). As can be seen, at the resolution of these plots, the analytical and numerical solutions are identical for times 0.0, 0.4, and 0.8.

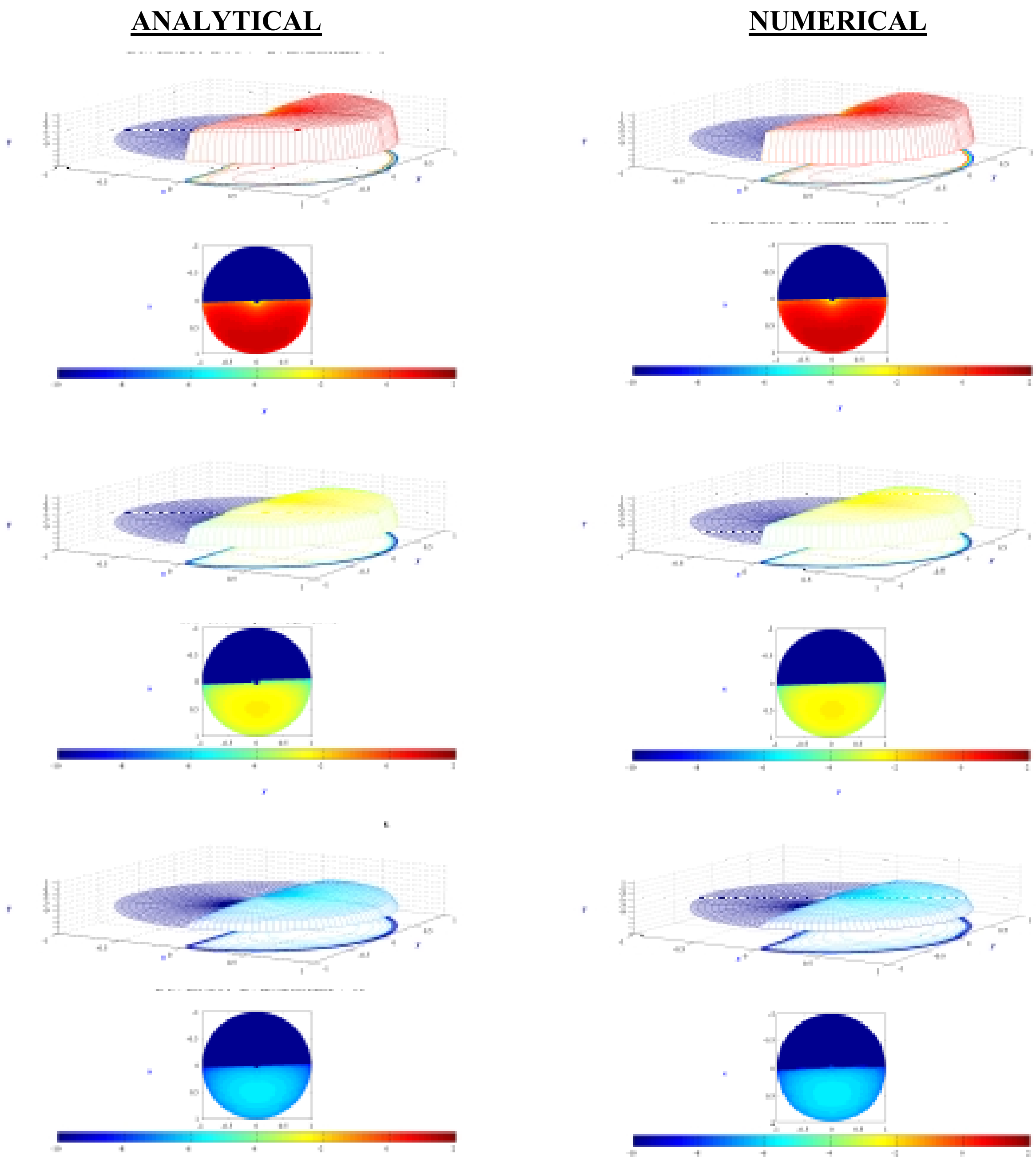
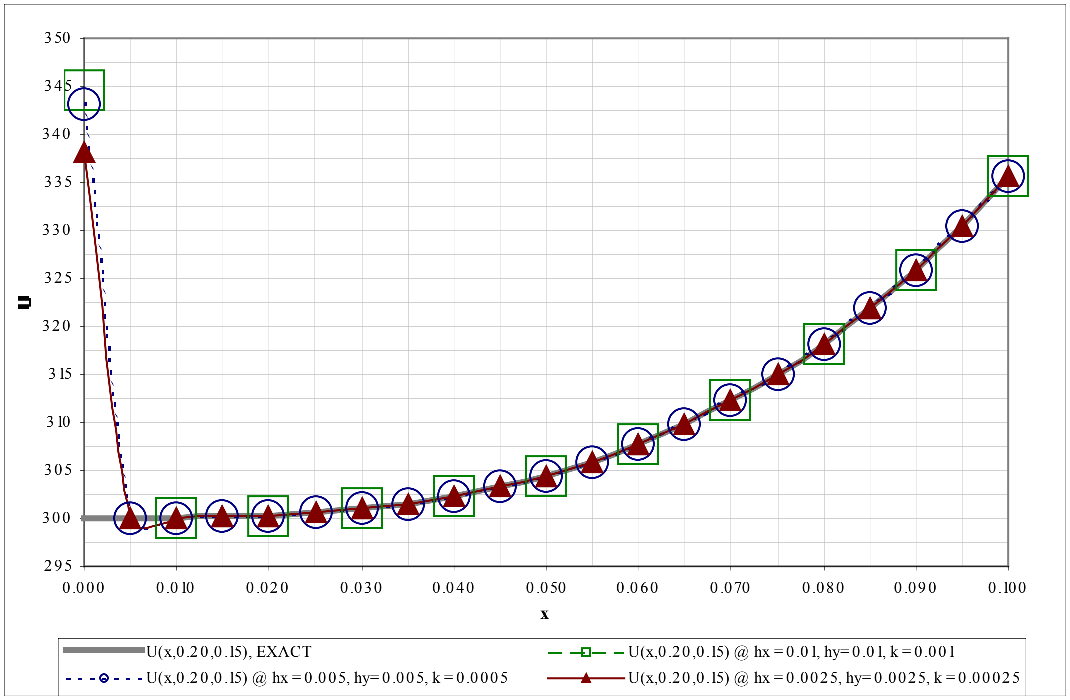


Table A- 12. Grid function convergence tests for the linear problem in Spherical system, *Test Problem #32* (Table 7), generated from the output of the corresponding *Dconv* files.

Newton-Kantorovich with Douglass-Gunn Time Splitting: Grid Convergence Tests for T27_NonlinSphNeuRob: *U(x,y,t) from Carslaw & Jaeger 1959*						
Grid Resolution Relationships	**Grid function Resolutions & Coordinates**	**U(x,y)**	**Absolute Grid Function Errors (W.R.T Exact Solution)**	**Absolute "Cauchy" Grid Function Errors (W.R.T Next Lower H Value)**	**Theoretical (Based on Absolute Errors) $R = E_h/E_{h/2}$**	**Computationally Observed (Based on "Cauchy" errors): $R' = e_h/e_{h/2}$**
			E= ABS{U_{exact}(i,j) - u(i,j)}	**e = ABS{$u^{(m+1)}$(i,j) - $u^{(m)}$(i,j)}**	**$R = 2^{N\_theo}$**	**$R' = 2^{N\_comp}$**
x =	**0.20**					
y =	**0.10**					
t =	**0.20**					
0.1	0.314159265	1.10102454E+0	3.14079E+03	3.50764E+03	**8.56**	133102.3
0.05	0.157079633	1.45178893E+0	3.66849E+02	2.63530E-02	**1.00**	
0.025	0.078539816	1.45178630E+0	3.66823E+02			
0.0125	0.039269908	0.00000000E+0				
0.00625	0.019634954	0.00000000E+0				
	EXACT SOLUTION	**1.41510403E+0**				
x =	**0.60**					
y =	**1.57**					
t =	**0.15**					
0.1	0.314159265	2.89753100E+0	1.57988E+02	1.08659E+00	**1.01**	2.0
0.05	0.157079633	2.89752013E+0	1.56901E+02	5.43010E-01	**1.00**	
0.025	0.078539816	2.89751470E+0	1.56358E+02			
0.0125	0.039269908	0.00000000E+0				
0.00625	0.019634954	0.00000000E+0				
	EXACT SOLUTION	**2.89595112E+0**				
x =	**0.65**					
y =	**2.83**					
t =	**0.15**					
0.1	0.314159265	3.78153294E+0	1.01015E+05	1.01052E+05	**2760.81**	112552.4
0.05	0.157079633	4.79205125E+0	3.65890E+01	8.97820E-01	**1.03**	
0.025	0.078539816	4.79204227E+0	3.56912E+01			
0.0125	0.039269908	0.00000000E+0				
0.00625	0.019634954	0.00000000E+0				
	EXACT SOLUTION	**4.79168536E+0**				
x =	**0.70**					
y =	**0.47**					
t =	**0.15**					
0.1	0.314159265	2.93380528E+0	3.43877E+04	3.47079E+04	**107.39**	291193.2
0.05	0.157079633	6.40459535E+0	3.20224E+02	1.19192E-01	**1.00**	
0.025	0.078539816	6.40458343E+0	3.20105E+02			
0.0125	0.039269908	0.00000000E+0				
0.00625	0.019634954	0.00000000E+0				
	EXACT SOLUTION	**6.37257293E+0**				
x =	**0.80**					
y =	**2.98**					
t =	**0.20**					
0.1	0.314159265	8.40414630E+0	3.53829E+03	3.62332E+03	**41.61**	1733.3
0.05	0.157079633	8.36791311E+0	8.50335E+01	2.09041E+00	**0.98**	
0.025	0.078539816	8.36789220E+0	8.71239E+01			
0.0125	0.039269908	0.00000000E+0				
0.00625	0.019634954	0.00000000E+0				
	EXACT SOLUTION	**8.36876344E+0**				
x =	**0.75**					
y =	**1.57**					
t =	**0.20**					
0.1	0.314159265	4.34689386E+0	9.76630E+04	9.79523E+04	**337.67**	73586.4
0.05	0.157079633	5.32641652E+0	2.89227E+02	1.33112E+00	**1.00**	
0.025	0.078539816	5.32640320E+0	2.87896E+02			
0.0125	0.039269908	0.00000000E+0				
0.00625	0.019634954	0.00000000E+0				
	EXACT SOLUTION	**5.32352425E+0**				
x =	**0.90**					
y =	**1.26**					
t =	**0.20**					
0.1	0.314159265	6.90277968E+0	2.54839E+03	3.45008E+00	**1.00**	2.0
0.05	0.157079633	6.90274518E+0	2.55184E+03	1.72499E+00	**1.00**	
0.025	0.078539816	6.90272793E+0	2.55357E+03			
0.0125	0.039269908	0.00000000E+0				
0.00625	0.019634954	0.00000000E+0				
	EXACT SOLUTION	**6.92826363E+0**				
x =	**0.70**					
y =	**1.41**					
t =	**0.20**					
0.1	0.314159265	3.30183838E+0	5.37760E+04	5.50235E+04	**43.11**	57166.7
0.05	0.157079633	3.85207371E+0	1.24753E+03	9.62510E-01	**1.00**	
0.025	0.078539816	3.85206409E+0	1.24657E+03			
0.0125	0.039269908	0.00000000E+0				
0.00625	0.019634954	0.00000000E+0				
	EXACT SOLUTION	**3.83959841E+0**				

Figure A- 22. Snapshots of profiles along the principal axes, for the linear problem in Spherical system, *Test Problem #32* (Table 7). (a) Snapshot profile parallel to the x-axis, at y = 0.15, t = 0.20. (b) Snapshot profile parallel to the y-axis, at x = 0.09, t = 0.20. Data from *Dsnap* output file. It must be noted that for the solution used to generate this problem, <u>errors are magnified by a factor *2.5 x 10^6*</u> (see Table A-7). Therefore the <u>errors are extremely magnified at *x=0*</u>, as shown here and in Figure A-23.

(a)



(b)

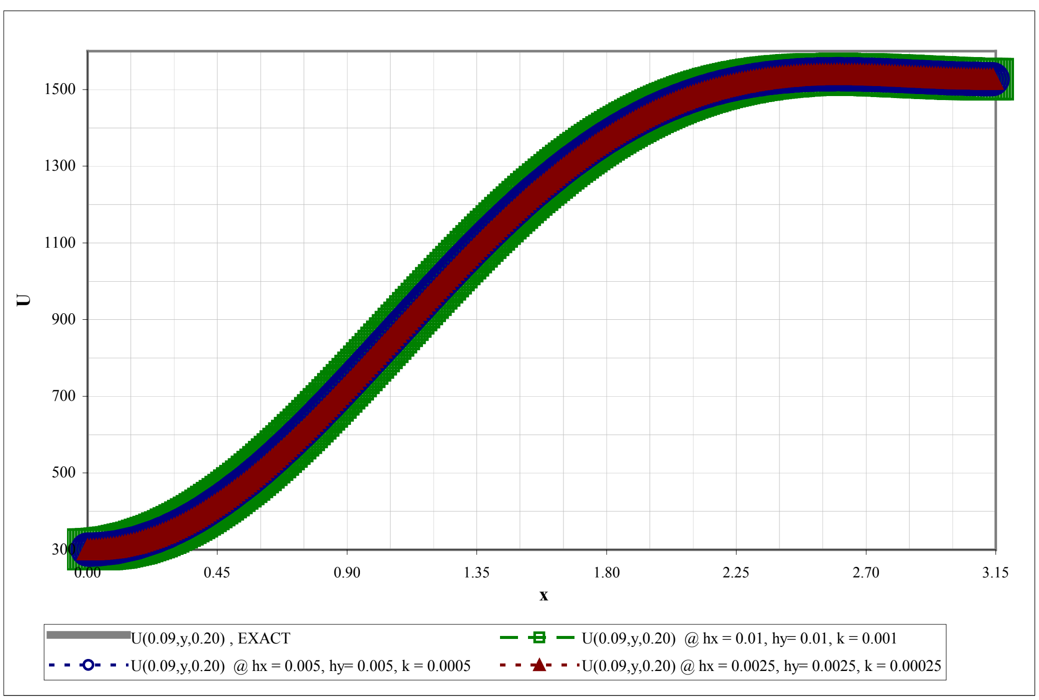
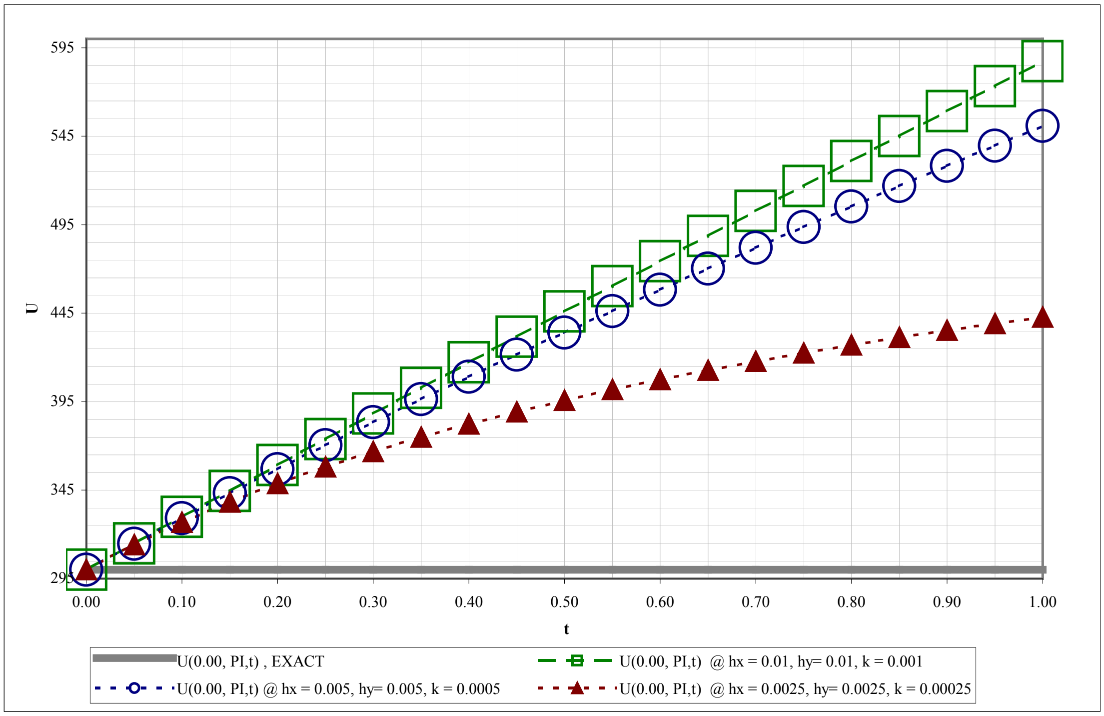


Figure A- 23. Evolution of grid functions with time, for the linear problem in Spherical system, Test Problem #32 (Table 7): x = 0.0, y =π. Data from Devol output file. It must be noted that for the solution used to generate this problem, errors are magnified by a factor 2.5 x 106 (see Table A-7). Therefore, the errors are extremely magnified at x=0, as shown here and in Figure A-22a.



A-3.4.1 Brief summary of validation tests

From the tests conducted (Table 7), it was deduced that the performance of *COND2D* was as predicted by theory, for "well behaved" problem data (BCs, thermal properties). The code can be used for a large number of problem types and coordinate systems. However, as seen from Table A-12 above, and Figures A-22(a) and A-23, when the thermal properties are highly nonlinear as was the case for *Tests Problem #30* onwards, and steep gradients exist in the solution (as simulated by the large, *2.5x* 10^6, factor in those solutions), very high resolutions are required to observe the second order convergence predicted by theory. In fact the solution chosen for these problems is a tough one since the gradients are everywhere extremely large due to the uniform domain-wise multiplication factor. Also, the truncation error at x=0 gets magnified by this multiplicative factor and therefore leads to large errors at the origin. So, what was observed in the last table and last set of figures was an artifact of the type of solution chosen, and not a problem with the code, as illustrated for the "well behaved" nonlinear case of *Test Problem #27*. So, if *COND2D* is to be applied to highly nonlinear problems, it seems imperative that very high resolutions be used – which means very long run times. This can be practically accomplished either by parallelizing the code to distribute the work load due to a large increase in time steps, or by finding ways of subdividing the domain of interest to reduce array sizes at a given resolution (and hence the arithmetic). In a typical application, a combination of these two approaches might have to be used.

REFERENCES

Bellman, R., 1948. On the existence and boundedness of solutions of nonlinear partial differential equations of parabolic type. Transactions of the American Mathematical Society, 64, 21-44.

Broadbridge, P., Lavrentiev, M. M., and Williams, G. H., 1999. Nonlinear heat conduction through an externally heated radiant plasma: Background analysis for a numerical study. Journal of Mathematical Analysis and Applications, 238, 353-368.

Byrelee, J., 1978. Friction of Rocks. Pure and Applied Geophysics,116, 617-626.

Carslaw, H. S., and Jaeger, J. C., 1959. Conduction of Heat in Solids. Oxford University Press. New York.

Dendy, J. E., 1977. Alternating direction methods for nonlinear time-dependent problems. SIAM Journal on Numerical Analysis, 14, 313-326.

Douglas, J. and Gunn, J. E., 1964. A general formulation of alternating direction methods, Part I: Parabolic and hyperbolic problems, Numerische Mathematik, 6, 428-453.

Douglas, J. and Rachford, H. H., 1956. On the numerical solution of heat conduction problems in two and three space dimensions, Transactions of the American Mathematical Society, 82, 421-439.

HP Fortran 90 Users Guide, 1998. HP 9000 computers (1st Ed.). B3909-90002, Hewlett Packard, San Jose.

Kanda, R. V. S., 2003. This Thesis. Linear and Nonlinear Modeling of Asperity Scale Frictional Melting In Brittle Fault Zones. M.S. Thesis, University of Kentucky.

Logan, J. M., Teufel, L. W., 1986. The effect of normal stresses on the real area of contact during frictional sliding in rocks. Pure and Applied Geophysics, 124, 471-485.

McDonough, J., M., 2001. Lectures in Basic Computational Numerical Analysis. Class Notes, http://www.engr.uky.edu/~egr537), University of Kentucky. . Email: jmmcd@uky.edu.

McDonough, J., M., 2002. Lectures on Computational Numerical Analysis of Partial Differential Equations. Class Notes, http://www.engr.uky.edu/~me690), University of Kentucky. Email: jmmcd@uky.edu.

McDonough, J. M., and Dong, S., 2001. 2-D to 3-D conversion for Navier-Stokes codes: parallelization issues. In Jenssen, C.B., Andersson, H.I., Ecer, A., Satofuka, N., Kvamsdal, T., Pettersen, B., Periaux, J., and Fox, P. (Eds.). Parallel Computational Fluid Dynamics – Trends and Applications: Proceedings of the Parallel CFD 2000 Conference. Elsevier, New York, 173-180.

Moon, P. H., and Spencer, D. E., 1988. Field Theory Handbook, Including Coordinate Systems, Differential Equations, and Their Solutions. Springer-Verlag, New York, 1-48.

Ranalli, G., 1995. Rheology of the Earth. Chapman & Hall, New York.

Timoshenko, S. P., Goodier, J. N., 1970. Theory of Elasticity. McGraw Hill. New York.

Touloukian, Y.S., Judd, W.R., Roy, R.F., 1981. Physical Properties of Rocks and Minerals. In Touloukian, Y.S., and Ho, C.Y. (Eds). McGraw-Hill/CINDAS Data Series on Material Properties, Volume II-2. McGraw Hill, New York.

Turcotte, D., L., and Schubert, G., 2001. Geodynamics. Cambridge, New York.

Yu, H., 2001. A local space-time adaptive scheme in solving two-dimensional parabolic problems based on domain decomposition methods. SIAM Journal on Scientific Computing, 23, 304-322.

APPENDIX B: *COND2D* -FORTRAN 90 CODE

```
!------------------------------------------------------------------------------------------------------------------------
!           Program for the solution of a GENERAL NON-LINEAR, 2D, TIME DEPENDENT HEAT CONDUCTION EQUATION (in Cartesian/
!           Cylindrical/Spherical coordinates OR in ANY USER DEFINED ANALYTIC SYSTEM), with general NON-LINEAR BOUNDARY CONDITIONS
!           USING DELTA-FORM OF QUASILINIARIZATION (NEWTON-KANTOROVICH PROCEDURE) IN CONJUNCTION WITH THE DELTA-FORM OF THE
!           DOUGLAS-GUNN TIME SPLITTING SCHEME (2-STEP). THIS CODE CAN ALSO BE USED FOR LINEAR PROBLEMS WITHOUT ANY CHANGES TO
!           THE CORE ALGORITHM IMPLEMENTED HERE. This code was written as part of the development of an "Asperity scale frictional
!           melting model" for my M.S. Thesis Research. This work was supported by NSF grant: XXXXX-XXXXX. - Ravi Kanda (November, 2002).
!           This program solves an equation of the form:
!           Ut = {1/(rho*cp)}*
!                [a1*{kt*(a2_x*U_x + a2*U_xx) + a2*kt_u*(U_x)^2} + b1*{kt*(b2_y*U_y + b2*U_yy) + b2*kt_u*(U_y)^2} + f(U,x,y,t)],
!           where the "_" denotes partial differentiation, obtained by expanding the ADJOINT form of the linear, but very general
!           Pure Conduction Equation. The values of functions a1, a2, b1, b2, kt(U) and cp(U) can be changed to match any
!           "regular closed domain" (i.e.. Cartesian, Cylindrical, Spherical, Elliptical or ANY USER DEFINED ANALYTIC SYSTEM domains),
!           in either of the three coordinate systems mentioned above. In addition, the treatment of the boundary conditions is very
!           general in that any type of convective/conductive/radiative heat transfer boundary condition can be applied at any of the
!           boundaries. The code adjusts the form of the equation in Spherical AND Cylindrical coordinates as r -> 0 ("left boundary"
!           in an equivalent cartesian grid representation), and in Spherical coordinates, as THETA -> 0 or PI. In these cases, the
!           coefficients of U_x (or U_y) in the generalized equation above (i.e., a2_x*a1 and b2_y*b1) are not ANALYTIC. The form
!           of the coordinate system can be specified using a "coord_flag" in the module "const_params". This program computes
!           the number of points in the spatial and time domains based on user supplied values of hx, hy & k, and computes the
!           "evloution" of the grid functions, Uji, for each "grid node" with time.
!
!           NOTE: IF A USER DEFINED SYSTEM IS CHOSEN, with NON-ANALYTIC {a1, a2, b1, b2}, THESE FUNCTIONS AND THEIR DERIVATIVES MUST
!           BE DEFINED CORRECTLY IN THE SUBROUTINES OF THE MODULE "pde_routines". CARE MUST ALSO BE TAKEN TO APPROPRIATELY
!           IMPLEMENT THE "INTERIOR" LOOP AND ALL THE "BOUNDARY CONDITION" LOOPS, IN THE SUBROUTINE "qldgts_coeff_rhs".
!
!           NOTE: For use with highly non-linear problems, a smoothing flag and parameter can be prescribed by the user, in the command
!           line, following the executable name. Either 1D or 2D Smoothing can be carried out using the simple Shuman filter, a low-pass
!           filter, that basically smooths out gradients in the domain at the end of each time step, at points (determined explicitly by
!           the user). IF SMOOTH FLAG IS NON-ZERO, THEN APPROPRIATE CHANGES NEED TO BE MADE BELOW, IN THE MAIN PROGRAM, TO MODIFY APPROPRIATE
!           GRID VALUES OF U.
!
!           The boundary conditions are specified in separate functions, as are the forcing function, f_rhs and the
!           exact solution, if known. f_rhs can be combined into the function f appearing in the general form of the equation
!           above to simplify the implementation and make it more flexible in incorporating certain non-linearities. Boundary
!           condition flags can be specified at two levels - linearity & type of BC (Dirichlet, Neumann or Robin) in the MAIN PROGRAM,
!           but defined in the module "Const_Params". This allows for SEVERAL changes in Boundary Condition types,
!           with time [as when an Initial Neumann BC changes later to a Dirichlet BC]. Further details of boundary condition
!           implementation are presented under the subroutine "qldgts_coeff_rhs", above. The initial condition is specified under a
!           separate function, and is passed on to the "qldgts" subroutine for the first time step. Time stepping is controlled by the
!           main program, which outputs data at selected time levels (user specified in the main program) to various output files to
!           facilitate easy post-processing. Subroutine "qldgts" outputs the values of the grid function Uji, at each time step, in a
!           two dimensional array in yj, and xi. The number of time steps to be plotted or gridded, as well as the number of output
!           files can be changed (by changing the "out" parameter array size and adding/removing file name elements in the "const_params"
!           module) can be changed in the main program. The program allows the output of grid function and plot data at any resolution
!           that the user chooses, with the maximum ALLOWED resolution, of course, being hx*hy. If lower resolutions of hx and hy than
!           allowed by the machine array limitations are needed, the code can be modified later to completely eliminate storage
!           in large arrays, and instead, directly print out only the required plot data to output files. Evolution of maximum
!           temperature is output to the screen at a few specified time levels.EXTENSIVE checks have been added to all subroutines
!           to improve ERROR TRAPPING.
!------------------------------------------------------------------------------------------------------------------------
MODULE const_params
IMPLICIT NONE
SAVE

!           Set precision and exponent required:

            INTEGER, PARAMETER :: rp = SELECTED_REAL_KIND(P=15, R=307), ip = SELECTED_INT_KIND(8)

!           INPUT/OUTPUT FILES: Specifying Output file pointers and output file names:

            INTEGER(KIND=ip) :: io
            INTEGER(KIND=ip), DIMENSION(5), PARAMETER :: out = (/ (io, io=1,5) /)
            CHARACTER(LEN=5), DIMENSION(SIZE(out)), PARAMETER :: outfile = (/ "Dgrid", "Derrg", "Dsnap", "Devol", "Dconv" /)

!           Mathematical Constants:

            REAL(KIND=rp), PARAMETER :: pi = 3.1415926535897932_rp, pi_sq = pi*pi

!           PDE Algorithm Limits: Coefficient magnitude limit; Grid size limit (usu. machine dependent):

            INTEGER(KIND=ip), PARAMETER :: max_points = 1000001
            REAL(KIND=rp), PARAMETER :: epsilon = 1.0E-30_rp            ! This parameter is for the LU-Decomposition Routine.

!           PDE Parameters:
!           -----------------
!           PDE LINEARITY FLAG : 1 for Linear, 0 (ZERO) for Non-Linear.
!           This will determine if the Newton-Kantorowich loop will be executed, or ONLY the Douglas-Gunn Time splitting
!           algorithm implemented, as is required for linear problems. Depending on the value of the LINEAR_FLAG, the grid
!           convergence tolerance is set in the MAIN PROGRAM. If LINEAR_FLAG = 1, this number is set to a very large number,
!           so the "qlindgts" loop is exited after one run:

            INTEGER(KIND=ip), PARAMETER :: linear_flag = 0

!           PDE COORDINATE SYSTEM FLAG: 0= User Specified PDE Coeffs, 1= Cartesian, 2= Cylindrical, and 3= Spherical.
!           If this flag is set to 0, the user needs to specify the functional form of the PDE coefficients a1, a2, b1,
!           b2, and their derivatives a2_x & b2_y, in MODULE "pde_routines":

            INTEGER(KIND=ip), PARAMETER :: coord_flag = 3

!           SMOOTHING FLAG: THIRD ARGUMENT AFTER THE PROGRAM EXECUTABLE. For highly non-linear problems, this smooths out the solution at the end
!           of each iteration at points (determined explicitly by the user) using either 1D or 2D smoothing. IF THIS VALUE IS NON-ZERO, THEN APPROPRIATE
!           CHANGES NEED TO BE MADE TO THE SUBROUTINE "qlin_dgts" TO MODIFY THE APPROPRIATE GRID VALUES OF U. Values for this flag are:
!           smooth_flag = 0, no smoothing, smooth_flag = 1, 1D smoothing, smooth_flag = 2, 2D smoothing.
!           NOTE: If smooth_flag is NON-ZERO, then a degree of smoothing between 2 and 1000 as the last argument after the program executable.
!           The larger the smoothing factor, the lesser the smoothing. The larger this value, the greater this smoothing.
!           DEFINE THESE TWO PARAMETERS GLOBALLY.

            INTEGER(KIND=ip) :: smooth_flag
            REAL(KIND=rp) :: smooth_factor

!           PDE BOUNDARY CONDITION FLAGS: SPECIFICATION HAS BEEN MOVED TO MAIN PROGRAM, TO ACCOMODATE TIME VARYING BC Types (once or several times -
!           as prescribed in the MAIN PROGRAM: Neumann to Dirichlet, and back, for instance). However, the flags have to be defined globally, for access
!           by various subroutines.
            INTEGER(KIND=ip) :: left_bc_flag, right_bc_flag, bottom_bc_flag, top_bc_flag, &
                                    & left_lin_flag, right_lin_flag, bottom_lin_flag, top_lin_flag
```

```
!           RBC Temperature SPECIFICATION: FOR TEST PROBLEM ONLY!
            REAL(KIND=rp), PARAMETER :: u_right = 300.0_rp

!           OPTIONAL Linear Robin Parameters, ALPHA_x & ALPHA_y for each of the two directions. Eg., in: L = U_x + alpha_x * U
            REAL(KIND=rp) :: alpha_x, alpha_y

            !           PDE BOUNDARY CONDITION FLAGS:       0 for DIRICHLET {i.e., Bbc(U) = B2bc(U)},
            !                                                           1 for NEUMANN {i.e., Bbc(U) = U_x*B1bc(U)},
            !                                                           2 for ROBIN {i.e., Bbc(u) = U_x*B1bc(U)}.
            !           All BCs are represented in the generalized non-linear forms encountered in heat conduction problems:
            !           Bbc(U) = U_x*B1bc(U)+ B2bc(U) or U_y*B1bc(U)+ B2bc(U). This form can be used to represent either NON-LINEAR or
            !           LINEAR BCs. PROVIDE ALL BOUNDARY OPERATORS, B, in this SPLIT FORM, using separate functions for B1 and B2, for
            !           EACH BC. These classifications and their implementations are discussed under the separate functions in the module
            !           "pde_routines", below, and ESPECIALLY UNDER THE SUBROUTINE "qldgts_coeff_rhs", where they are used:
            !
            !           INTEGER(KIND=ip), PARAMETER :: left_bc_flag  = 1, right_bc_flag  = 1, bottom_bc_flag  = 1, top_bc_flag  = 1
            !           OPTIONAL Linear Robin Parameters, ALPHA_x & ALPHA_y for each of the two directions. Eg., in: L = U_x + alpha_x * U
            !           REAL(KIND=rp), PARAMETER :: alpha_x = 0.0_rp, alpha_y = 0.0_rp
            !
            !           BOUNDARY CONDITION LINEARITY FLAGS: 1 if linear, 0 if non-linear.
            !           These will affect the forms and values of the corresponding boundary condition functionals (lbc1, bbc1, tbc2, etc.)
            !           below. If any of these flags is 0 (non-linear BC) then the forms of these functionals have to be defined in the
            !           respective subroutines in MODULE "pde_routines":
            !
            !           INTEGER(KIND=ip), PARAMETER :: left_lin_flag = 1, right_lin_flag = 0, bottom_lin_flag = 1, top_lin_flag = 1

!           PDE EXACT SOLUTION FLAG: Set this flag to 1 if the closed form of the exact analytical solution to this problem is
!           known. Then set it up under the function "f_exact". If no exact solution exists, or is not available, set this flag
!           to 0. This will affect the type of diagnostic information the program outputs for this problem. If exact solution
!           exists, the program computes and outputs the exact error, otherwise, it outputs an estimated value based on
!           iteration errors and the "asymptotic spectral radius" of the spatial discretization matrix.

            INTEGER(KIND=ip), PARAMETER :: exact_sol_flag = 1

!           PDE DOMAIN DECLARATION AND LOWEST PERMITTED GRID RESOLUTION:
!           (a) PDE DOMAIN SPECIFICATION:
!                                                       NOTE: Changing x-range affects x_snap and x_time & grid_conv(:,1) below!
!                                                       Similarly,     y-range affects y_snap and y_time & grid_conv(:,2) below!
!                                                                                      ------     ------   -------------
!                                                       THIS ALSO AFFECTS t0, the pulse duration, and hence, t_snap values below!
!                                                                         --                                  ------

            REAL(KIND=rp), PARAMETER ::             x_left = 0.0_rp, x_right = 0.1_rp, &
                                                        &             y_bottom = 0.0_rp,   y_top = pi,     &
                                                        &            t_initial = 0.0_rp, t_final = 1.0_rp


!           (b) SMALLEST GRID RESOLUTION: Define the maximum allowable grid spacings. The main program specifies different resolutions
!           using the grid resolution flag, "res_flag" (see Main Program):

            REAL(KIND=rp), PARAMETER :: hx_max = (x_right - x_left)/10.0_rp, hy_max = (y_top - y_bottom)/10.0_rp

!           INPUT/OUTPUT PARAMETERS: Specify output grid spacings for solution evolution, grid and plot files defined above. Note that
!           the grid and plot grid spacings can be reassigned in the main program if these resolutions are finer than hx or hy. Also
!           specify the time levels at which the plot and grid output is written out to the corresponding output files.

            REAL(KIND=rp), PARAMETER :: tf = t_final
            REAL(KIND=rp) :: out_x_grid_spacing = hx_max/2.0_rp, out_y_grid_spacing = 0.010_rp, t_evol_spacing = tf/20.0_rp

            REAL(KIND=rp), DIMENSION(11), PARAMETER :: t_snap = (/      t_initial,  0.15_rp*tf, 0.20_rp*tf, 0.30_rp*tf, 0.40_rp*tf, &
                                                                                    0.50_rp*tf, 0.60_rp*tf, 0.70_rp*tf, 0.80_rp*tf, 0.90_rp*tf, &
                                                                                    t_final /)

!           CONVERGENCE & EVOLUTION PARAMETERS:
!           (a) SNAPSHOT OF PROFILE ALONG A LINE PARALLEL TO x-axis:

            REAL(KIND=rp), PARAMETER :: y_xsnap = 0.20_rp, t_xsnap = t_snap(2)

!           (b) SNAPSHOT OF PROFILE ALONG A LINE PARALLEL TO y-axis:

            REAL(KIND=rp), PARAMETER :: x_ysnap = 0.90_rp*x_right, t_ysnap = t_snap(3)

!           (c) EVOLUTION OF GRID FUNCTION VALUES AT A SINGLE GRID POINT AS A FUNCTION OF TIME, t:

            REAL(KIND=rp), PARAMETER :: x_time = x_left, y_time = y_top

!           (d) POINT GRID CONVERGENCE TEST LOCATIONS - 8 points, at different space & time coordinates:

            REAL(KIND=rp), PARAMETER :: xr = x_right, yt = y_top
            REAL(KIND=rp), DIMENSION(8,3), PARAMETER :: grid_conv = RESHAPE(                                                                  &
                                                        (/0.55_rp*xr, 0.60_rp*xr, 0.65_rp*xr, 0.70_rp*xr, 0.80_rp*xr, 0.75_rp*xr, 0.90_rp*xr, 0.70_rp*xr,  &
                                                          0.10_rp*yt, 0.50_rp*yt, 0.90_rp*yt, 0.15_rp*yt, 0.95_rp*yt, 0.50_rp*yt, 0.40_rp*yt, 0.45_rp*yt,  &
                                                          t_snap(2), t_snap(2), t_snap(2), t_snap(2), t_snap(3), t_snap(3), t_snap(3), t_snap(3)/), &
                                                        (/ 8,3 /))

!           (e) SET THE LEVEL OF DETAIL IN SCREEN OUTPUT: Set verbose_flag = 1 if detailed output is required at every time step on grid function maxima
!               as well as non-linear iteration convergence information at each time step:

            INTEGER(KIND=ip), PARAMETER :: verbose_flag = 1

!           GLOBAL VARIABLES:

!           Define the variables "quasi_epsilon" for iteration tolerance, and "quasi_iterations" for the max number of
!           Newton-Kantorovich iterations. Due to the quadratic convergence expected if this method works, this number
!           need not be large (about 10-15 is "quite sufficient").
            REAL(KIND=rp) :: quasi_epsilon
            INTEGER(KIND=ip) :: quasi_iterations

!           Declare all arrays required by subroutine "delta_qlin_dgts" here, and allocate them through the MAIN program:
            REAL(KIND=rp), ALLOCATABLE, DIMENSION(:,:) :: coeff, u_n, u_old
            REAL(KIND=rp), ALLOCATABLE, DIMENSION(:) :: rhs, rs

!           Save one of the FUNCTIONAL DERIVATIVE VALUES globally to conserve arithmetic in the "qldgts_coeff_rhs" routine, as
!           they are used in both time stages of the D-G discretization.
            REAL(KIND=rp), ALLOCATABLE, DIMENSION(:,:) :: NSu_m, Nu_m

END MODULE const_params

!-----------------------------------------------------------------------------------------------------------------------------
```

```
MODULE fault_params
USE const_params
IMPLICIT NONE
SAVE
!-----------------------------------------------------------------------------------------------------------------------
!           Set the values of the physical fault parameters and/or their ranges: All units in SI system, and for QUARTZ.
!
!           Where indicated, the temperature dependence of parameters, and their extrema are adapted from: Touloukian, Y.S., Judd, W.R.,
!           and Roy, R.F., "Physical Properties of Rocks and Minerals.", in Touloukian, Y.S., and Ho, C.Y., Ed., "McGraw-Hill/CINDAS Data
!           Series on Material Properties", Volume II-2, McGraw Hill, New York, 1981.
!
!           DEFINITIONS:
!           ------------
!           asp_rad  = asperity radius (m)
!           cp       = Specific Heat at constant Pressure (J/kg-K). [MIN & MAX values based on ambient T=300K, and QUARTZ melting temp, 1700k, respectively.]
!           e_y      = Young's Modulus for asperity material (GPa)
!           kappa    = thermal diffusivity (m^2/s)
!           kt       = thermal conductivity (W/m-K). [MAX & MIN values based on ambient T=300K, and QUARTZ melting temp, 1700k, respectively.]
!           mu          = Coefficient of rock friction (dimensionless). [MIN & MAX values based on Byrelee's results: 0.6-0.85]
!           nu_ps    = Poisson's Ratio (dimensionless)
!           rho      = Density of asperity material         (kg/m^3). [MAX & MIN values based on Variation in composition of FELSIC rocks.]
!           slip_v   = slip velocity (m/s)
!           tau      = shear stress (Pa)        [MAX & MIN values based on Nadeau and Johnson, 1998 & Logan & Teufel, 1986 - See Thesis References]
!-----------------------------------------------------------------------------------------------------------------------
            REAL(KIND=rp), PARAMETER ::         asp_rad_min = 0.001_rp,                                            &
                                                            asp_rad_max = 1.00_rp,                                             &
                                                            cp_min          =  447.50_rp + 1.025_rp* 300.0_rp,                 &
                                                            cp_max      = 1093.80_rp + 0.100_rp*1700.0_rp,                 &
                                                            e_y         = 20.0_rp,                                         &
                                                            kt_min      = 0.9452102585026962_rp,                               &
                                                            kt_max          = 7.5420193400746197_rp,                           &
                                                            mu_min          = 0.60_rp,                                                     &
                                                            mu_max          = 0.85_rp,                                                     &
                                                            nu_ps       = 0.20_rp,                                             &
                                                            rho_min     = 2500.0_rp,                                           &
                                                            rho_max     = 3000.0_rp,                                           &
                                                            slip_v_min  = 0.1_rp,                                              &
                                                            slip_v_max  = 1.0_rp,                                              &
                                                            tau_min     = 1.0E6_rp,                                            &
                                                            tau_max     = 1.0E9_rp

!           ASSIGN VALUES for Fault Parameters for this run: Parameters defined here for the first time are:
!                       rc = Radius of circular contact area between two ELASTIC spheres.
!                       t0 = Time taken for the two contacting spheres to pass each other - time duration of heat flux input from frictional contact.
!           NOTE: THE CONST VALUES FOR LINEAR PROBLEM ARE TEMPERATURE WEIGHTED AVERAGES.
            REAL(KIND=rp), PARAMETER ::         cp_const = 1167.95_rp,                                             &
                                                            kt_const = 3.03_rp,                                                &
                                                            mu            = mu_min,                                            &
                                                            rho      = rho_max,                                                &
                                                            slip_v   = slip_v_max,                                             &
                                                            tau      = tau_max,                                                &
                                                            rc_by_r0 = 3.0_rp*pi*(1.0_rp - nu_ps*nu_ps)*tau/(4.0_rp*e_y*1.0E9_rp*mu),          &
                                                            rc       = rc_by_r0*x_right,                                       &
                                                            t0       = 4.0_rp*rc/slip_v

!           DEFINE FAULT PARAMETERS THAT NEED TO BE ACCESSIBLE GLOBALLY:
!           y0 = Half the angle (theta) subtended at the center of either asperity, by the circular contact area.

            REAL(KIND=rp) :: y0

END MODULE fault_params

!-----------------------------------------------------------------------------------------------------------------------
MODULE pde_routines
USE const_params
USE fault_params

            CONTAINS

            !-----------------------------------------------------------------------------------------------------------------------
            FUNCTION kt(u,x,y,t)
            IMPLICIT NONE
            !           This function computes the value of the temperature dependent THERMAL conductivity, kt, that appears in the
            !           PDE: a1*(a2*kt*U_x)_x + b1*(b2*kt*U_y)_y + f(U,x,y,t) = rho*cp*U_t, where the "_" denotes a partial derivative.
            !           Functional expression is assigned for the NON-LINEAR case. Otherwise, it is set to the constant value
            !           prescribed in the module "fault_params" above. Since kt appears in Nuxx_m & Nuyy_m, which are part of coeff denominators, it
            !           cannot have a zero value.
            !           FUNCTIONAL FORM OF kt is a BEST FIT CURVE (kt = 1 + a/(U**b)) TO THE data for QUARTZ adapted from: Touloukian, Y.S., Judd, W.R.,
            !           and Roy, R.F., "Physical Properties of Rocks and Minerals.", in Touloukian, Y.S., and Ho, C.Y., Ed., "McGraw-Hill/CINDAS Data
            !           Series on Material Properties", Volume II-2, McGraw Hill, New York, 1981.
            !           Function & Arguments
                        REAL(KIND=rp), INTENT(IN) :: u
                        REAL(KIND=rp), INTENT(IN), OPTIONAL :: x,y,t
                        REAL(KIND=rp) :: kt

                        IF (linear_flag == 1) THEN
                                    kt =  kt_const
                        ELSE
                                    kt = 1.0_rp + 14.2920_rp*(EXP(-0.0030_rp*u))
!                                   ! R^2 fit value = 0.9953 in the range 300-1000K; U in DEGREES KELVIN. kt has a slope > 1 for u <~ -1050
!                                   kt = 1.0_rp + u
!                                   IF (u < 0.0_rp) THEN
!                                               kt = 15.2920_rp
!                                   ELSE
!                                               kt = 1.0_rp + 14.2920_rp*(EXP(-0.0030_rp*u))    ! R^2 fit value = 0.9953 in the range 300-1000K; U in DEGREES KELVIN.
!                                   END IF

!                                   IF ( u <= 0.0_rp ) THEN  ! Based on functional limitation in Cp expression.
!                                               kt = 1.0_rp + (162144.4558_rp)*(20.0_rp**(-1.7559_rp))
!                                   ELSE
!                                               kt = 1.0_rp + (162144.4558_rp)*(u**(-1.7559_rp)) ! R^2 fit value = 0.9838 in the range 300-1000K; U in DEGREES KELVIN.
!                                   END IF
                        END IF

            END FUNCTION kt

            !-----------------------------------------------------------------------------------------------------------------------
```

```
            FUNCTION kt_u(u,x,y,t)
            IMPLICIT NONE
            !           This function computes the value of the FIRST temperature derivative of THERMAL conductivity, kt, that
            !           appears in the PDE: a1*(a2*kt*U_x)_x + b1*(b2*kt*U_y)_y + f(U,x,y,t) = rho*cp*U_t, where the "_" denotes a
            !           partial derivative. Functional expression is assigned for the NON-LINEAR case. It is equal to 0 for the LINEAR
            !           CASE.
            !           Function & Arguments
                        REAL(KIND=rp), INTENT(IN) :: u
                        REAL(KIND=rp), INTENT(IN), OPTIONAL :: x,y,t
                        REAL(KIND=rp) :: kt_u

                        IF (linear_flag == 1) THEN
                                    kt_u = 0.0_rp
                        ELSE
                                    kt_u = 14.2920_rp*(-0.0030_rp)*(EXP(-0.0030_rp*u))                                  ! Based on the Definition of kt above.
!                                   kt_u = 1.0_rp
!                                   IF (u < 0.0_rp) THEN
!                                               kt_u = 0.0_rp
!                                   ELSE
!                                               kt_u = 14.2920_rp*(-0.0030_rp)*(EXP(-0.0030_rp*u))                      ! Based on the Definition of kt above.
!                                   END IF
!                                   IF (u <= 20.0_rp) THEN  ! Based on functional limitation in Cp expression.
!                                               kt_u = 0.0_rp
!                                   ELSE
!                                               kt_u = ( (162144.4558_rp)*(-1.7559_rp) )*(u**(-1.7559_rp - 1.0_rp))     ! Based on the Definition of kt above.
!                                   END IF
                        END IF

            END FUNCTION kt_u

            !-----------------------------------------------------------------------------------------------------------------
            FUNCTION kt_uu(u,x,y,t)
            IMPLICIT NONE
            !           This function computes the value of the SECOND temperature derivative of THERMAL conductivity, kt, that
            !           appearsin the PDE: a1*(a2*kt*U_x)_x + b1*(b2*kt*U_y)_y + f(U,x,y,t) = rho*cp*U_t, where the "_" denotes a
            !           partial derivative. Functional expression is assigned for the NON-LINEAR case. It is equal to 0 for the LINEAR
            !           CASE.
            !           Function & Arguments
                        REAL(KIND=rp), INTENT(IN) :: u
                        REAL(KIND=rp), INTENT(IN), OPTIONAL :: x,y,t
                        REAL(KIND=rp) :: kt_uu

                        IF (linear_flag == 1) THEN
                                    kt_uu = 0.0_rp
                        ELSE
                                    kt_uu = 14.2920_rp*(-0.0030_rp)*(-0.0030_rp)*(EXP(-0.0030_rp*u))        ! Based on the Definition of kt above.
!                                   kt_uu = 0.0_rp
!                                   IF (u < 0.0_rp) THEN
!                                               kt_uu = 0.0_rp
!                                   ELSE
!                                               kt_uu = 14.2920_rp*(-0.0030_rp)*(-0.0030_rp)*(EXP(-0.0030_rp*u))        ! Based on the Definition of kt above.
!                                   END IF
!                                   IF (u <= 20.0_rp) THEN  ! Based on functional limitation in Cp expression.
!                                               kt_uu = 0.0_rp
!                                   ELSE
!                                               kt_uu = ( (162144.4558_rp)*(-1.7559_rp)*(-1.7559_rp) )*(u**(-1.7559_rp - 2.0_rp)) ! Based on the Definition of kt above.
!                                   END IF
                        END IF

            END FUNCTION kt_uu

            !-----------------------------------------------------------------------------------------------------------------
            FUNCTION cp(u,x,y,t)
            IMPLICIT NONE
            !           This function computes the value of the temperature dependent SPECIFIC HEAT (THERMAL HEAT CAPACITY), c, that
            !           appears in the PDE: a1*(a2*kt*U_x)_x + b1*(b2*kt*U_y)_y + f(x,y,t) = rho*cp*U_t, where the "_" denotes a
            !           partial derivative. Functional expression is assigned for the NON-LINEAR case. Otherwise, it is set to the
            !           constant value prescribed in the module "const_params" above.Since Cp appears in in the denominators of ALL Functionals, it
            !           cannot have a zero value.
            !           FUNCTIONAL FORM OF Ct is a BEST FIT CURVE (Cp = a*LN(U) + b) TO THE data for QUARTZ adapted from: Touloukian, Y.S., Judd, W.R.,
            !           and Roy, R.F., "Physical Properties of Rocks and Minerals.", in Touloukian, Y.S., and Ho, C.Y., Ed., "McGraw-Hill/CINDAS Data
            !           Series on Material Properties", Volume II-2, McGraw Hill, New York, 1981.
            !           Function & Arguments
                        REAL(KIND=rp), INTENT(IN) :: u
                        REAL(KIND=rp), INTENT(IN), OPTIONAL :: x,y,t
                        REAL(KIND=rp) :: cp

                        IF (linear_flag == 1) THEN
                                    cp =  cp_const
                        ELSE
                                    cp = 1500.0_rp*( 1.0_rp - 0.5105_rp*EXP(-0.0008_rp*u) )
!                                   ! R^2 fit value = 0.84 in 300-1500K; U in DEGREES KELVIN. Cp has a slope > 1 for u<~ -9755, and is NEGATIVE for u<~ -841.
!                                   cp = 1.0_rp + u
!                                   IF (u < 0.0_rp) THEN
!                                               cp = 734.25_rp
!                                   ELSE
!                                               cp = 1500.0_rp*( 1.0_rp - 0.5105_rp*EXP(-0.0008_rp*u) ) ! R^2 fit value = 0.84 in 300-1500K; U in DEGREES KELVIN.
!                                   END IF
!                                   IF (u <= 20.0_rp) THEN
!                                               cp = 299.24_rp*(LOG(20.0_rp)) - 891.19_rp
!                                   ELSE
!                                               cp = 299.24_rp*(LOG(u)) - 891.19_rp ! R^2 fit value = 0.90 in 300-1500K; U in DEGREES KELVIN.
!                                   END IF
                        END IF
            END FUNCTION cp
            !-----------------------------------------------------------------------------------------------------------------
            FUNCTION cp_u(u,x,y,t)
            IMPLICIT NONE
            !           This function computes the value of the FIRST temperature derivative of SPECIFIC HEAT (THERMAL HEAT CAPACITY),
            !           cp, that appears in the PDE: a1*(a2*kt*U_x)_x + b1*(b2*kt*U_y)_y + f(x,y,t) = rho*cp*U_t, where the "_" denotes
            !           a partial derivative. Functional expression is assigned for the NON-LINEAR case. It is equal to 0 for the
            !           LINEAR CASE.
            !           Function & Arguments
                        REAL(KIND=rp), INTENT(IN) :: u
                        REAL(KIND=rp), INTENT(IN), OPTIONAL :: x,y,t
                        REAL(KIND=rp) :: cp_u

                        IF (linear_flag == 1) THEN
                                    cp_u = 0.0_rp
                        ELSE
                                    cp_u = 1500.0_rp*(- 0.5105_rp)*(-0.0008_rp)*EXP(-0.0008_rp*u)  ! From the expression for Cp defined above.
!                                   cp_u = 1.0_rp
!                                   IF (u < 0.0_rp) THEN
!                                               cp_u = 0.0_rp
```

```
!                                   ELSE
!                                               cp_u = 1500.0_rp*(- 0.5105_rp)*(-0.0008_rp)*EXP(-0.0008_rp*u)  ! From the expression for Cp defined above.
!                                   END IF
!                                   IF (u <= 20.0_rp) THEN
!                                               cp_u = 299.24_rp/(20.0_rp)
!                                   ELSE
!                                               cp_u = 299.24_rp/(u)                            ! From the expression for Cp defined above.
!                                   END IF
                        END IF

            END FUNCTION cp_u

            !------------------------------------------------------------------------------------------------------------
            FUNCTION f_exact(x,y,t)
            IMPLICIT NONE
            !           Function & Arguments
                        REAL(KIND=rp), INTENT(IN) :: t,x,y
                        REAL(KIND=rp) :: f_exact

            !           Local Variables
                        REAL(KIND=rp) :: sy

                        sy = SIN(y)
                        f_exact = 300.0_rp + 2500000.0_rp*(EXP(-t))*(x - SIN(x))*( (y*y/2.0_rp) + y*sy + sy*sy )

            END FUNCTION f_exact

            !------------------------------------------------------------------------------------------------------------
            FUNCTION f_initial(x,y)
            IMPLICIT NONE
            !           Function & Arguments
                        REAL(KIND=rp), DIMENSION(:), INTENT(IN) :: x, y
                        REAL(KIND=rp), DIMENSION(SIZE(y), SIZE(x)) :: f_initial

            !           Local Variables
                        INTEGER(KIND=ip) :: i,j
                        REAL(KIND=rp) :: sxi, syj, xi, yj

                        DO i = 1, SIZE(x)
                                    DO j = 1, SIZE(y)
                                                sxi = SIN(x(i))
                                                syj = SIN(y(j))
                                                xi = x(i)
                                                yj = y(j)
                                                f_initial(j,i) = 300.0_rp + 2500000.0_rp*(xi - sxi)*( (yj*yj/2.0_rp) + yj*syj + syj*syj )
                                    END DO
                        END DO

            END FUNCTION f_initial

            !------------------------------------------------------------------------------------------------------------
            FUNCTION f_rhs(u,x,y,t)
            IMPLICIT NONE
            !           In entering this function, and its overall sign, keep in mind its location in the general PDE being solved here:
            !           Ut = {1/(rho*cp)}*
            !           a1*{kt*(a2_x*U_x + a2*U_xx) + a2*kt_u*(U_x)^2} + b1*{kt*(b2_y*U_y + b2*U_yy) + b2*kt_u*(U_y)^2} + f(U,x,y,t)]

            !           Function & Arguments
                        REAL(KIND=rp), INTENT(IN) :: t, u, x, y
                        REAL(KIND=rp) :: f_rhs

            !           Local Variables
                        REAL(KIND=rp) :: cy, e2t, et, fx, fx1, fx2, gy, gy1, gy2, rho_cp, sy

                        cy          = COS(y)
                        et          = 2500000.0_rp*EXP(-t)
                        e2t         = et*et
                        fx          = x - SIN(x)
                        fx1         = 1.0_rp - COS(x)
                        fx2         = SIN(x)
                        sy          = SIN(y)
                        gy          = (y*y/2.0_rp) + y*sy + sy*sy
                        gy1         = y*(1.0_rp + cy) + sy*(1.0_rp + 2.0_rp*cy)
                        gy2         = 1.0_rp - y*sy + 2.0_rp*( cy + COS(2.0_rp*y) )
                        rho_cp      = rho*cp(u,x,y,t)
                        IF (x /= 0.0_rp) THEN
                                    IF ( (y == 0.0_rp) .OR. (y == pi) ) THEN
                                       f_rhs = (300.0_rp - u)*rho_cp - (  kt(u,x,y,t))*et *(  gy*( (2.0_rp*fx1/x) + fx2 ) + (fx/(x*x))*(     gy2     + gy2 )  ) &
                                                   &                           - (kt_u(u,x,y,t))*e2t*( fx1*fx1*gy*gy + (fx*fx*gy1*gy1/(x*x)) )
                                    ELSE
                                       f_rhs = (300.0_rp - u)*rho_cp - (  kt(u,x,y,t))*et *(  gy*( (2.0_rp*fx1/x) + fx2 ) + (fx/(x*x))*( (cy*gy1/sy) + gy2 )  ) &
                                                   &                           - (kt_u(u,x,y,t))*e2t*( fx1*fx1*gy*gy + (fx*fx*gy1*gy1/(x*x)) )
                                    END IF
                        ELSE
                                    f_rhs = 300.0_rp*rho_cp
                        END IF
            END FUNCTION f_rhs

            !------------------------------------------------------------------------------------------------------------
            FUNCTION f_rhs_u(u,x,y,t)
            IMPLICIT NONE
            !           This is the derivative of the right hand side function defined in the last subroutine with respect to the
            !           dependent variable U. The RHS function appears in the general PDE being solved here as shown:
            !           Ut = {1/(rho*cp)}*
            !           a1*{kt*(a2_x*U_x + a2*U_xx) + a2*kt_u*(U_x)^2} + b1*{kt*(b2_y*U_y + b2*U_yy) + b2*kt_u*(U_y)^2} + f(U,x,y,t)]

            !           Function & Arguments
                        REAL(KIND=rp), INTENT(IN) :: t, u, x, y
                        REAL(KIND=rp) :: f_rhs_u

            !           Local Variables
                        REAL(KIND=rp) :: cy, e2t, et, fx, fx1, fx2, gy, gy1, gy2, rho_cp, rho_cp_u, sy

                        cy           = COS(y)
                        et           = 2500000.0_rp*EXP(-t)
                        e2t          = et*et
                        fx           = x - SIN(x)
                        fx1          = 1.0_rp - COS(x)
                        fx2          = SIN(x)
                        sy           = SIN(y)
                        gy           = (y*y/2.0_rp) + y*sy + sy*sy
                        gy1          = y*(1.0_rp + cy) + sy*(1.0_rp + 2.0_rp*cy)
                        gy2          = 1.0_rp - y*sy + 2.0_rp*( cy + COS(2.0_rp*y) )
                        rho_cp       = rho*  cp(u,x,y,t)
```

```
          rho_cp_u = rho*cp_u(u,x,y,t)
          IF (x /= 0.0_rp) THEN
                    IF ( (y == 0.0_rp) .OR. (y == pi) ) THEN
                       f_rhs_u = -rho_cp - u*rho_cp_u - (kt_u(u,x,y,t))*et *( gy*( (2.0_rp*fx1/x) + fx2 ) + (fx/(x*x))*(    gy2     + gy2) ) &
                                &                                -(kt_uu(u,x,y,t))*e2t*( fx1*fx1*gy*gy + (fx*fx*gy1*gy1/(x*x)) )
                    ELSE
                       f_rhs_u = -rho_cp - u*rho_cp_u - (kt_u(u,x,y,t))*et *( gy*( (2.0_rp*fx1/x) + fx2 ) + (fx/(x*x))*((cy*gy1/sy) + gy2) ) &
                                &                                -(kt_uu(u,x,y,t))*e2t*( fx1*fx1*gy*gy + (fx*fx*gy1*gy1/(x*x)) )
                    END IF
          ELSE
                    f_rhs_u = -rho_cp
          END IF

END FUNCTION f_rhs_u

!------------------------------------------------------------------------------------------------------------------
FUNCTION f_left(y,t)
IMPLICIT NONE
!         Define LEFT BC - Just enter the functional representation. The type of BC (Dirichlet/Neumann/Robin) will be
!         determined from the value of the parameter "left_bc_flag" in the module CONST_PARAMS above.

!         Function & Arguments
          REAL(KIND=rp), INTENT(IN) :: t, y
          REAL(KIND=rp) :: f_left

          f_left = 0.0_rp

END FUNCTION f_left

!------------------------------------------------------------------------------------------------------------------
FUNCTION lbc1(u_j1,yj,tn)
IMPLICIT NONE
!         First component of the left BC operator, Lbc(U,x,y,t) {= Ux*Lbc1(U,x,y,t) + Lbc2(U,x,y,t) = f_left(y,t)}
!         Function & Arguments
          REAL(KIND=rp), INTENT(IN) :: tn,u_j1,yj
          REAL(KIND=rp) :: lbc1

          IF (left_lin_flag == 1) THEN                    ! Linear BC
                    IF (left_bc_flag == 0) THEN
                              lbc1 = 0.0_rp                                  ! Linear Dirichlet
                    ELSE
                              lbc1 = 1.0_rp                                  ! Linear Neumann or Robin
                    END IF
          ELSE                                                     ! Non-Linear BC
                    IF (left_bc_flag == 0) THEN
                              lbc1 = 0.0_rp                                  ! Non-Linear Dirichlet
                    ELSE
                              lbc1 = kt(u_j1,x_left,yj,tn)        ! Non-Linear Neumann or Robin.
                              IF (lbc1 == 0.0_rp) lbc1 = epsilon  ! lbc1 appears in the denominator of lbc_u for Non-Linear Neumann/Robin BCs.
                    END IF                                                   ! Can be any function of U as required by BC.
          END IF

END FUNCTION lbc1

!------------------------------------------------------------------------------------------------------------------
FUNCTION lbc2(u_j1,yj,tn)
IMPLICIT NONE
!         Second component of the left BC operator, Lbc(U,x,y,t) {= Ux*Lbc1(U,x,y,t) + Lbc2(U,x,y,t) = f_left(y,t)}
!         Function & Arguments
          REAL(KIND=rp), INTENT(IN) :: tn,u_j1,yj
          REAL(KIND=rp) :: lbc2

          IF (left_lin_flag == 1) THEN                    ! Linear BC
                    IF (left_bc_flag == 0) THEN
                              lbc2 = f_left(yj,tn)                 ! Linear Dirichlet
                    ELSE      IF(left_bc_flag == 1) THEN
                              lbc2 = 0.0_rp                                  ! Linear Neumann
                    ELSE
                              lbc2 = alpha_x*u_j1                  ! Linear Robin
                    END IF
          ELSE                                                     ! Non-Linear BC
                    IF (left_bc_flag == 1) THEN
                              lbc2 = 0.0_rp                                  ! Non-Linear Neumann
                    ELSE
                              lbc2 = 0.5_rp*u_j1*(1.0_rp + u_j1)  ! Non-Linear Dirichlet or Robin.
                    END IF                                                             ! Can be any function of U as required by BC.
          END IF
END FUNCTION lbc2
!------------------------------------------------------------------------------------------------------------------
FUNCTION lbc_u(u_j1,yj,tn)
IMPLICIT NONE
!         Derivative w.r.t U, of the ENTIRE left BC operator, Lbc(U,x,y,t) {= Ux*Lbc1(U,x,y,t) + Lbc2(U,x,y,t) = f_left(y,t)}.
!         The derivatives of the two individual components of the boundary operator (lbc1 and lbc2) are not required
!         separately by the algorithm used here.

!         Function & Arguments
          REAL(KIND=rp), INTENT(IN) :: tn,u_j1,yj
          REAL(KIND=rp) :: lbc_u

          IF (left_lin_flag == 1) THEN                    ! Linear BC
                    IF (left_bc_flag == 0) THEN
                              lbc_u = 1.0_rp                                 ! Linear Dirichlet
                    ELSE      IF(left_bc_flag == 1) THEN
                              lbc_u = 0.0_rp                                 ! Linear Neumann
                    ELSE
                              lbc_u = alpha_x                                ! Linear Robin
                    END IF
          ELSE                                                     ! ANY Non-Linear BC
                    ! For the above choices of lbc1 & lbc2 (both = U), this will take on the value Ux + 1. Ux can be obtained from
                    ! the left boundary condition as shown below.
                    IF (left_bc_flag == 0) THEN                                        ! NonLinear Dirichlet
                              lbc_u = 0.5_rp*(1.0_rp + 2.0_rp*u_j1)
                    ELSE IF (left_bc_flag == 1) THEN                       ! NonLinear Neumann
                              lbc_u = ( (kt_u(u_j1,x_left,yj,tn))*(f_left(yj,tn) - lbc2(u_j1,yj,tn)) )/lbc1(u_j1,yj,tn)
                    ELSE                                                                     ! NonLinear Robin
                              lbc_u = 0.5_rp*(1.0_rp + 2.0_rp*u_j1) &
                                        + ( (kt_u(u_j1,x_left,yj,tn))*(f_left(yj,tn) - lbc2(u_j1,yj,tn)) )/lbc1(u_j1,yj,tn)
                    END IF
          END IF

END FUNCTION lbc_u

!------------------------------------------------------------------------------------------------------------------
```

```
FUNCTION lbc_ux(u_j1,yj,tn)
IMPLICIT NONE
!           Derivative w.r.t ux, of the left BC operator, Lbc(U,x,y,t) {= Ux*Lbc1(U,x,y,t) + Lbc2(U,x,y,t) = f_left(y,t)}
!           i.e., Lbc1.
!           Function & Arguments
            REAL(KIND=rp), INTENT(IN) :: tn, u_j1, yj
            REAL(KIND=rp) :: lbc_ux

            IF (left_lin_flag == 1) THEN                    ! Linear BC
                        IF (left_bc_flag == 0) THEN
                                    lbc_ux = 0.0_rp                                    ! Linear Dirichlet
                        ELSE
                                    lbc_ux = 1.0_rp                                    ! Linear Neumann or Robin
                        END IF
            ELSE                                                        ! ANY Non-Linear BC
                        ! For the above choices of lbc1 & lbc2, this will take on the value kt.
                        lbc_ux = kt(u_j1,x_left,yj,tn)
                        IF (lbc_ux == 0.0_rp) lbc_ux = epsilon          ! lbc_ux appears in the denominator in one of the terms of LBC coeff/rhs computations.
            END IF

END FUNCTION lbc_ux

!-----------------------------------------------------------------------------------------------------------------
FUNCTION f_right(y,t)
IMPLICIT NONE
!           Define RIGHT BC - Just enter the functional representation. The type of BC (Dirichlet/Neumann/Robin) will be
!           determined from the value of the parameter "right_bc_flag" in the module CONST_PARAMS above.

!           Function & Arguments
            REAL(KIND=rp), INTENT(IN) :: t, y
            REAL(KIND=rp) :: f_right

!           Local Variables
            REAL(KIND=rp) :: et, gy, k_cond, sy, u_sol

            et = EXP(-t)
            sy = SIN(y)
            gy = (y*y/2.0_rp) + y*sy + sy*sy
            f_right = 2500000.0_rp*et*gy*( 1.0_rp - COS(x_right) )
            u_sol = 300.0_rp + 2500000.0_rp*et*gy*(x_right - SIN(x_right))
            k_cond = 1.0_rp + 14.2920_rp*(EXP(-0.0030_rp*u_sol))
            f_right = f_right*k_cond

END FUNCTION f_right

!-----------------------------------------------------------------------------------------------------------------
FUNCTION rbc1(u_jnx,yj,tn)
IMPLICIT NONE
!           First component of the right BC operator, Rbc(U,x,y,t) {= Ux*Rbc1(U,x,y,t) + Rbc2(U,x,y,t) = f_right(y,t)}
!           Function & Arguments
            REAL(KIND=rp), INTENT(IN) :: tn,u_jnx,yj
            REAL(KIND=rp) :: rbc1

            IF (right_lin_flag == 1) THEN                   ! Linear BC
                        IF (right_bc_flag == 0) THEN
                                    rbc1 = 0.0_rp                                      ! Linear Dirichlet
                        ELSE
                                    rbc1 = 1.0_rp                                      ! Linear Neumann or Robin
                        END IF
            ELSE                                                        ! Non-Linear BC
                        IF (right_bc_flag == 0) THEN
                                    rbc1 = 0.0_rp                                      ! Non-Linear Dirichlet
                        ELSE
                                    rbc1 = kt(u_jnx,x_right,yj,tn)      ! Non-Linear Neumann or Robin.
                                    IF (rbc1 == 0.0_rp) rbc1 = epsilon  ! rbc1 appears in the denominator of rbc_u for Non-Linear Neumann/Robin BCs.
                        END IF                                                             ! Can be any function of U as required by BC.
            END IF

END FUNCTION rbc1
!-----------------------------------------------------------------------------------------------------------------
FUNCTION rbc2(u_jnx,yj,tn)
IMPLICIT NONE
!           Second component of the right BC operator, Rbc(U,x,y,t) {= Ux*Rbc1(U,x,y,t) + Rbc2(U,x,y,t) = f_right(y,t)}
!           Function & Arguments
            REAL(KIND=rp), INTENT(IN) :: tn,u_jnx,yj
            REAL(KIND=rp) :: rbc2

            IF (right_lin_flag == 1) THEN                   ! Linear BC
                        IF (right_bc_flag == 0) THEN
                                    rbc2 = f_right(yj,tn)               ! Linear Dirichlet
                        ELSE        IF(right_bc_flag == 1) THEN
                                    rbc2 = 0.0_rp                                      ! Linear Neumann
                        ELSE
                                    rbc2 = alpha_x*u_jnx                ! Linear Robin
                        END IF
            ELSE                                                        ! Non-Linear BC
                        IF (right_bc_flag == 1) THEN
                                    rbc2 = 0.0_rp                                      ! Non-Linear Neumann
                        ELSE
                                    rbc2 = 0.5_rp*u_jnx*(1.0_rp + u_jnx)              ! Non-Linear Dirichlet or Robin.
                        END IF                                                                         ! Can be any function of U as required by BC.
            END IF

END FUNCTION rbc2

!-----------------------------------------------------------------------------------------------------------------
FUNCTION rbc_u(u_jnx,yj,tn)
IMPLICIT NONE
!           Derivative w.r.t U, of the ENTIRE right BC operator, Rbc(U,x,y,t) {= Ux*Rbc1(U,x,y,t) + Rbc2(U,x,y,t) = f_right(y,t)}.
!           The derivatives of the two individual components of the boundary operator (rbc1 and rbc2) are not required
!           separately by the algorithm used here.

!           Function & Arguments
            REAL(KIND=rp), INTENT(IN) :: tn,u_jnx,yj
            REAL(KIND=rp) :: rbc_u

            IF (right_lin_flag == 1) THEN                   ! Linear BC
                        IF (right_bc_flag == 0) THEN
                                    rbc_u = 1.0_rp                                     ! Linear Dirichlet
                        ELSE        IF(right_bc_flag == 1) THEN
                                    rbc_u = 0.0_rp                                     ! Linear Neumann
                        ELSE
                                    rbc_u = alpha_x                                    ! Linear Robin
                        END IF
```

```
            ELSE                                                    ! ANY Non-Linear BC
                        ! For the above choices of rbc1 & rbc2 (both = U), this will take on the value Ux + 1. Ux can be obtained from
                        ! the right boundary condition as shown below.
                        IF (right_bc_flag == 0) THEN                                        ! NonLinear Dirichlet
                                    rbc_u = 0.5_rp*(1.0_rp + 2.0_rp*u_jnx)
                        ELSE IF (right_bc_flag == 1) THEN                       ! NonLinear Neumann
                                    rbc_u = ( (kt_u(u_jnx,x_right,yj,tn))*(f_right(yj,tn) - rbc2(u_jnx,yj,tn)) )/rbc1(u_jnx,yj,tn)
                        ELSE                                                                                    ! NonLinear Robin
                            rbc_u = 0.5_rp*(1.0_rp + 2.0_rp*u_jnx) &
                                        + ( (kt_u(u_jnx,x_right,yj,tn))*(f_right(yj,tn) - rbc2(u_jnx,yj,tn)) )/rbc1(u_jnx,yj,tn)
                        END IF
            END IF

END FUNCTION rbc_u

!---------------------------------------------------------------------------------------------------------------
FUNCTION rbc_ux(u_jnx,yj,tn)
IMPLICIT NONE
!           Derivative w.r.t ux, of the right BC operator, Rbc(U,x,y,t) {= Ux*Rbc1(U,x,y,t) + Rbc2(U,x,y,t) = f_right(y,t)},
!           i.e.. Rbc1!
!           Function & Arguments
            REAL(KIND=rp), INTENT(IN) :: tn, u_jnx, yj
            REAL(KIND=rp) :: rbc_ux

            IF (right_lin_flag == 1) THEN                   ! Linear BC
                        IF (right_bc_flag == 0) THEN
                                    rbc_ux = 0.0_rp                                 ! Linear Dirichlet
                        ELSE
                                    rbc_ux = 1.0_rp                                 ! Linear Neumann or Robin
                        END IF
            ELSE                                                    ! ANY Non-Linear BC
                        ! For the above choices of rbc1 & rbc2 (both = U), this will take on the value U.
                        rbc_ux = kt(u_jnx,x_right,yj,tn)
                        IF (rbc_ux == 0.0_rp) rbc_ux = epsilon          ! rbc_ux appears in the denominator in one of the terms of RBC coeff/rhs
computations.
            END IF

END FUNCTION rbc_ux

!---------------------------------------------------------------------------------------------------------------
FUNCTION f_bottom(x,t)
IMPLICIT NONE
!           Define BOTTOM BC - Just enter the functional representation. The type of BC (Dirichlet/Neumann/Robin) will be
!           determined from the value of the parameter "bottom_bc_flag" in the module CONST_PARAMS above.

!           Function & Arguments
            REAL(KIND=rp), INTENT(IN) :: t, x
            REAL(KIND=rp) :: f_bottom

            f_bottom = 0.0_rp

END FUNCTION f_bottom

!---------------------------------------------------------------------------------------------------------------
FUNCTION bbc1(u_1i,xi,tn)
IMPLICIT NONE
!           First component of the bottom BC operator, Bbc(U,x,y,t) {= Uy*Bbc1(U,x,y,t) + Bbc2(U,x,y,t) = f_bottom(x,t)}
!           Function & Arguments
            REAL(KIND=rp), INTENT(IN) :: tn,u_1i,xi
            REAL(KIND=rp) :: bbc1

            IF (bottom_lin_flag == 1) THEN                  ! Linear BC
                        IF (bottom_bc_flag == 0) THEN
                                    bbc1 = 0.0_rp                                   ! Linear Dirichlet
                        ELSE
                                    bbc1 = 1.0_rp                                   ! Linear Neumann or Robin
                        END IF
            ELSE                                                    ! Non-Linear BC
                        IF (bottom_bc_flag == 0) THEN
                                    bbc1 = 0.0_rp                                   ! Non-Linear Dirichlet
                        ELSE
                                    bbc1 = kt(u_1i,xi,y_bottom,tn)      ! Non-Linear Neumann or Robin.
                                    IF (bbc1 == 0.0_rp) bbc1 = epsilon  ! bbc1 appears in the denominator of bbc_u for Non-Linear Neumann/Robin BCs.
                        END IF                                                      ! Can be any function of U as required by BC.
            END IF

END FUNCTION bbc1

!---------------------------------------------------------------------------------------------------------------
FUNCTION bbc2(u_1i,xi,tn)
IMPLICIT NONE
!           Second component of the bottom BC operator, Bbc(U,x,y,t) {= Uy*Bbc1(U,x,y,t) + Bbc2(U,x,y,t) = f_bottom(x,t)}
!           Function & Arguments
            REAL(KIND=rp), INTENT(IN) :: tn,u_1i,xi
            REAL(KIND=rp) :: bbc2

            IF (bottom_lin_flag == 1) THEN                  ! Linear BC
                        IF (bottom_bc_flag == 0) THEN
                                    bbc2 = f_bottom(xi,tn)              ! Linear Dirichlet
                        ELSE        IF(bottom_bc_flag == 1) THEN
                                    bbc2 = 0.0_rp                                   ! Linear Neumann
                        ELSE
                                    bbc2 = alpha_y*u_1i                 ! Linear Robin
                        END IF

            ELSE                                                    ! Non-Linear BC
                        IF (bottom_bc_flag == 1) THEN
                                    bbc2 = 0.0_rp                                   ! Non-Linear Neumann
                        ELSE
                                    bbc2 = 0.5_rp*u_1i*(1.0_rp + u_1i)  ! Non-Linear Dirichlet or Robin.
                        END IF                                                                  ! Can be any function of U as required by BC.
            END IF

END FUNCTION bbc2

!---------------------------------------------------------------------------------------------------------------
FUNCTION bbc_u(u_1i,xi,tn)
IMPLICIT NONE
!           Derivative w.r.t U, of the ENTIRE bottom BC operator, Bbc(U,x,y,t) {= Uy*Bbc1(U,x,y,t) + Bbc2(U,x,y,t) = f_bottom(x,t)}.
!           The derivatives of the two individual components of the boundary operator (bbc1 and bbc2) are not required
!           separately by the algorithm used here.
```

```
!           Function & Arguments
            REAL(KIND=rp), INTENT(IN) :: tn,u_1i,xi
            REAL(KIND=rp) :: bbc_u

            IF (bottom_lin_flag == 1) THEN                  ! Linear BC
                        IF (bottom_bc_flag == 0) THEN
                                    bbc_u = 1.0_rp                                  ! Linear Dirichlet
                        ELSE        IF(bottom_bc_flag == 1) THEN
                                    bbc_u = 0.0_rp                                  ! Linear Neumann
                        ELSE
                                    bbc_u = alpha_y                                 ! Linear Robin
                        END IF
            ELSE                                                        ! ANY Non-Linear BC
                        ! For the above choices of bbc1 & bbc2 (both = U), this will take on the value Uy + 1. Uy can be obtained from
                        ! the bottom boundary condition as shown below.
                        IF (bottom_bc_flag == 0) THEN                                           ! NonLinear Dirichlet
                                    bbc_u = 0.5_rp*(1.0_rp + 2.0_rp*u_1i)
                        ELSE IF (bottom_bc_flag == 1) THEN                      ! NonLinear Neumann
                                    bbc_u = ( (kt_u(u_1i,xi,y_bottom,tn))*(f_bottom(xi,tn) - bbc2(u_1i,xi,tn)) )/bbc1(u_1i,xi,tn)
                        ELSE                                                                            ! NonLinear Robin
                            bbc_u = 0.5_rp*(1.0_rp + 2.0_rp*u_1i) &
                                        + ( (kt_u(u_1i,xi,y_bottom,tn))*(f_bottom(xi,tn) - bbc2(u_1i,xi,tn)) )/bbc1(u_1i,xi,tn)
                        END IF
            END IF

END FUNCTION bbc_u

!-----------------------------------------------------------------------------------------------------------------
FUNCTION bbc_uy(u_1i,xi,tn)
IMPLICIT NONE
!           Derivative w.r.t Uy, of the bottom BC operator, Bbc(U,x,y,t) {= Uy*Bbc1(U,x,y,t) + Bbc2(U,x,y,t) = f_bottom(x,t)},
!           i.e., Lbc1.
!           Function & Arguments
            REAL(KIND=rp), INTENT(IN) :: tn, u_1i, xi
            REAL(KIND=rp) :: bbc_uy

            IF (bottom_lin_flag == 1) THEN                  ! Linear BC
                        IF (bottom_bc_flag == 0) THEN
                                    bbc_uy = 0.0_rp                                 ! Linear Dirichlet
                        ELSE
                                    bbc_uy = 1.0_rp                                 ! Linear Neumann or Robin
                        END IF
            ELSE                                                        ! ANY Non-Linear BC
                        ! For the above choices of bbc1 & bbc2 (both = U), this will take on the value U.
                        bbc_uy = kt(u_1i,xi,y_bottom,tn)
                        IF (bbc_uy == 0.0_rp) bbc_uy = epsilon          ! bbc_uy appears in the denominator in one of the terms of BBC coeff/rhs
computations.
            END IF

END FUNCTION bbc_uy

!-----------------------------------------------------------------------------------------------------------------
FUNCTION f_top(x,t)
IMPLICIT NONE
!           Define TOP BC - Just enter the functional representation. The type of BC (Dirichlet/Neumann/Robin) will be
!           determined from the value of the parameter "top_bc_flag" in the module CONST_PARAMS above.

!           Function & Arguments
            REAL(KIND=rp), INTENT(IN) :: t, x
            REAL(KIND=rp) :: f_top

            f_top = 0.0_rp

END FUNCTION f_top

!-----------------------------------------------------------------------------------------------------------------
FUNCTION tbc1(u_nyi,xi,tn)
IMPLICIT NONE
!           First component of the top BC operator, Tbc(U,x,y,t) {= Uy*Tbc1(U,x,y,t) + Tbc2(U,x,y,t) = f_top(x,t)}
!           Function & Arguments
            REAL(KIND=rp), INTENT(IN) :: tn,u_nyi,xi
            REAL(KIND=rp) :: tbc1

            IF (top_lin_flag == 1) THEN                     ! Linear BC
                        IF (top_bc_flag == 0) THEN
                                    tbc1 = 0.0_rp                                   ! Linear Dirichlet
                        ELSE
                                    tbc1 = 1.0_rp                                   ! Linear Neumann or Robin
                        END IF
            ELSE                                                        ! Non-Linear BC
                        IF (top_bc_flag == 0) THEN
                                    tbc1 = 0.0_rp                                   ! Non-Linear Dirichlet
                        ELSE
                                    tbc1 = kt(u_nyi,xi,y_top,tn)        ! Non-Linear Neumann or Robin.
                                    IF (tbc1 == 0.0_rp) tbc1 = epsilon  ! tbc1 appears in the denominator of tbc_u for Non-Linear Neumann/Robin BCs.
                        END IF                                                  ! Can be any function of U as required by BC.
            END IF
END FUNCTION tbc1

!-----------------------------------------------------------------------------------------------------------------
FUNCTION tbc2(u_nyi,xi,tn)
IMPLICIT NONE
!           Second component of the top BC operator, Tbc(U,x,y,t) {= Uy*Tbc1(U,x,y,t) + Tbc2(U,x,y,t) = f_top(x,t)}
!           Function & Arguments
            REAL(KIND=rp), INTENT(IN) :: tn,u_nyi,xi
            REAL(KIND=rp) :: tbc2

            IF (top_lin_flag == 1) THEN                     ! Linear BC
                        IF (top_bc_flag == 0) THEN
                                    tbc2 = f_top(xi,tn)                 ! Linear Dirichlet
                        ELSE        IF(top_bc_flag == 1) THEN
                                    tbc2 = 0.0_rp                                   ! Linear Neumann
                        ELSE
                                    tbc2 = alpha_y*u_nyi                ! Linear Robin
                        END IF
            ELSE                                                        ! Non-Linear BC
                        IF (top_bc_flag == 1) THEN
                                    tbc2 = 0.0_rp                                   ! Non-Linear Neumann
                        ELSE
                                    tbc2 = 0.5_rp*u_nyi*(1.0_rp + u_nyi)            ! Non-Linear Dirichlet or Robin.
                        END IF                                                              ! Can be any function of U as required by BC.
            END IF
END FUNCTION tbc2
!-----------------------------------------------------------------------------------------------------------------
```

```
FUNCTION tbc_u(u_nyi,xi,tn)
IMPLICIT NONE
!           Derivative w.r.t U, of the ENTIRE top BC operator, Tbc(U,x,y,t) {= Uy*Tbc1(U,x,y,t) + Tbc2(U,x,y,t) = f_top(x,t)}.
!           The derivatives of the two individual components of the boundary operator (tbc1 and tbc2) are not required
!           separately by the algorithm used here.

!           Function & Arguments
            REAL(KIND=rp), INTENT(IN) :: tn,u_nyi,xi
            REAL(KIND=rp) :: tbc_u

            IF (top_lin_flag == 1) THEN                     ! Linear BC
                        IF (top_bc_flag == 0) THEN
                                    tbc_u = 1.0_rp                                  ! Linear Dirichlet
                        ELSE        IF(top_bc_flag == 1) THEN
                                    tbc_u = 0.0_rp                                  ! Linear Neumann
                        ELSE
                                    tbc_u = alpha_y                                 ! Linear Robin
                        END IF
            ELSE                                                        ! ANY Non-Linear BC
                        ! For the above choices of tbc1 & tbc2 (both = U), this will take on the value Uy + 1. Uy can be obtained from
                        ! the top boundary condition as shown below.
                        IF (top_bc_flag == 0) THEN                                              ! NonLinear Dirichlet
                                    tbc_u = 0.5_rp*(1.0_rp + 2.0_rp*u_nyi)
                        ELSE IF (top_bc_flag == 1) THEN                         ! NonLinear Neumann
                                    tbc_u = ( (kt_u(u_nyi,xi,y_top,tn))*(f_top(xi,tn) - tbc2(u_nyi,xi,tn)) )/tbc1(u_nyi,xi,tn)
                        ELSE                                                                                                ! NonLinear Robin
                            tbc_u = 0.5_rp*(1.0_rp + 2.0_rp*u_nyi) &
                                        + ( (kt_u(u_nyi,xi,y_top,tn))*(f_top(xi,tn) - tbc2(u_nyi,xi,tn)) )/tbc1(u_nyi,xi,tn)
                        END IF
            END IF

END FUNCTION tbc_u

!-------------------------------------------------------------------------------------------------------------------
FUNCTION tbc_uy(u_nyi,xi,tn)
IMPLICIT NONE
!           Derivative w.r.t Uy, of the top BC operator, Tbc(U,x,y,t) {= Uy*Tbc1(U,x,y,t) + Tbc2(U,x,y,t) = f_top(x,t)},
!           i.e., Lbc1.
!           Function & Arguments
            REAL(KIND=rp), INTENT(IN) :: tn, u_nyi, xi
            REAL(KIND=rp) :: tbc_uy

            IF (top_lin_flag == 1) THEN                     ! Linear BC
                        IF (top_bc_flag == 0) THEN
                                    tbc_uy = 0.0_rp                                 ! Linear Dirichlet
                        ELSE
                                    tbc_uy = 1.0_rp                                 ! Linear Neumann or Robin
                        END IF
            ELSE                                                        ! ANY Non-Linear BC
                        ! For the above choices of tbc1 & tbc2 (both = U), this will take on the value U.
                        tbc_uy = kt(u_nyi,xi,y_top,tn)
                        IF (tbc_uy == 0.0_rp) tbc_uy = epsilon          ! tbc_uy appears in the denominator in one of the terms of TBC coeff/rhs computations.
            END IF

END FUNCTION tbc_uy

!-------------------------------------------------------------------------------------------------------------------
FUNCTION a1(x,y,t)
IMPLICIT NONE
!           This function computes the value of the inner coefficient of x derivatives in the adjoint form of the HEAT
!           CONDUCTION EQUATION:
!           a1*(a2*kt*U_x)_x + b1*(b2*kt*U_y)_y + f(U,x,y,t) = rho*cp*U_t, where the "_" denotes a partial derivative. For
!           the Cartesian system, a1 = 1, for the Cylindrical system, a1 = 1/x, and for Spherical the system, a1 = 1/x^2.
!           Function & Arguments
            REAL(KIND=rp), INTENT(IN) :: t, x, y
            REAL(KIND=rp) :: a1

!           If needed, a specific function a1(x,y,t) can be defined, instead of the standard forms for Cartesian, Cylindrical,
!           or Spherical coordinate systems, that are defined below, by setting coord_flag = 0 in the MODULE "const_params".

            SELECT CASE (coord_flag)
            CASE (0)
                        a1 = SIN(x - t)                     ! This can be any function a1(x,y,t).
            CASE (1)
                        a1 =  1.0_rp
            CASE (2)
                        IF (x /= 0.0_rp) THEN               ! a1 is independent of y in Cylindrcal coordinates.
                                    a1 = 1.0_rp/x
                        ELSE
                        ! This really does not matter as x=0 is the axis of cylindrical symmetry or point of spherical
                        ! symmetry. So, at x=0, the PDE itself has a different form, as determined using L'Hospital's
                        ! rule (see routine "qldgts_coeff_rhs"). This value is just assigned as a "safety trap" value
                        ! and SIMULATES the fact that in computing the coefficients at x=0 in the routine
                        ! "qldgts_coeff_rhs", a1 and a2 occur as a paired product and will cancel each other out.
                                    a1 = 1.0_rp
                        END IF
            CASE (3)
                        IF (x /= 0.0_rp) THEN               ! a1 is independent of y in Spherical coordinates.
                                    a1 = 1.0_rp/(x*x)
                        ELSE
                        ! This really does not matter as x=0 is the axis of cylindrical symmetry or point of spherical
                        ! symmetry. So, at x=0, the PDE itself has a different form, as determined using L'Hospital's
                        ! rule (see routine "qldgts_coeff_rhs"). This value is just assigned as a "safety trap" value
                        ! and SIMULATES the fact that in computing the coefficients at x=0 in the routine
                        ! "qldgts_coeff_rhs", a1 and a2 occur as a paired product and will cancel each other out.
                                    a1 = 1.0_rp
                        END IF
            CASE DEFAULT
                        PRINT *, "Coordinate Flag should be an integer from 0 to 3. Exiting program!"
                        STOP
            END SELECT

END FUNCTION a1

!-------------------------------------------------------------------------------------------------------------------
```

```
FUNCTION a2(x,y,t)
IMPLICIT NONE
!           This function computes the value of the outer coefficient of x derivatives in the adjoint form of the HEAT
!           CONDUCTION EQUATION:
!           a1*(a2*kt*U_x)_x + b1*(b2*kt*U_y)_y + f(U,x,y,t) = rho*cp*U_t, where the "_" denotes a partial derivative. For
!           the Cartesian system, a2 = 1, for the Cylindrical system, a2 = x, and for the Spherical system, a2 = x^2.
!           Function & Arguments
            REAL(KIND=rp), INTENT(IN) :: t, x, y
            REAL(KIND=rp) :: a2

!           If needed, a specific function a2(x,y,t) can be defined, instead of the standard forms for Cartesian, Cylindrical,
!           or Spherical coordinate systems, that are defined below, by setting coord_flag = 0 in the MODULE "const_params".

            SELECT CASE (coord_flag)
            CASE (0)
                        a2 = SIN(y + t)                     ! This can be any function a2(x,y,t).
            CASE (1)
                        a2 =  1.0_rp
            CASE (2)
                        IF (x /= 0.0_rp) THEN               ! a2 is independent of y in Cylindrcal coordinates.
                                    a2 = x
                        ELSE
                        ! This really does not matter as x=0 is the axis of cylindrical symmetry or point of spherical
                        ! symmetry. So, at x=0, the PDE itself has a different form, as determined using L'Hospital's
                        ! rule (see routine "qldgts_coeff_rhs"). This value is just assigned as a "safety trap" value
                        ! and SIMULATES the fact that in computing the coefficients at x=0 in the routine
                        ! "qldgts_coeff_rhs", a1 and a2 occur as a paired product and will cancel each other out.
                                    a2 = 1.0_rp
                        END IF
            CASE (3)
                        IF (x /= 0.0_rp) THEN               ! a2 is independent of y in Spherical coordinates.
                                    a2 = x*x


                        ELSE
                        ! This really does not matter as x=0 is the axis of cylindrical symmetry or point of spherical
                        ! symmetry. So, at x=0, the PDE itself has a different form, as determined using L'Hospital's
                        ! rule (see routine "qldgts_coeff_rhs"). This value is just assigned as a "safety trap" value
                        ! and SIMULATES the fact that in computing the coefficients at x=0 in the routine
                        ! "qldgts_coeff_rhs", a1 and a2 occur as a paired product and will cancel each other out.
                                    a2 = 1.0_rp
                        END IF
            END SELECT
            !           CASE DEFAULT statement is not needed here since "coord_flag" value has already been checked in the
            !           subroutine a1 above.

END FUNCTION a2

!-------------------------------------------------------------------------------------------------------------------
FUNCTION a2_x(x,y,t)
IMPLICIT NONE
!           This function computes the value of the FIRST DERIVATIVE of the outer coefficient of x derivatives in the
!           adjoint form of the HEAT CONDUCTION EQUATION:
!           a1*(a2*kt*U_x)_x + b1*(b2*kt*U_y)_y + f(U,x,y,t) = rho*cp*U_t, where the "_" denotes a partial derivative. For
!           the values of a2 defined above, the value of this function for the Cartesian system is, a2_x = 0, for the
!           Cylindrical system, a2_x = 1, and for the Spherical system, a2_x = 2x.
!           Function & Arguments
            REAL(KIND=rp), INTENT(IN) :: t, x, y
            REAL(KIND=rp) :: a2_x

!           If needed, a specific function a2(t,x,y) can be defined above, instead of the standard forms for Cartesian,
!           Cylindrical, or Spherical coordinate systems, that are defined below. In that case, its partial derivative
!           a2_x can be easily computed analytically, and specified below. This value is computed independently of whether
!           x = 0 or y = 0 because it is not needed for the spherical and cylindrical PDE functionals defined there.

            SELECT CASE (coord_flag)
            CASE (0)
                        a2_x = 0.0_rp                                   ! Based on the Function a2(x,y,t), above.
            CASE (1)
                        a2_x = 0.0_rp
            CASE (2)
                        a2_x = 1.0_rp
            CASE (3)
                        a2_x = 2.0_rp*x
            END SELECT
            !           CASE DEFAULT statement is not needed here since "coord_flag" value has already been checked in the
            !           subroutine a1 above.

END FUNCTION a2_x

!-------------------------------------------------------------------------------------------------------------------
FUNCTION b1(x,y,t)
IMPLICIT NONE
!           This function computes the value of the inner coefficient of y derivatives in the adjoint form of the HEAT
!           CONDUCTION EQUATION:
!           a1*(a2*kt*U_x)_x + b1*(b2*kt*U_y)_y + f(U,x,y,t) = rho*c*U_t, where the "_" denotes a partial derivative. For
!           the Cartesian system, b1 = 1, for the Cylindrical system, b1 = 1/x^2, and for Spherical the system,
!           b1 = 1/(x^2*SIN(y)).
!           Function & Arguments
            REAL(KIND=rp), INTENT(IN) :: t, x, y
            REAL(KIND=rp) :: b1

!           If needed, a specific function b1(x,y,t) can be defined, instead of the standard forms for Cartesian, Cylindrical,
!           or Spherical coordinate systems that are defined below, by setting coord_flag = 0 in the MODULE "const_params".

            SELECT CASE (coord_flag)
            CASE (0)
                        b1 = COS(y - t)                     ! This can be any function b1(x,y,t).
            CASE (1)
                        b1 =  1.0_rp                                    ! b1 is independent of y in Cylindrcal coordinates.
            CASE (2)
                        IF (x /= 0.0_rp) THEN
                                    b1 = 1.0_rp/(x*x)
                        ELSE
                        ! This really does not matter as x=0 is the axis of cylindrical symmetry or point of spherical
                        ! symmetry. So, at x=0, the PDE itself has a different form, as determined using L'Hospital's
                        ! rule (see routine "qldgts_coeff_rhs"). This value is just assigned as a "safety trap" value
                        ! and SIMULATES the fact that in computing the coefficients at x=0 in the routine
                        ! "qldgts_coeff_rhs", b1 and b2 occur as a paired product and will cancel each other out.
                                    b1 = 1.0_rp
                        END IF
```

```
            CASE (3)
                        IF (x /= 0.0_rp) THEN
                                    IF (y /= 0.0_rp) THEN
                                                b1 = 1.0_rp/(x*x*SIN(y))
                                    ELSE
                                                b1 = 1.0_rp/(x*x)       ! b1 depends on BOTH x & y in Spherical coordinates.
                                    END IF
                        ELSE
                        ! This really does not matter as x=0 is the axis of cylindrical symmetry or point of spherical
                        ! symmetry. So, at x=0, the PDE itself has a different form, as determined using L'Hospital's
                        ! rule (see routine "qldgts_coeff_rhs"). A similar argument applies to y=0 in the spherical
                        ! symmetry case. This value is just assigned as a "safety trap" value and SIMULATES the fact
                        ! that in computing the coefficients at y=0 in the routine "qldgts_coeff_rhs", b1 and b2 occur
                        ! as a paired product and will cancel each other out. In computing the coefficients at x=0, the
                        ! modified PDE (via. L'Hospital's rule) does not have any y dependent terms.
                                    b1 = 1.0_rp
                        END IF
            END SELECT
            !           CASE DEFAULT statement is not needed here since "coord_flag" value has already been checked in the
            !           subroutine a1 above.

END FUNCTION b1

!-------------------------------------------------------------------------------------------------------------------
FUNCTION b2(x,y,t)
IMPLICIT NONE
!           This function computes the value of the outer coefficient of y derivatives in the adjoint form of the HEAT
!           CONDUCTION EQUATION:
!           a1*(a2*kt*U_x)_x + b1*(b2*kt*U_y)_y + f(U,x,y,t) = rho*cp*U_t, where the "_" denotes a partial derivative. For
!           the Cartesian system, b2 = 1, for the Cylindrical system, b2 = 1, and for the Spherical system, b2 = SIN(y).
!           Function & Arguments
            REAL(KIND=rp), INTENT(IN) :: t, x, y
            REAL(KIND=rp) :: b2

!           If needed, a specific function b2(x,y,t) can be defined, instead of the standard forms for Cartesian, Cylindrical,
!           or Spherical coordinate systems that are defined below, by setting coord_flag = 0 in the MODULE "const_params".

            SELECT CASE (coord_flag)
            CASE (0)
                        b2 = COS(x + t)                     ! This can be any function b2(x,y,t).
            CASE (1:2)
                        b2 =  1.0_rp
            CASE (3)
                        IF (y /= 0.0_rp) THEN
                                    b2 = SIN(y)             ! b2 depends ONLY on y in Spherical coordinates.
                        ELSE
                                    ! This really does not matter as the PDE itself has a different form at y=0, as determined
                                    ! using L'Hospital's rule (see routine "qldgts_coeff_rhs"). A similar argument applies to y=0
                                    ! in the spherical symmetry case. This value is just assigned as a "safety trap" value and
                                    ! SIMULATES the fact that in computing the coefficients at y=0 in the routine "qldgts_coeff_rhs",
                                    ! b1 and b2 occur as a paired product and will cancel each other out. In computing the
                                    ! coefficients at x=0, the modified PDE (via. L'Hospital's rule) does not have any y dependent
                                    ! terms. Even when x /= 0, b1*b2 = 1/(x*x). So, b2=1 works fine for this system of coordinates.
                                    b2 = 1.0_rp
                        END IF
            END SELECT
            !           CASE DEFAULT statement is not needed here since "coord_flag" value has already been checked in the
            !           subroutine a1 above.

END FUNCTION b2

!-------------------------------------------------------------------------------------------------------------------
FUNCTION b2_y(x,y,t)
IMPLICIT NONE
!           This function computes the value of the FIRST DERIVATIVE of the outer coefficient of x derivatives in the
!           adjoint form of the HEAT CONDUCTION EQUATION:
!           a1*(a2*kt*U_x)_x + b1*(b2*kt*U_y)_y + f(U,x,y,t) = rho*cp*U_t, where the "_" denotes a partial derivative. For
!           the values of b2 defined above, the value of this function for the Cartesian system is, b2_y = 0, for the
!           Cylindrical system, b2_y = 0, and for the Spherical system, b2_y = COS(y).
!           Function & Arguments
            REAL(KIND=rp), INTENT(IN) :: t, x, y
            REAL(KIND=rp) :: b2_y

!           If needed, a specific function b2(x,y,t) can be defined above, instead of the standard forms for Cartesian,
!           Cylindrical, or Spherical coordinate systems, that are defined below. In that case, its partial derivative
!           b2_y can be easily computed analytically, and specified below. This value is computed independently of whether
!           x = 0 or y = 0 because it is not needed for the spherical and cylindrical PDE functionals defined there.

            SELECT CASE (coord_flag)
            CASE (0)
                        b2_y = 0.0_rp                       ! Based on Function b2(x,y,t) above.
            CASE (1:2)
                        b2_y = 0.0_rp
            CASE (3)
                        b2_y = COS(y)
            END SELECT
            !           CASE DEFAULT statement is not needed here since "coord_flag" value has already been checked in the
            !           subroutine a1 above.

END FUNCTION b2_y

!-------------------------------------------------------------------------------------------------------------------
FUNCTION u_x(j,i,uj,x,y,tn)
IMPLICIT NONE
!           This function computes the value of the FIRST PARTIAL DERIVATIVE of u w.r.t x, for computing the
!           Frechet-Taylor Coefficients of the Linearlized form of the original adjoint form PDE:
!           a1*(a2*kt*U_x)_x + b1*(b2*kt*U_y)_y + f(U,x,y,t) = rho*cp*U_t, where the "_" denotes a partial derivative.
!           Function & Arguments
            REAL(KIND=rp), DIMENSION(:), INTENT(IN) :: uj,x,y           ! Only the corresponding row is needed.
            REAL(KIND=rp), INTENT(IN) :: tn
            INTEGER(KIND=ip), INTENT(IN) :: i,j
            REAL(KIND=rp) :: u_x

            !           Local Variables
            REAL(KIND=rp) :: hx
            INTEGER(KIND=ip) :: nx



            !           Main Calculations.
            nx = SIZE(x)
            hx = (x(nx) - x(1))/REAL(nx - 1)
            IF (i>1 .AND. i<nx ) THEN           ! Interior point (including the TOP/BOTTOM boundary). Use Centered Differencing.
                        u_x = (uj(i+1) - uj(i-1))/(2.0_rp*hx)
```

```fortran
            ELSE IF (i==1) THEN                 ! Left boundary point.
                          ! If Left BC is non-"Dirichlet" (see notes in MODULE "const_params"), then use BC to compute u_x
                          ! Otherwise, the assignment of the left Dirichlet BC to the grid function obviates the need for u_x.
                          IF (left_bc_flag /= 0) THEN
                                    u_x = ( f_left(y(j),tn) - lbc2(uj(1),y(j),tn) )/lbc1(uj(1),y(j),tn)
                          END IF
            ELSE
                          ! If Right BC is non-"Dirichlet" (see notes in MODULE "const_params"), then use BC to compute u_x
                          ! Otherwise, the assignment of the right Dirichlet BC to the grid function obviates the need for u_x.
                          IF (right_bc_flag /= 0) THEN
                                    u_x = ( f_right(y(j),tn) - rbc2(uj(nx),y(j),tn) )/rbc1(uj(nx),y(j),tn)
                          END IF
            END IF

END FUNCTION u_x

!-------------------------------------------------------------------------------------------------------------------
FUNCTION u_y(j,i,ui,x,y,tn)
IMPLICIT NONE
!           This function computes the value of the FIRST PARTIAL DERIVATIVE of u w.r.t y, for computing the
!           Frechet-Taylor Coefficients of the Linearlized form of the original adjoint form PDE:
!           a1*(a2*kt*U_x)_x + b1*(b2*kt*U_y)_y + f(U,x,y,t) = rho*cp*U_t, where the "_" denotes a partial derivative.
!           Function & Arguments
            REAL(KIND=rp), DIMENSION(:), INTENT(IN) :: ui,x,y           ! Only the corresponding column is needed.
            REAL(KIND=rp), INTENT(IN) :: tn
            INTEGER(KIND=ip), INTENT(IN) :: i,j
            REAL(KIND=rp) :: u_y

            !           Local Variables
            REAL(KIND=rp) :: hy
            INTEGER(KIND=ip) :: ny

            !           Main Calculations.
            ny = SIZE(y)
            hy = (y(ny) - y(1))/REAL(ny - 1)
            IF (j>1 .AND. j<ny ) THEN           ! Interior point (including the LEFT/RIGHT boundary). Use Centered Differencing.
                        u_y = (ui(j+1) - ui(j-1))/(2.0_rp*hy)
            ELSE IF (j==1) THEN                 ! Left boundary point.
                          ! If Bottom BC is non-"Dirichlet" (see notes in MODULE "const_params"), then use BC to compute u_y
                          ! Otherwise, the assignment of the bottom Dirichlet BC to the grid function obviates the need for u_y.
                          IF (bottom_bc_flag /= 0) THEN
                                    u_y = ( f_bottom(x(i),tn) - bbc2(ui(1),x(i),tn) )/bbc1(ui(1),x(i),tn)
                          END IF
            ELSE
                          ! If Top BC is non-"Dirichlet" (see notes in MODULE "const_params"), then use BC to compute u_y
                          ! Otherwise, the assignment of the top Dirichlet BC to the grid function obviates the need for u_y.
                          IF (top_bc_flag /= 0) THEN
                                    u_y = ( f_top(x(i),tn) - tbc2(ui(ny),x(i),tn) )/tbc1(ui(ny),x(i),tn)
                          END IF
            END IF

END FUNCTION u_y

!-------------------------------------------------------------------------------------------------------------------
FUNCTION u_xx(j,i,u,x,y,tn)
IMPLICIT NONE
!           This function computes the value of the SECOND PARTIAL DERIVATIVE of u w.r.t x, for computing the
!           Frechet-Taylor Coefficients of the Linearlized form of the original adjoint form PDE:
!           a1*(a2*kt*U_x)_x + b1*(b2*kt*U_y)_y + f(U,x,y,t) = rho*cp*U_t, where the "_" denotes a partial derivative.
!           Function & Arguments
            REAL(KIND=rp), DIMENSION(:,:), INTENT(IN) :: u
            REAL(KIND=rp), DIMENSION(:), INTENT(IN) :: x,y
            REAL(KIND=rp), INTENT(IN) :: tn
            INTEGER(KIND=ip), INTENT(IN) :: i,j
            REAL(KIND=rp) :: u_xx

            !           Local Variables
            REAL(KIND=rp) :: hx, a
            INTEGER(KIND=ip) :: nx

            !           Main Calculations.
            nx = SIZE(x)
            hx = (x(nx) - x(1))/REAL(nx - 1)
            IF (i>1 .AND. i<nx ) THEN
                        ! Interior point (including the TOP/BOTTOM boundary). Use 2nd order Centered Differencing.
                        u_xx = (u(j,i-1) - 2.0_rp*u(j,i) + u(j,i+1))/(hx*hx)
            ELSE IF (i==1) THEN                 ! Left boundary point.
                          ! If Left BC is Dirichlet, then u_xx values are not needed for any calculations due to the
                          ! assignment of the left Dirichlet BC to the grid function values at this boundary. Otherwise,
                          ! use the boundary value to compute the image element (corresponding to the 0th column), and
                          ! thus compute the centered difference estimation.
                          IF (left_bc_flag /= 0) THEN
                                    a = ( f_left(y(j),tn) - lbc2(u(j,1),y(j),tn) )/lbc1(u(j,1),y(j),tn)
                                    u_xx = (2.0_rp/(hx*hx))*(u(j,2) - u(j,1) - hx*a)
                          END IF
            ELSE
                          ! If Right BC is Dirichlet, then u_xx values are not needed for any calculations due to the
                          ! assignment of the right Dirichlet BC to the grid function values at this boundary. Otherwise,
                          ! use the boundary value to compute the image element (corresponding to the Nx + 1st column),
                          ! and thus compute the centered difference estimation.
                          IF (right_bc_flag /= 0) THEN
                                    a = ( f_right(y(j),tn) - rbc2(u(j,nx),y(j),tn) )/rbc1(u(j,nx),y(j),tn)
                                    u_xx = (2.0_rp/(hx*hx))*(u(j,nx-1) - u(j,nx) + hx*a)
                          END IF
            END IF
END FUNCTION u_xx

!-------------------------------------------------------------------------------------------------------------------
FUNCTION u_yy(j,i,u,x,y,tn)
IMPLICIT NONE
!           This function computes the value of the SECOND PARTIAL DERIVATIVE of u w.r.t y, for computing the
!           Frechet-Taylor Coefficients of the Linearlized form of the original adjoint form PDE:
!           a1*(a2*kt*U_x)_x + b1*(b2*kt*U_y)_y + f(U,x,y,t) = rho*cp*U_t, where the "_" denotes a partial derivative.
!           Function & Arguments
            REAL(KIND=rp), DIMENSION(:,:), INTENT(IN) :: u
            REAL(KIND=rp), DIMENSION(:), INTENT(IN) :: x,y
            REAL(KIND=rp), INTENT(IN) :: tn
            INTEGER(KIND=ip), INTENT(IN) :: i,j
            REAL(KIND=rp) :: u_yy

            !           Local Variables
            REAL(KIND=rp) :: hy, a
            INTEGER(KIND=ip) :: ny
```

```
		!		Main Calculations.
		ny = SIZE(y)
		hy = (y(ny) - y(1))/REAL(ny - 1)
		IF (j>1 .AND. j<ny ) THEN
			! Interior point (including the TOP/BOTTOM boundary). Use 2nd order Centered Differencing.
			u_yy = (u(j-1,i) - 2.0_rp*u(j,i) + u(j+1,i))/(hy*hy)
		ELSE IF (j==1) THEN			! Left boundary point.
			! If Bottom BC is Dirichlet, then u_yy values are not needed for any calculations due to the
			! assignment of the bottom Dirichlet BC to the grid function values at this boundary. Otherwise,
			! use the boundary value to compute the image element (corresponding to the 0th row), and thus
			! compute the centereddifference estimation.
			IF (bottom_bc_flag /= 0) THEN
				a = ( f_bottom(x(i),tn) - bbc2(u(1,i),x(i),tn) )/bbc1(u(1,i),x(i),tn)
				u_yy = (2.0_rp/(hy*hy))*(u(2,i) - u(1,i) - hy*a)
			END IF
		ELSE
			! If Top BC is Dirichlet, then u_yy values are not needed for any calculations due to the
			! assignment of the top Dirichlet BC to the grid function values at this boundary. Otherwise,
			! use the boundary value to compute the image element (corresponding to the Ny + 1st row), and
			! thus compute the centereddifference estimation.
			IF (top_bc_flag /=0) THEN
				a = ( f_top(x(i),tn) - tbc2(u(ny,i),x(i),tn) )/tbc1(u(ny,i),x(i),tn)
				u_yy = (2.0_rp/(hy*hy))*(u(ny-1,i) - u(ny,i) + hy*a)
			END IF
		END IF

	END FUNCTION u_yy
	!------------------------------------------------------------------------------------------------------------

END MODULE pde_routines

!------------------------------------------------------------------------------------------------------------------------
MODULE solver_routines
USE const_params
USE fault_params
USE pde_routines

	CONTAINS

	!------------------------------------------------------------------------------------------------------------
	SUBROUTINE lud_trid(a, b)
	IMPLICIT NONE
	!------------------------------------------------------------------------------------------------------------
	!		A subroutine that solves a TRI-DIAGONAL system of linear equations (ANY NUMBER, upto MACHINE MEMORY LIMIT)
	!		Ax = b: where A is a "compressed" tri-diagonal matrix, of dimension n X 3, and b is a vector of dimension n.
	!		This routine gets matrices A, and b as INPUTS. It RETURNS the solution in vector b. This algorithm uses the
	!		space allocated for the A matrix to simultaneously store the elements of the lower (L) and upper (U)
	!		triangular matrices into which A is decomposed. It does this by not storing or using the diagonal elements
	!		of U, which are all equal to 1.
	!------------------------------------------------------------------------------------------------------------

	!		Function & Arguments
		REAL(KIND=rp), DIMENSION(:,:), INTENT(INOUT) :: a
		REAL(KIND=rp), DIMENSION(:), INTENT(INOUT) :: b

	!		Local variables
		INTEGER(KIND=ip) :: i, num_steps

	!		Input checks for argument consistency.
		IF (SIZE(a,1) == 0) THEN
			PRINT*, "ERROR: Input array A should be a square matrix with AT LEAST ONE element."
			STOP
		END IF
		IF (SIZE(a,1) /= SIZE(b)) THEN
			PRINT*, "ERROR: Input array dimensions for A and b do not match. Please check your inputs."
			STOP
		END IF

	!		Main Calculations.
		num_steps = SIZE(a,1)

	!		Compute the elements of L and U, within the three columns of A. The sub-diagonal elements of L are in the
	!		first column of A, the diagonal elements of L are in the second column of A, and the super-diagonal elements
	!		of U are in the third column of A. The diagonal elements of U are all equal to 1 and are neither stored, nor
	!		explicitly used. The first column of A are identical to the sub-diagonal elements of L. So, only the second
	!		and third columns of A need be explicitly computed. Also, the first element of the second column is identical
	!		to the first diagonal element of L. Also, the first element of the first column of A as well as the last
	!		element of its third (last) column, are both ZERO.

		DO i = 1, num_steps
			IF (i > 1) a(i,2) = a(i,2) - a(i,1)*a(i-1, 3)
			IF (ABS(a(i,2)) < epsilon) THEN
				PRINT *, "ERROR: Coefficient of the diagonal element corresponding to ROW# ", i, " is very small."
				PRINT *, "This Tridiagonal algorithm cannot handle TINY or ZERO diagonal elements. EXITING PROGRAM!"
				STOP
			END IF
			IF (i < num_steps) a(i,3) = a(i, 3)/a(i,2)
		END DO

		!		Forward Substitution Step - Solving the system Ly = b, where, y = Ux. The vector y is stored in b:
			DO i = 1, num_steps
				IF (i == 1) THEN
					b(i) = b(i)/a(i,2)
				ELSE
					b(i) = (b(i) - a(i, 1)*b(i-1))/a(i,2)
				END IF
			END DO

		!		Backward Substitution - Ux = b. The RHS vector (b=y) is REPLACED by the solution vector, x:
			DO i = num_steps-1, 1, -1
				b(i) = b(i) - a(i,3)*b(i+1)
			END DO

	END SUBROUTINE lud_trid

	!------------------------------------------------------------------------------------------------------------
```

```
SUBROUTINE qldgts_coeff_rhs(x, y, t, k, stage_flag, iter, u, u_m, residual)
IMPLICIT NONE
!-----------------------------------------------------------------------------------------------------------------
!           This function computes a predefined coefficient matrix "coeff", and the "rhs" vector that are needed to
!           construct the tri-diagonal system at each each of the time split stages of the DOUGLAS-GUNN TIME SPLITTING
!           algorithm, applied to the PDE:
!           Ut = {1/(rho*cp)}*
!           [a1*{kt*(a2_x*U_x + a2*U_xx) + a2*kt_u*(U_x)^2} + b1*{kt*(b2_y*U_y + b2*U_yy) + b2*kt_u*(U_y)^2} + f(U,x,y,t)],
!           where the "_" denotes partial differentiation, obtained by expanding the ADJOINT form of the linear, but very
!           general Pure Conduction Equation. The values of functions a1, a2, b1, b2, kt(U) and cp(U) can be changed to
!           match any regular, closed domain. THIS ROUTINE ACCOUNTS FOR ALL 4 BOUNDARY CONDITIONS, OF ANY TYPE
!           (LINEAR/NON-LINEAR - Dirichlet/Neumann/Robin). Appropriate boundary condition flags must be set in the module
!           "const_params" above, and  boundary condition values are computed using about 30 different functions that preceed
!           this subroutine. Depending on the value of the stage_flag, either the first or the second stage arrays are
!           constructed, as follows:
!           {coeff_1(n+1)}*{U*(n+1)} = {rhs_1(n)}  &
!           {coeff_2(n+1)}*{U(n+1)} = U*(n+1)
!           where n denotes the time step and "coeff_i" are tridiagonal matrices of dimension nx*ny (=n).
!           Only the band diagonal elements of the tridiagonal systems are computed & stored in this program, to minimize
!           storage. They are stored in the form of n X 3 matrices, where the three columns are, respectively, the
!           sub-diagonal, diagonal, and super-diagonal elements of the original n X n system matrix.
!
!           NOTE: THIS SAME ROUTINE CAN BE USED FOR LINEAR OR NON-LINEAR PDE (WITH LINEAR/NON-LINEAR BCs). It can be shown
!           that the same functional expression applies to BOTH the linear and non-linear cases of the generalized PDE being
!           solved here (See documentation for all proofs/derivations). While linear cases are automatically accounted for by
!           the subroutines in the MODULE "pde_routines", the specific non-linear functional components have to be defined
!           for each problem, as required. This means that the LINEAR CASE CAN BE TREATED AS A SPECIAL CASE OF THE NON-LINEAR
!           CASE, and one compact notation can be used throughout.
!
!-----------------------------------------------------------------------------------------------------------------

            !           Argument Variable Declarations
            REAL(KIND=rp), DIMENSION(:), INTENT(IN) :: x, y
            REAL(KIND=rp), INTENT(IN) :: k, t
            REAL(KIND=rp), DIMENSION(:,:), INTENT(IN) :: u, u_m
            INTEGER(KIND=ip), INTENT(IN) :: iter, stage_flag
            REAL(KIND=rp), DIMENSION(:), INTENT(OUT), OPTIONAL :: residual

            !           Local Variable Declarations. NOTE variable Nu_m is defined Globally under the MODULE "const_params".
            REAL(KIND=rp) :: hx, hy, B_m, Bu_m, Buy_m, L_m, Lu_m, Lux_m, n1, n2, n3, n4, n5, n6, n7, n8, n9, n10, N0_m, N0_n,&
                             & N0u_m, N0ux_m, N0uxx_m, N0uxx_n, N_n, N_m, NS_m, NS_n, NSux_m, NSuxx_m, NSuy_m, NSuyy_m, Nux_m, &
                             & Nuy_m, Nuxx_m, Nuyy_m, rb, R_m, rt, Ru_m, Rux_m, rx, ry, T_m, Tu_m, Tuy_m, t_np1, t_n, ux, uy
            REAL(KIND=rp), DIMENSION(SIZE(y), SIZE(x)) :: r, r1
            REAL(KIND=rp), DIMENSION(SIZE(y), SIZE(x), 3) :: cf
            REAL(KIND=rp), DIMENSION(SIZE(x)) :: uj, uj_n
            INTEGER(KIND=ip) :: alloc_error, dealloc_error, i, i_end, i_start, j, j_end, j_start, l, m, n, nt, nx, ny

            !           Main Calculations.
            nx = SIZE(x)
            ny = SIZE(y)
            hx = (x(nx) - x(1))/REAL(nx - 1)
            hy = (y(ny) - y(1))/REAL(ny - 1)
            n  = nx*ny

            t_np1 = t
            t_n   = t - k
            rx = k/(2.0_rp*hx*hx)
            ry = k/(2.0_rp*hy*hy)

            !           NOTE: For BOTH stages , LHS is computed at the previous iteration (m), and current time level n.
            !           For stage 1, RHS is computed at both n & n-1 levels. For stage 2, the RHS depends on the intermediate
            !           "solution" at the end of stage 1, dv(1), and the previous iterate, U_m, and current time level n.
            !           Values at different time levels are computed separately.

            !           If the PDE in question is NON-LINEAR, first compute the derivative of the NON-LINEAR FUNCTIONAL w.r.t u,
            !           Nu_m, which was defined GLOBALLY in the MODULE "const_params". This is used in both time stages of this
            !           routine, and saving this cuts down a considerable amount of arithmetic. In the LINEAR case, this
            !           becomes 0 (ZERO) identically, since kt and cp are CONSTANTS and f_rhs is independent of U, as defined
            !           in the MODULE pde_routines above, and all their derivatives w.r.t. U are zero.

            !           STEP 1: GRID INTERIOR - Computing LHS Coefficients and RHS values for all grid points (INTERIOR for NON-Cartesian):
            !           -----------------------------------------------------------------------------------------------------------
            !           (1a) COMPUTING THE TRIDIAGONAL COEFFICIENT MATRIX FOR BOTH STAGES, "COEFF":
            !                       ALL the coefficients will be first computed as a 3-D array cf(j,i,3), i.e., three coefficients
            !           for each grid node, for clarity, and to minimize any calculation errors. This array will then be
            !           converted to the 2D "coeff" array (Dimension: N x 3 = Nx*Ny x 3), after all BCs have been accounted for.
            !           (1b) RESIDUAL:
            !                       THE ARRAY "r1" STORES THE RESIDUAL VECTOR AT EACH ITERATION.
            !           (1c) COMPUTING THE RHS VECTOR, "rhs", FOR THE FIRST STAGE:
            !                       ALL the RHS coefficients will be computed as a 2-D array r(j,i), for clarity and to minimize any
            !           notational errors. This array will later be converted to the 1-D "rhs" vector, after all BCs have been
            !           accounted for.

            SELECT CASE (coord_flag)
            CASE (0)
            ! For a User Defined System, ASSUME ALL COEFFICIENTS ARE ANALYTIC IN THE PROBLEM DOMAIN. Hence, compute BOTH Interior & Boundary
            ! points for Neumann or Robin BCs, as in the Cartesian system.
                        IF (left_bc_flag == 0) THEN
                                    i_start = 2
                        ELSE
                                    i_start = 1
                        END IF
                        IF (right_bc_flag == 0) THEN
                                    i_end = nx-1
                        ELSE
                                    i_end = nx
                        END IF
                        IF (bottom_bc_flag == 0) THEN
                                    j_start = 2
                        ELSE
                                    j_start = 1
                        END IF
                        IF (top_bc_flag == 0) THEN
                                    j_end = ny-1
                        ELSE
                                    j_end = ny
                        END IF
            CASE (1)    ! For Cartesian System, Compute BOTH Interior & Boundary points for Neumann or Robin BCs.
                        IF (left_bc_flag == 0) THEN
                                    i_start = 2
                        ELSE
                                    i_start = 1
                        END IF
```

```
            IF (right_bc_flag == 0) THEN
                        i_end = nx-1
            ELSE
                        i_end = nx
            END IF
            IF (bottom_bc_flag == 0) THEN
                        j_start = 2
            ELSE
                        j_start = 1
            END IF
            IF (top_bc_flag == 0) THEN
                        j_end = ny-1
            ELSE
                        j_end = ny
            END IF
CASE (2)    ! For Cylindrical System, Compute Interior & Boundary Points, EXCEPT LEFT x-BC, for Neumann or Robin BCs.
            i_start = 2
            IF (right_bc_flag == 0) THEN
                        i_end = nx-1
            ELSE
                        i_end = nx
            END IF
            IF (bottom_bc_flag == 0) THEN
                        j_start = 2
            ELSE
                        j_start = 1
            END IF
            IF (top_bc_flag == 0) THEN
                        j_end = ny-1
            ELSE
                        j_end = ny
            END IF
CASE (3)    ! For Spherical System, Compute Interior & only MIDDLE PORTION OF RIGHT x-BC, for Neumann or Robin BCs.
            i_start = 2
            IF (right_bc_flag == 0) THEN
                        i_end = nx-1
            ELSE
                        i_end = nx
            END IF
            j_start = 2
            j_end = ny-1
END SELECT
IF (stage_flag == 1) THEN           ! FIRST STAGE.
            !           FOR ALL GRID POINTS: Compute Nu_m and store it for use in the second stage of this iteration.
            !           Compute N_m, Nux_m, Nuxx_m, and N_n; Then compute the LHS Coefficient array, the residual array
            !           for the current iteration, AND the RHS vector for the FIRST stage.
            DO j = j_start, j_end
                        DO m = 1, nx
                                      uj(m) = u_m(j,m)
                                    uj_n(m) =   u(j,m)
                        END DO
                        DO i = i_start, i_end
                                    ux = u_x(j,i,uj,x,y,t_np1)
                                    uy = u_y(j,i,u_m(:,i),x,y,t_np1)
                                    n1 = (kt_u(u_m(j,i)))*cp(u_m(j,i)) - (cp_u(u_m(j,i)))*kt(u_m(j,i))
                                    n2 = (a1(x(i), y(j), t_np1))*(a2_x(x(i), y(j), t_np1))*ux
                                    n3 = (a1(x(i), y(j), t_np1))*(  a2(x(i), y(j), t_np1))*u_xx(j,i,u_m,x,y,t_np1)
                                    n4 = (b1(x(i), y(j), t_np1))*(b2_y(x(i), y(j), t_np1))*uy
                                    n5 = (b1(x(i), y(j), t_np1))*(  b2(x(i), y(j), t_np1))*u_yy(j,i,u_m,x,y,t_np1)
                                    n6 = (kt_uu(u_m(j,i)))*cp(u_m(j,i)) - (cp_u(u_m(j,i)))*kt_u(u_m(j,i))
                                    n7 = (a1(x(i), y(j), t_np1))*(a2(x(i), y(j), t_np1))*ux*ux
                                    n8 = (b1(x(i), y(j), t_np1))*(b2(x(i), y(j), t_np1))*uy*uy
                                    n9 = (f_rhs_u(u_m(j,i), x(i), y(j), t_np1))*  cp(u_m(j,i)) &
                                      &  - (f_rhs(u_m(j,i), x(i), y(j), t_np1))*cp_u(u_m(j,i))
                                    Nu_m(j,i) = (n1*(n2 + n3 + n4 + n5) + n6*(n7 + n8) + n9)/( rho*(cp(u_m(j,i)))*cp(u_m(j,i)) )

                                    n1 = kt(u_m(j,i))
                                    n6 = kt_u(u_m(j,i))
                                    n9 = f_rhs(u_m(j,i), x(i), y(j), t_np1)
                                    N_m = (n1*(n2 + n3 + n4 + n5) + n6*(n7 + n8) + n9)/( rho*cp(u_m(j,i)) ) ! Use n2-n5,n7,n8 from Nu_m.

                                    n4 = (a1(x(i), y(j), t_np1))/( rho*cp(u_m(j,i)) )
                                    n2 = n1*a2_x(x(i), y(j), t_np1)                               ! Use n1 from N_m calculation.
                                    n3 = 2.0_rp*(a2(x(i), y(j), t_np1))*n6*ux       ! Use n6 from N_m calculation.
                                    Nux_m = n4*(n2+n3)

                                    Nuxx_m = n4*(a2(x(i), y(j), t_np1))*n1          ! Use n4 from Nux_m calculation, n1 from N_m calculation.

                                    n2 = hx*Nux_m/(2.0_rp*Nuxx_m)
                                    n3 = ( 4.0_rp - k*Nu_m(j,i) )/(4.0_rp*rx*Nuxx_m)
                                    cf(j,i,1) = 1.0_rp - n2                                           ! First LHS Coefficient.
                                    cf(j,i,2) = -(2.0_rp + n3)                                        ! Second LHS Coefficient.
                                    cf(j,i,3) = 1.0_rp + n2                                           ! Third LHS Coefficient.

                                    ux = u_x(j,i,uj_n,x,y,t_n)
                                    uy = u_y(j,i,u(:,i),x,y,t_n)
                                    n1 = kt(u(j,i))
                                    n2 = (a1(x(i), y(j), t_n))*(a2_x(x(i), y(j), t_n))*ux
                                    n3 = (a1(x(i), y(j), t_n))*(  a2(x(i), y(j), t_n))*u_xx(j,i,u,x,y,t_n)
                                    n4 = (b1(x(i), y(j), t_n))*(b2_y(x(i), y(j), t_n))*uy
                                    n5 = (b1(x(i), y(j), t_n))*(  b2(x(i), y(j), t_n))*u_yy(j,i,u,x,y,t_n)
                                    n6 = kt_u(u(j,i))
                                    n7 = (a1(x(i), y(j), t_n))*(  a2(x(i), y(j), t_n))*ux*ux
                                    n8 = (b1(x(i), y(j), t_n))*(  b2(x(i), y(j), t_n))*uy*uy
                                    n9 = f_rhs(u_m(j,i), x(i), y(j), t_n)
                                    N_n = (n1*(n2 + n3 + n4 + n5) + n6*(n7 + n8) + n9)/( rho*cp(u(j,i)) )

                                    IF (linear_flag == 1) THEN          ! Linear PDE. FIRST STAGE RHS VECTOR.
                                                r(j,i) = -( u(j,i) - u_m(j,i) + 0.5_rp*k*(N_m + N_n) )/(rx*Nuxx_m)

                                    ELSE                                            ! Non-Linear PDE.
                                                r1(j,i) =   u(j,i) - u_m(j,i) + 0.5_rp*k*(N_m + N_n)        ! NON-LINEAR RESIDUAL.
                                                r(j,i) = -r1(j,i)/(rx*Nuxx_m)                               ! FIRST STAGE RHS VECTOR.
                                    END IF
                        END DO
            END DO
ELSE                                    ! SECOND STAGE.
            DO j = j_start, j_end
                        DO i = i_start, i_end
                                    n1     = (b1(x(i), y(j), t_np1))/( rho*cp(u_m(j,i)) )
                                    n2     = (kt(u_m(j,i)))*b2_y(x(i), y(j), t_np1)
                                    n3     = 2.0_rp*(b2(x(i), y(j), t_np1))*(kt_u(u_m(j,i)))*u_y(j,i,u_m(:,i),x,y,t_np1)
                                    Nuy_m  = n1*(n2+n3)
```

```fortran
                        Nuyy_m = n1*(b2(x(i), y(j), t_np1))*(kt(u_m(j,i)))          ! Use n1 from Nuy_m calculation.

                        n4 = hy*Nuy_m/(2.0_rp*Nuyy_m)
                        n5 = ( 4.0_rp - k*Nu_m(j,i) )/(4.0_rp*ry*Nuyy_m)
                        cf(j,i,1) = 1.0_rp - n4                                      ! First LHS Coefficient.
                        cf(j,i,2) = -(2.0_rp + n5)                                   ! Second LHS Coefficient.
                        cf(j,i,3) = 1.0_rp + n4                                      ! Third LHS Coefficient.

                        r(j,i) = -u(j,i)/(ry*Nuyy_m)                                 ! SECOND STAGE RHS VECTOR.
                END DO
        END DO
END IF

!       STEP 2: GRID BOUNDARIES - Compute the Coeff & RHS array values for BOUNDARY grid points:
!       ------------------------------------------------------------------------------------
!       This step is carried out for any combination of GENERAL (i.e., Linear/Non-Linear) Dirichlet/ Neumann/ Robin BCs.
!       This is determined from the BC flags in the module "const_params". Also, at this stage, corner points are
!       adjusted based on type of BCs along intersecting boundaries.

!       STEP 2(a): LEFT BOUNDARY (i = 1) & LEFT CORNER POINTS (i = 1, WITH j = 1 OR ny).
!       ----------------------------------------------------------------------------
!       The generalized BC is given by: L_m(U) = Ux*L1(U) + L2(U) = f_left(y,t(m)). Therefore, Lu_m = Ux*L1u + L2u;
!       and Lux_m = L1. Also, for a generalized Dirichlet BC, L1 = 0 => L1u = 0. So, in this case, L_m = L2,
!       Lu_m = L2u, and Lux_m = 0. For the linear case, L_m = f_left(y,t(n)), Lu_m = alpha_x (= 0 for linear
!       Neumann BC), and Lux_m = 1. In terms of components, for linear Neumann, L1_m = 1, L2_m = 0; for linear
!       Robin BC, L1_m = 1, L2_m = alpha_x*U_n. All these values are taken care of, in the module "pde_routines",
!       above, where separate subroutines are defined for each component of the left BC.

IF (left_bc_flag == 0) THEN                             ! GENERAL LINEAR/NON-LINEAR Dirichlet Left BC.
        DO j = 1, ny
                cf(j,1,1) = 0.0_rp
                cf(j,1,2) = 1.0_rp
                cf(j,1,3) = 0.0_rp
                IF (linear_flag == 1) THEN
                        L_m  =  lbc2(u_m(j,1),y(j),t_n)
                        Lu_m = lbc_u(u_m(j,1),y(j),t_n)
                ELSE
                        IF (iter == 1) THEN
                                L_m  =  lbc2(u_m(j,1),y(j),t_n)
                                Lu_m = lbc_u(u_m(j,1),y(j),t_n)
                        ELSE
                                L_m  =  lbc2(u_m(j,1),y(j),t_np1)
                                Lu_m = lbc_u(u_m(j,1),y(j),t_np1)
                        END IF
                END IF
                r(j,1) = (f_left(y(j),t_np1) - L_m )/Lu_m       ! RHS value if Left BC is Linear/Non-Linear Dirichlet.
                IF (stage_flag == 1) THEN
                        IF (linear_flag /= 1) r1(j,1) = 0.0_rp          ! Calculate residual for the 1st stage of a Non-Linear PDE.
                END IF
        END DO
        IF (bottom_bc_flag == 0) THEN
                IF (linear_flag == 1) THEN
                        B_m =   bbc2(u_m(1,1),x(1),t_n)
                        Bu_m = bbc_u(u_m(1,1),x(1),t_n)
                ELSE
                        IF (iter == 1) THEN
                                B_m =   bbc2(u_m(1,1),x(1),t_n)
                                Bu_m = bbc_u(u_m(1,1),x(1),t_n)
                        ELSE
                                B_m =   bbc2(u_m(1,1),x(1),t_np1)
                                Bu_m = bbc_u(u_m(1,1),x(1),t_np1)
                        END IF
                END IF
                rb = (f_bottom(x(1),t_np1) - B_m )/Bu_m
                r(1,1)= 0.5_rp*(r(1,1)+ rb)                             ! BOTH Left & Bottom BCs are General Dirichlet.
                IF (stage_flag == 1) THEN
                        IF (linear_flag /= 1) r1(1,1) = 0.0_rp          ! Calculate residual for the 1st stage of a Non-Linear PDE.
                END IF
        END IF
        IF (top_bc_flag == 0) THEN
                IF (linear_flag == 1) THEN
                        T_m =   tbc2(u_m(ny,1),x(1),t_n)
                        Tu_m = tbc_u(u_m(ny,1),x(1),t_n)
                ELSE
                        IF (iter == 1) THEN
                                T_m =   tbc2(u_m(ny,1),x(1),t_n)
                                Tu_m = tbc_u(u_m(ny,1),x(1),t_n)
                        ELSE
                                T_m =   tbc2(u_m(ny,1),x(1),t_np1)
                                Tu_m = tbc_u(u_m(ny,1),x(1),t_np1)
                        END IF
                END IF
                rt = (f_top(x(1),t_np1) - T_m )/Tu_m
                r(ny,1) = 0.5_rp*( r(ny,1) + rt)                ! BOTH Left & Top BCs are General Dirichlet.
                IF (stage_flag == 1) THEN
                        IF (linear_flag /= 1) r1(ny,1) = 0.0_rp         ! Calculate residual for the 1st stage of a Non-Linear PDE.
                END IF
        ENDIF
ELSE                                                            ! GENERAL LINEAR/NON-LINEAR Neumann OR Robin Left BC.
        j_start = 1
        j_end = ny
        IF (bottom_bc_flag == 0) THEN
                cf(1,1,1) = 0.0_rp
                cf(1,1,2) = 1.0_rp
                cf(1,1,3) = 0.0_rp
                IF (linear_flag == 1) THEN
                        B_m  =  bbc2(u_m(1,1),x(1),t_n)
                        Bu_m = bbc_u(u_m(1,1),x(1),t_n)
                ELSE
                        IF (iter == 1) THEN
                                B_m  =  bbc2(u_m(1,1),x(1),t_n)
                                Bu_m = bbc_u(u_m(1,1),x(1),t_n)
                        ELSE
                                B_m  =  bbc2(u_m(1,1),x(1),t_np1)
                                Bu_m = bbc_u(u_m(1,1),x(1),t_np1)
                        END IF
                END IF
                r(1,1) = (f_bottom(x(1),t_np1) - B_m )/Bu_m     ! ONLY Bottom BC is General Dirichlet.
                IF (stage_flag == 1) THEN
                        IF (linear_flag /= 1) r1(1,1) = 0.0_rp          ! Calculate residual for the 1st stage of a Non-Linear PDE.
                END IF
                j_start = 2
        END IF
```

```
            IF (top_bc_flag == 0) THEN
                        cf(ny,1,1) = 0.0_rp
                        cf(ny,1,2) = 1.0_rp
                        cf(ny,1,3) = 0.0_rp
                        IF (linear_flag == 1) THEN
                                    T_m  =  tbc2(u_m(ny,1),x(1),t_n)
                                    Tu_m = tbc_u(u_m(ny,1),x(1),t_n)
                        ELSE
                                    IF (iter == 1) THEN
                                                T_m  =  tbc2(u_m(ny,1),x(1),t_n)
                                                Tu_m = tbc_u(u_m(ny,1),x(1),t_n)
                                    ELSE
                                                T_m  =  tbc2(u_m(ny,1),x(1),t_np1)
                                                Tu_m = tbc_u(u_m(ny,1),x(1),t_np1)
                                    END IF
                        END IF
                        r(ny,1) = (f_top(x(1),t_np1) - T_m )/Tu_m       ! ONLY Top BC is General Dirichlet.
                        IF (stage_flag == 1) THEN
                                    IF (linear_flag /= 1) r1(ny,1) = 0.0_rp         ! Calculate residual for the 1st stage of a Non-Linear PDE.
                        END IF
                        j_end = ny-1
            END IF
            DO j = j_start, j_end
               IF (stage_flag == 1) THEN
                        DO m = 1, nx
                                    uj(m) = u_m(j,m)
                        END DO
                        IF (linear_flag /= 1) THEN                           ! This is different, unlike for Dirichlet BCs above.
                                    L_m  = (u_x(j,1,uj,x,y,t_np1))*lbc1(u_m(j,1), y(j), t_np1) + lbc2(u_m(j,1), y(j), t_np1)
                        ELSE
                                    L_m  = (u_x(j,1,uj,x,y,t_n))*  lbc1(u_m(j,1), y(j), t_n)   + lbc2(u_m(j,1), y(j), t_n)
                        END IF
                        Lu_m = lbc_u(u_m(j,1), y(j), t_np1)
                        Lux_m = lbc1(u_m(j,1), y(j), t_np1)
                        n1 = 2.0_rp*hx*Lu_m/Lux_m
                        n2 = 2.0_rp*hx*( f_left(y(j),t_np1) - L_m )/Lux_m
                        IF (coord_flag <= 1) THEN                          ! CARTESIAN COORDINATES.
                                    n3          =   (a1(x(1), y(j), t_np1))/( rho*cp(u_m(j,1)) )
                                    n4          = (a2_x(x(1), y(j), t_np1))*kt(u_m(j,1))
                                    n5 = 2.0_rp*(a2(x(1), y(j), t_np1))*(kt_u(u_m(j,1)))*u_x(j,1,uj,x,y,t_np1)
                                    Nux_m  = n3*(n4+n5)

                                    Nuxx_m = n3*(a2(x(1), y(j), t_np1))*(kt(u_m(j,1)))          ! Use n3 from Nux_m calculation.

                                    n6 = hx*Nux_m/(2.0_rp*Nuxx_m)
                                    cf(j,1,1) = 0.0_rp
                                    cf(j,1,2) = cf(j,1,2) + n1*(1.0_rp-n6)
                                    cf(j,1,3) = 2.0_rp
                                    r(j,1) =   r(j,1) + n2*(1.0_rp-n6)
                        ELSE                                                        ! CYLINDRICAL or SPHERICAL COORDINATES. ADJUST as r -> 0.
                                    DO m = 1, nx
                                                  uj(m) = u_m(j,m)
                                                uj_n(m) =   u(j,m)
                                    END DO
                                    n3 = (a1(x(1), y(j), t_np1))*(a2(x(1), y(j), t_np1))
                                    n4 = (a1(x(1), y(j), t_n))*(a2(x(1), y(j), t_n))
                                    IF(coord_flag == 2) THEN
                                                n5 = 1.0_rp
                                    ELSE
                                                n5 = 2.0_rp
                                    END IF

                                    ux = u_x(j,1,uj,x,y,t_np1)
                                    n6 = (kt_u(u_m(j,1)))*cp(u_m(j,1)) - (cp_u(u_m(j,1)))*kt(u_m(j,1))
                                    n7 = (n5 + n3)*u_xx(j,1,u_m,x,y,t_np1)
                                    n8 = (kt_uu(u_m(j,1)))*cp(u_m(j,1)) - (cp_u(u_m(j,1)))*kt_u(u_m(j,1))
                                    n9 = n3*ux*ux
                                    n10 = (f_rhs_u(u_m(j,1), x(1), y(j), t_np1))*  cp(u_m(j,1)) &
                                      &  - (f_rhs(u_m(j,1), x(1), y(j), t_np1))*cp_u(u_m(j,1))
                                    N0u_m = (n6*n7 + n8*n9 + n10)/( rho*(cp(u_m(j,1)))*cp(u_m(j,1)) )

                                    n6 = kt(u_m(j,1))
                                    n8 = kt_u(u_m(j,1))
                                    n10 = f_rhs(u_m(j,1), x(1), y(j), t_np1)
                                    N0_m = (n6*n7 + n8*n9 + n10)/( rho*cp(u_m(j,1)) )           ! Use n7 and n9 from N0u_m calculation.

                                    N0ux_m = 2.0_rp*n3*n8*ux/( rho*cp(u_m(j,1)) )                             ! Use n8 from N0_m calculation.
                                    N0uxx_m  = (n5 + n3)*(kt(u_m(j,1)))/( rho*cp(u_m(j,1)) )

                                    ux = u_x(j,1,uj_n,x,y,t_n)
                                    n6 = kt(u(j,1))
                                    n7 = (n5 + n4)*u_xx(j,1,u,x,y,t_n)
                                    n8 = kt_u(u(j,1))
                                    n9 = n4*ux*ux
                                    n10 = f_rhs(u(j,1), x(1), y(j), t_n)
                                    N0_n = (n6*n7 + n8*n9 + n10)/( rho*cp(u(j,1)) )

                                    n5 = hx*N0ux_m/(2.0_rp*N0uxx_m)
                                    n6 = ( 4.0_rp - k*N0u_m )/(4.0_rp*rx*N0uxx_m)
                                    cf(j,1,1) = 0.0_rp
                                    cf(j,1,2) = -(2.0_rp + n6) + n1*(1.0_rp-n5)
                                    cf(j,1,3) = 2.0_rp
                                    IF (linear_flag == 1) THEN          ! Linear PDE. FIRST STAGE RHS VECTOR.
                                                r(j,1) = -( (( u(j,1) - u_m(j,1) +  0.5_rp*k*(N0_m + N0_n) )/(rx*N0uxx_m)) ) + n2*(1.0_rp-n5)
                                    ELSE                                            ! Non-Linear PDE.
                                                r1(j,1) =  u(j,1) - u_m(j,1) +  0.5_rp*k*(N0_m + N0_n)                  ! NON-LINEAR RESIDUAL.
                                                r(j,1) = -( r1(j,1)/(rx*N0uxx_m) ) + n2*(1.0_rp-n5)        ! FIRST STAGE RHS VECTOR.
                                    END IF
                        END IF
               ELSE
                        ! For stage 2, compute cf and r elements at the left boundary, ONLY for Cylindrical OR Spherical Systems. In these
                        ! cases, the L'Hospital Rule adjusted PDE at the left boundary DOES NOT CONTAIN any y derivative terms, due to the
                        ! symmetry requirement for the Rule to be applied (i.e., U_y, U_yy have to be BOTH 0 (ZERO) as x --> 0.
                        IF ( (coord_flag == 2) .OR. (coord_flag == 3) ) THEN
                                    cf(j,1,1) = 0.0_rp
                                    cf(j,1,2) = 1.0_rp
                                    cf(j,1,3) = 0.0_rp
                                    r(j,1) =  u(j,1)
                        END IF
               END IF
            END DO
END IF
```

```
!           STEP 2(b): Right Boundary (i = nx) & Right corner points (i = nx, WITH j = 1 OR ny).
!           For spherical coordinate system, the corner points are NOT considered here since the top and bottom boundary
!           functionals are different from the one in the interior.
!           ---------------------------------------------------------------------------------
!           The generalized BC is given by: R_m(U) = Ux*R1(U) + R2(U) = f_right(y,t(m)). Therefore, Ru_m = Ux*R1u + R2u;
!           and Rux_m =R1. Also, for a generalized Dirichlet BC, R1 = 0 => R1u = 0. So, in this case, R_m = R2,
!           Ru_m = R2u, and Rux_m = 0. For the linear case, R_m = f_right(y,t(n)), Ru_m = alpha_x (= 0 for linear
!           Neumann BC), and Rux_m = 1. In terms of components, for linear Neumann, R1_m = 1, R2_m = 0; for linear
!           Robin BC, R1_m = 1, R2_m = alpha_x*U_n. All these values are taken care of, in the module "pde_routines",
!           above, where separate subroutines are defined for each component of the right BC.

IF (right_bc_flag == 0) THEN                                ! GENERAL LINEAR/NON-LINEAR Dirichlet Left BC.
            DO j = 1, ny
                        cf(j,nx,1) = 0.0_rp
                        cf(j,nx,2) = 1.0_rp
                        cf(j,nx,3) = 0.0_rp
                        IF (linear_flag == 1) THEN
                                    R_m =   rbc2(u_m(j,nx),y(j),t_n)
                                    Ru_m = rbc_u(u_m(j,nx),y(j),t_n)
                        ELSE
                                    IF (iter == 1) THEN
                                                R_m =   rbc2(u_m(j,nx),y(j),t_n)
                                                Ru_m = rbc_u(u_m(j,nx),y(j),t_n)
                                    ELSE
                                                R_m =   rbc2(u_m(j,nx),y(j),t_np1)
                                                Ru_m = rbc_u(u_m(j,nx),y(j),t_np1)
                                    END IF
                        END IF
                        r(j,nx) = (f_right(y(j),t_np1) - R_m )/Ru_m     ! RHS value if Right BC is Linear/Non-Linear Dirichlet.
                        IF (stage_flag == 1) THEN
                                    IF (linear_flag /= 1) r1(j,nx) = 0.0_rp         ! Calculate residual for the 1st stage of a Non-Linear PDE.
                        END IF
            END DO
            IF (bottom_bc_flag == 0) THEN
                        IF (linear_flag == 1) THEN
                                    B_m =   bbc2(u_m(1,nx),x(nx),t_n)
                                    Bu_m = bbc_u(u_m(1,nx),x(nx),t_n)
                        ELSE
                                    IF (iter == 1) THEN
                                                B_m =   bbc2(u_m(1,nx),x(nx),t_n)
                                                Bu_m = bbc_u(u_m(1,nx),x(nx),t_n)
                                    ELSE
                                                B_m =   bbc2(u_m(1,nx),x(nx),t_np1)
                                                Bu_m = bbc_u(u_m(1,nx),x(nx),t_np1)
                                    END IF
                        END IF
                        rb = (f_bottom(x(nx),t_np1) - B_m )/Bu_m
                        r(1,nx)= 0.5_rp*(r(1,nx) + rb)                              ! RHS value if Right & Bottom BCs are Dirichlet.
                        IF (stage_flag == 1) THEN
                                    IF (linear_flag /= 1) r1(1,nx) = 0.0_rp         ! Calculate residual for the 1st stage of a Non-Linear PDE.
                        END IF
            END IF
            IF (top_bc_flag == 0) THEN
                        IF (linear_flag == 1) THEN
                                    T_m =   tbc2(u_m(ny,nx),x(nx),t_n)
                                    Tu_m = tbc_u(u_m(ny,nx),x(nx),t_n)
                        ELSE
                                    IF (iter == 1) THEN
                                                T_m =   tbc2(u_m(ny,nx),x(nx),t_n)
                                                Tu_m = tbc_u(u_m(ny,nx),x(nx),t_n)
                                    ELSE
                                                T_m =   tbc2(u_m(ny,nx),x(nx),t_np1)
                                                Tu_m = tbc_u(u_m(ny,nx),x(nx),t_np1)
                                    END IF
                        END IF
                        rt = (f_top(x(nx),t_np1) - T_m )/Tu_m
                        r(ny,nx) = 0.5_rp*( r(ny,nx) + rt)              ! RHS value if Right & Top BCs are Dirichlet.
                        IF (stage_flag == 1) THEN
                              IF (linear_flag /= 1) r1(ny,nx) = 0.0_rp ! Calculate residual for the 1st stage of a Non-Linear PDE.
                        END IF
            ENDIF
 ELSE                                                                   ! GENERAL LINEAR/NON-LINEAR Neumann OR Robin Left BC.
            j_start = 1
            j_end = ny
            IF (bottom_bc_flag == 0) THEN
                        cf(1,nx,1) = 0.0_rp
                        cf(1,nx,2) = 1.0_rp
                        cf(1,nx,3) = 0.0_rp
                        IF (linear_flag == 1) THEN
                                    B_m =   bbc2(u_m(1,nx),x(nx),t_n)
                                    Bu_m = bbc_u(u_m(1,nx),x(nx),t_n)
                        ELSE
                                    IF (iter == 1) THEN
                                                B_m =   bbc2(u_m(1,nx),x(nx),t_n)
                                                Bu_m = bbc_u(u_m(1,nx),x(nx),t_n)
                                    ELSE
                                                B_m =   bbc2(u_m(1,nx),x(nx),t_np1)
                                                Bu_m = bbc_u(u_m(1,nx),x(nx),t_np1)
                                    END IF
                        END IF
                        r(1,nx) = (f_bottom(x(nx),t_np1) - B_m )/Bu_m   ! ONLY Bottom BC is General Dirichlet.
                        IF (stage_flag == 1) THEN
                                    IF (linear_flag /= 1) r1(1,nx) = 0.0_rp         ! Calculate residual for the 1st stage of a Non-Linear PDE.
                        END IF
                        j_start = 2
            ELSE
                        !           For spherical system, and non-Dirichlet BCs, do not compute right-bottom corner point because the
                        !           form of the functional changes for THETA = 0 or PI (corresponding to "bottom" and "top" BCs).
                        IF (coord_flag == 3) j_start = 2
            END IF
            IF (top_bc_flag == 0) THEN
                        cf(ny,nx,1) = 0.0_rp
                        cf(ny,nx,2) = 1.0_rp
                        cf(ny,nx,3) = 0.0_rp
                        IF (linear_flag == 1) THEN
                                    T_m =   tbc2(u_m(ny,nx),x(nx),t_n)
                                    Tu_m = tbc_u(u_m(ny,nx),x(nx),t_n)
                        ELSE
                                    IF (iter == 1) THEN
                                                T_m =   tbc2(u_m(ny,nx),x(nx),t_n)
                                                Tu_m = tbc_u(u_m(ny,nx),x(nx),t_n)
```

```
                        ELSE
                                    T_m =   tbc2(u_m(ny,nx),x(nx),t_np1)
                                    Tu_m = tbc_u(u_m(ny,nx),x(nx),t_np1)
                        END IF
            END IF
            r(ny,nx) = (f_top(x(nx),t_np1) - T_m )/Tu_m     ! ONLY Top BC is General Dirichlet.
            IF (stage_flag == 1) THEN
                  IF (linear_flag /= 1) r1(ny,nx) = 0.0_rp ! Calculate residual for the 1st stage of a Non-Linear PDE.
            END IF
            j_end = ny-1
      ELSE
            !           For spherical system, and non-Dirichlet BCs, do not compute right-bottom corner point because the
            !           form of the functional changes for THETA = 0 or PI (corresponding to "bottom" and "top" BCs).
            IF (coord_flag == 3) j_end = ny-1
      END IF
      DO j = j_start, j_end
         IF (stage_flag == 1) THEN                    ! For the RIGHT Boundary, STAGE 2 NEED NOT be modified from that above.
            DO m = 1, nx
                        uj(m) = u_m(j,m)
            END DO
            IF (linear_flag /= 1) THEN                        ! This is different, unlike for Dirichlet BCs above.
                        R_m  = (u_x(j,nx,uj,x,y,t_np1))*rbc1(u_m(j,nx), y(j), t_np1) + rbc2(u_m(j,nx), y(j), t_np1)
            ELSE
                        R_m  = (u_x(j,nx,uj,x,y,t_n))*  rbc1(u_m(j,nx), y(j), t_n)   + rbc2(u_m(j,nx), y(j), t_n)
            END IF
            Ru_m = rbc_u(u_m(j,nx), y(j), t_np1)
            Rux_m = rbc1(u_m(j,nx), y(j), t_np1)
            n1 = 2.0_rp*hx*Ru_m/Rux_m
            n2 = 2.0_rp*hx*( f_right(y(j),t_np1) - R_m )/Rux_m
            n3      =   (a1(x(nx), y(j), t_np1))/( rho*cp(u_m(j,nx)) )
            n4      = (a2_x(x(nx), y(j), t_np1))*kt(u_m(j,nx))
            n5 = 2.0_rp*(a2(x(nx), y(j), t_np1))*(kt_u(u_m(j,nx)))*u_x(j,nx,uj,x,y,t_np1)
            Nux_m = n3*(n4+n5)

            Nuxx_m = n3*(a2(x(nx), y(j), t_np1))*(kt(u_m(j,nx)))                    ! Use n3 from Nux_m calculation.

            n6 = hx*Nux_m/(2.0_rp*Nuxx_m)
            cf(j,nx,1) = 2.0_rp
            cf(j,nx,2) = cf(j,nx,2) - n1*(1.0_rp + n6)
            cf(j,nx,3) = 0.0_rp
            r(j,nx) =   r(j,nx) - n2*(1.0_rp + n6)
         END IF
      END DO
END IF

!           STEP 2(c): Bottom Boundary: Corners have been taken care of under the left and right boundary loops.
!           ---------------------------------------------------------------------------------------------------
!           The generalized BC is given by: B_m(U) = Ux*B1(U) + B2(U) = f_bottom(x,t(m)). Therefore, Bu_m = Ux*B1u + B2u;
!           and Bux_m =B1. Also, for a generalized Dirichlet BC, B1 = 0 => B1u = 0. So, in this case, B_m = B2,
!           Bu_m = B2u, and Bux_m = 0. For the linear case, B_m = f_bottom(x,t(n)), Bu_m = alpha_y (= 0 for linear
!           Neumann BC), and Bux_m = 1. In terms of components, for linear Neumann, B1_m = 1, B2_m = 0; for linear
!           Robin BC, B1_m = 1, B2_m = alpha_x*U_n. All these values are taken care of, in the module "pde_routines",
!           above, where separate subroutines are defined for each component of the bottom BC.

IF (bottom_bc_flag == 0) THEN                   ! GENERALIZED LINEAR/NON-LINEAR Dirichlet Bottom BC.
      DO i = 2, nx                              ! i_end=nx: Right-Bottom Corner Point was not included in Right BC for Spher. System.
            cf(1,i,1) = 0.0_rp
            cf(1,i,2) = 1.0_rp
            cf(1,i,3) = 0.0_rp
            IF (linear_flag == 1) THEN
                        B_m  =  bbc2(u_m(1,i), x(i), t_n)
                        Bu_m = bbc_u(u_m(1,i), x(i), t_n)
            ELSE
                        IF (iter == 1) THEN
                                    B_m  =  bbc2(u_m(1,i), x(i), t_n)
                                    Bu_m = bbc_u(u_m(1,i), x(i), t_n)
                        ELSE
                                    B_m  =  bbc2(u_m(1,i), x(i), t_np1)
                                    Bu_m = bbc_u(u_m(1,i), x(i), t_np1)
                        END IF
            END IF
            r(1,i) = ( f_bottom(x(i),t_np1) - B_m )/Bu_m    ! RHS value if Bottom BC is Linear/Non-Linear Dirichlet.
            IF (stage_flag == 1) THEN
                        IF (linear_flag /= 1) r1(1,i) = 0.0_rp          ! Calculate residual for the 1st stage of a Non-Linear PDE.
            END IF
      END DO
ELSE                                                          ! GENERALIZED LINEAR/NON-LINEAR Neumann OR Robin Bottom BC.
      ! For the BOTTOM Boundary, STAGE 1 NEED NOT be modified from that above, EXCEPT for the Spherical
      ! coordinate system (coord_flag = 3). In that case, for all i, the bottom LHS coefficients and
      ! RHS vector are identical in form to the Non-spherical cases, except that N and its derivatives are
      ! replaced by the spherical non-linear functional at the bottom boundary, Ns, in both stages.
      IF (stage_flag == 1) THEN
         IF (coord_flag == 3) THEN        ! For Spherical Coordinate System
            DO m = 1, nx
                  uj(m) = u_m(1,m)
               uj_n(m) =   u(1,m)
            END DO
            DO i = i_start, i_end
                        ux = u_x(1,i,uj,x,y,t_np1)
                        uy = u_y(1,i,u_m(:,i),x,y,t_np1)
                        n1 = (kt_u(u_m(1,i)))*cp(u_m(1,i)) - (cp_u(u_m(1,i)))*kt(u_m(1,i))
                        n2 =        (a1(x(i), y(1), t_np1))*(a2_x(x(i), y(1), t_np1))*ux
                        n3 =        (a1(x(i), y(1), t_np1))*(  a2(x(i), y(1), t_np1))*u_xx(1,i,u_m,x,y,t_np1)
                        n4 = 2.0_rp*(b1(x(i), y(1), t_np1))*(  b2(x(i), y(1), t_np1))*u_yy(1,i,u_m,x,y,t_np1)
                        n5 = (kt_uu(u_m(1,i)))*cp(u_m(1,i)) - (cp_u(u_m(1,i)))*kt_u(u_m(1,i))
                        n6 =        (a1(x(i), y(1), t_np1))*(  a2(x(i), y(1), t_np1))*ux*ux
                        n7 =        (b1(x(i), y(1), t_np1))*(  b2(x(i), y(1), t_np1))*uy*uy
                        n8 = (f_rhs_u(u_m(1,i), x(i), y(1), t_np1))*  cp(u_m(1,i)) &
                          &  - (f_rhs(u_m(1,i), x(i), y(1), t_np1))*cp_u(u_m(1,i))
                        NSu_m(1,i) = ( n1*(n2 + n3 + n4) + n5*(n6 + n7) + n8 )/( rho*(cp(u_m(1,i)))*cp(u_m(1,i)) )

                        n1 = kt(u_m(1,i))
                        n5 = kt_u(u_m(1,i))
                        n8 = f_rhs(u_m(1,i), x(i), y(1), t_np1)
                        NS_m = ( n1*(n2 + n3 + n4) + n5*(n6 + n7) + n8 )/( rho*cp(u_m(1,i)) ) ! Use n2-n4 & n6-n7 from NSu_m.

                        n4 = (a1(x(i), y(1), t_np1))/( rho*cp(u_m(1,i)) )
                        n2 = n1*a2_x(x(i), y(1), t_np1)                                         ! Use n1 from NS_m calculation.
                        n3 = 2.0_rp*n5*(a2(x(i), y(1), t_np1))*ux                   ! Use n5 from NS_m calculation.
                        NSux_m = n4*(n2+n3)

                        NSuxx_m = n4*(a2(x(i), y(1), t_np1))*n1                     ! Use n4 from NSux_m calc., n1 from NS_m calc.

                        n2 = hx*NSux_m/(2.0_rp*NSuxx_m)
```

```fortran
                        n3 = ( 4.0_rp - k*NSu_m(1,i) )/(4.0_rp*rx*NSuxx_m)
                        cf(1,i,1) = 1.0_rp - n2                                                 ! First LHS Coefficient.
                        cf(1,i,2) = -(2.0_rp + n3)                                              ! Second LHS Coefficient.
                        cf(1,i,3) = 1.0_rp + n2                                                 ! Third LHS Coefficient.

                        ux = u_x(1,i,uj_n,x,y,t_n)
                        uy = u_y(1,i,u(:,i),x,y,t_n)
                        n1 = kt(u(1,i))
                        n2 =        (a1(x(i), y(1), t_n))*(a2_x(x(i), y(1), t_n))*ux
                        n3 =        (a1(x(i), y(1), t_n))*(  a2(x(i), y(1), t_n))*u_xx(1,i,u,x,y,t_n)
                        n4 = 2.0_rp*(b1(x(i), y(1), t_n))*(  b2(x(i), y(1), t_n))*u_yy(1,i,u,x,y,t_n)
                        n5 = kt_u(u(1,i))
                        n6 =        (a1(x(i), y(1), t_n))*(  a2(x(i), y(1), t_n))*ux*ux
                        n7 =        (b1(x(i), y(1), t_n))*(  b2(x(i), y(1), t_n))*uy*uy
                        n8 = f_rhs(u(1,i), x(i), y(1), t_n)
                        NS_n = ( n1*(n2 + n3 + n4) + n5*(n6 + n7) + n8 )/( rho*cp(u(1,i)) )

                        IF (linear_flag == 1) THEN          ! Linear PDE. FIRST STAGE RHS VECTOR.
                                r(1,i) = -( u(1,i) - u_m(1,i) + 0.5_rp*k*(NS_m + NS_n) )/(rx*NSuxx_m)

                        ELSE                                            ! Non-Linear PDE.
                                r1(1,i) =   u(1,i) - u_m(1,i) + 0.5_rp*k*(NS_m + NS_n)      ! NON-LINEAR RESIDUAL.
                                 r(1,i) = -r1(1,i)/(rx*NSuxx_m)                             ! FIRST STAGE RHS VECTOR.
                        END IF
                END DO
           END IF
        ELSE                                        ! Second Stage
                DO i = i_start, i_end   ! Any coordinate system. For Cart. or Cyl. system, no calculation for i=nx.
                        IF (linear_flag /= 1) THEN
                           B_m  = (u_y(1,i,u_m(:,i),x,y,t_np1))*bbc1(u_m(1,i), x(i), t_np1) + bbc2(u_m(1,i), x(i), t_np1)
                        ELSE
                           B_m  = (u_y(1,i,u_m(:,i),x,y,t_n))*bbc1(u_m(1,i), x(i), t_n) + bbc2(u_m(1,i), x(i), t_n)
                        END IF
                        Bu_m = bbc_u(u_m(1,i), x(i), t_np1)
                        Buy_m = bbc1(u_m(1,i), x(i), t_np1)
                        n1 = 2.0_rp*hy*Bu_m/Buy_m
                        n2 = 2.0_rp*hy*( f_bottom(x(i),t_np1) - B_m )/Buy_m
                        n3   =      (b1(x(i), y(1), t_np1))/( rho*cp(u_m(1,i)) )
                        n4   =   (b2_y(x(i), y(1), t_np1))*kt(u_m(1,i))
                        n5 = 2.0_rp*(b2(x(i), y(1), t_np1))*(kt_u(u_m(1,i)))*u_y(1,i,u_m(:,i),x,y,t_np1)
                        IF (coord_flag <= 2) THEN           ! FOR USER DEFINED ANALYTIC SYSTEM, CARTESIAN AND CYLINDRICAL COORDINATES.
                                IF (i /= nx) THEN       ! Right-Bottom Corner Point Already Computed under Right BC.
                                        Nuy_m = n3*(n4+n5)
                                        Nuyy_m = n3*(b2(x(i), y(1), t_np1))*(kt(u_m(1,i)))

                                        n6 = hy*Nuy_m/(2.0_rp*Nuyy_m)
                                        cf(1,i,1) = 0.0_rp
                                        cf(1,i,2) = cf(1,i,2) + n1*(1.0_rp-n6)
                                        cf(1,i,3) = 2.0_rp
                                        r(1,i) = r(1,i) + n2*(1.0_rp-n6)
                                END IF


                        ELSE                            ! FOR SPHERICAL COORDINATES. Includes Right-Bottom Corner point.
                                NSuy_m = n3*n5
                                NSuyy_m = 2.0_rp*n3*(b2(x(i), y(1), t_np1))*(kt(u_m(1,i)))

                                n6 = ( 4.0_rp - k*NSu_m(1,i) )/(4.0_rp*ry*NSuyy_m)
                                n7 = hy*NSuy_m/(2.0_rp*NSuyy_m)
                                cf(1,i,1) = 0.0_rp
                                cf(1,i,2) = -(2.0_rp + n6) + n1*(1.0_rp-n7)
                                cf(1,i,3) = 2.0_rp
                                r(1,i) = -(u(1,i))/(ry*NSuyy_m) + n2*(1.0_rp-n7)
                        END IF
                END DO
        END IF
END IF

!       STEP 2(d): Top Boundary: Corners have been taken care of under the left and right boundary loops.
!       -----------------------------------------------------------------------------------------------
!       The generalized BC is given by: T_m(U) = Ux*T1(U) + T2(U) = f_top(x,t(m)). Therefore, Tu_m = Ux*T1u + T2u;
!       and Tux_m =T1. Also, for a generalized Dirichlet BC, T1 = 0 => T1u = 0. So, in this case, T_m = T2,
!       Tu_m = T2u, and Tux_m = 0. For the linear case, T_m = f_top(x,t(n)), Tu_m = alpha_y (= 0 for linear
!       Neumann BC), and Tux_m = 1. In terms of components, for linear Neumann, T1_m = 1, T2_m = 0; for linear
!       Robin BC, T1_m = 1, T2_m = alpha_x*U_n. All these values are taken care of, in the module "pde_routines",
!       above, where separate subroutines are defined for each component of the top BC.

IF (top_bc_flag == 0) THEN                      ! GENERALIZED LINEAR/NON-LINEAR Dirichlet Top BC.
        DO i = 2, nx                            ! i_end=nx: Right-Top Corner Point was not included in Right BC for Spherical System.
                cf(ny,i,1) = 0.0_rp
                cf(ny,i,2) = 1.0_rp
                cf(ny,i,3) = 0.0_rp
                IF (linear_flag == 1) THEN
                        T_m  =  tbc2(u_m(ny,i), x(i), t_n)
                        Tu_m = tbc_u(u_m(ny,i), x(i), t_n)
                ELSE
                        IF (iter == 1) THEN
                                T_m  =  tbc2(u_m(ny,i), x(i), t_n)
                                Tu_m = tbc_u(u_m(ny,i), x(i), t_n)
                        ELSE
                                T_m  =  tbc2(u_m(ny,i), x(i), t_np1)
                                Tu_m = tbc_u(u_m(ny,i), x(i), t_np1)
                        END IF
                END IF
                r(ny,i) = (f_top(x(i),t_np1) - T_m )/Tu_m
                IF (stage_flag == 1) THEN
                        IF (linear_flag /= 1) r1(ny,i) = 0.0_rp         ! Calculate residual for the 1st stage of a Non-Linear PDE.
                END IF
        END DO
ELSE                                                    ! GENERALIZED LINEAR/NON-LINEAR Neumann OR Robin Top BC.
        ! For the TOP Boundary, STAGE 1 NEED NOT be modified from that above, EXCEPT for the Spherical
        ! coordinate system (coord_flag = 3). In that case, for all i, the top LHS coefficients and
        ! RHS vector are identical in form to the Non-spherical cases, except that N and its derivatives are
        ! replaced by the spherical non-linear functional at the top boundary, Ns, in both stages.
        IF (stage_flag == 1) THEN
           IF (coord_flag == 3) THEN        ! For Spherical Coordinate System
                DO m = 1, nx
                     uj(m) = u_m(ny,m)
                   uj_n(m) =   u(ny,m)
                END DO
```

```
                    DO i = i_start, i_end
                            ux = u_x(ny,i,uj,x,y,t_np1)
                            uy = u_y(ny,i,u_m(:,i),x,y,t_np1)
                            n1 = (kt_u(u_m(ny,i)))*cp(u_m(ny,i)) - (cp_u(u_m(ny,i)))*kt(u_m(ny,i))
                            n2 =        (a1(x(i), y(ny), t_np1))*(a2_x(x(i), y(ny), t_np1))*ux
                            n3 =        (a1(x(i), y(ny), t_np1))*(  a2(x(i), y(ny), t_np1))*u_xx(ny,i,u_m,x,y,t_np1)
                            n4 = 2.0_rp*(b1(x(i), y(ny), t_np1))*(  b2(x(i), y(ny), t_np1))*u_yy(ny,i,u_m,x,y,t_np1)
                            n5 = (kt_uu(u_m(ny,i)))*cp(u_m(ny,i)) - (cp_u(u_m(ny,i)))*kt_u(u_m(ny,i))
                            n6 =        (a1(x(i), y(ny), t_np1))*(  a2(x(i), y(ny), t_np1))*ux*ux
                            n7 =        (b1(x(i), y(ny), t_np1))*(  b2(x(i), y(ny), t_np1))*uy*uy
                            n8 = (f_rhs_u(u_m(ny,i), x(i), y(ny), t_np1))*  cp(u_m(ny,i)) &
                              &  - (f_rhs(u_m(ny,i), x(i), y(ny), t_np1))*cp_u(u_m(ny,i))
                            NSu_m(2,i) = ( n1*(n2 + n3 + n4) + n5*(n6 + n7) + n8 )/( rho*(cp(u_m(ny,i)))*cp(u_m(ny,i)) )

                            n1 = kt(u_m(ny,i))
                            n5 = kt_u(u_m(ny,i))
                            n8 = f_rhs(u_m(ny,i), x(i), y(ny), t_np1)
                            NS_m = ( n1*(n2 + n3 + n4) + n5*(n6 + n7) + n8 )/(rho*cp(u_m(ny,i))) ! Use n2-n4 & n6-n7 from NSu_m.

                            n4 = (a1(x(i), y(ny), t_np1))/( rho*cp(u_m(ny,i)) )
                            n2 = n1*a2_x(x(i), y(ny), t_np1)                            ! Use n1 from NS_m calculation.
                            n3 = 2.0_rp*n5*(a2(x(i), y(ny), t_np1))*ux                  ! Use n5 from NS_m calculation.
                            NSux_m = n4*(n2+n3)

                            NSuxx_m = n4*(a2(x(i), y(ny), t_np1))*n1       ! Use n4 from NSux_m calc., n1 from NS_m calc.

                            n2 = hx*NSux_m/(2.0_rp*NSuxx_m)
                            n3 = ( 4.0_rp - k*NSu_m(2,i) )/(4.0_rp*rx*NSuxx_m)
                            cf(ny,i,1) = 1.0_rp - n2                                                ! First LHS Coefficient.
                            cf(ny,i,2) = -(2.0_rp + n3)                                             ! Second LHS Coefficient.
                            cf(ny,i,3) = 1.0_rp + n2                                                ! Third LHS Coefficient.

                            ux = u_x(ny,i,uj_n,x,y,t_n)
                            uy = u_y(ny,i,u(:,i),x,y,t_n)
                            n1 = kt(u(ny,i))
                            n2 =        (a1(x(i), y(ny), t_n))*(a2_x(x(i), y(ny), t_n))*ux
                            n3 =        (a1(x(i), y(ny), t_n))*(  a2(x(i), y(ny), t_n))*u_xx(ny,i,u,x,y,t_n)
                            n4 = 2.0_rp*(b1(x(i), y(ny), t_n))*(  b2(x(i), y(ny), t_n))*u_yy(ny,i,u,x,y,t_n)
                            n5 = kt_u(u(ny,i))
                            n6 =        (a1(x(i), y(ny), t_n))*(  a2(x(i), y(ny), t_n))*ux*ux
                            n7 =        (b1(x(i), y(ny), t_n))*(  b2(x(i), y(ny), t_n))*uy*uy
                            n8 = f_rhs(u(ny,i), x(i), y(ny), t_n)
                            NS_n = ( n1*(n2 + n3 + n4) + n5*(n6 + n7) + n8 )/( rho*cp(u(ny,i)) )

                            IF (linear_flag == 1) THEN          ! Linear PDE. FIRST STAGE RHS VECTOR.
                                    r(ny,i) = -( u(ny,i) - u_m(ny,i) + 0.5_rp*k*(NS_m + NS_n) )/(rx*NSuxx_m)
                            ELSE                                            ! Non-Linear PDE.
                                    r1(ny,i) =   u(ny,i) - u_m(ny,i) + 0.5_rp*k*(NS_m + NS_n)   ! NON-LINEAR RESIDUAL.
                                     r(ny,i) = -r1(ny,i)/(rx*NSuxx_m)                           ! FIRST STAGE RHS VECTOR.
                            END IF
                    END DO
               END IF
            ELSE
                    DO i = i_start, i_end   ! Any coordinate system. For Cart. or Cyl. system, no calculation for i=nx.
                            IF (linear_flag /= 1) THEN
                               T_m  = (u_y(ny,i,u_m(:,i),x,y,t_np1))*tbc1(u_m(ny,i), x(i), t_np1) + tbc2(u_m(ny,i), x(i), t_np1)
                            ELSE
                               T_m  = (u_y(ny,i,u_m(:,i),x,y,t_n))*tbc1(u_m(ny,i), x(i), t_n) + tbc2(u_m(ny,i), x(i), t_n)
                            END IF
                            Tu_m = tbc_u(u_m(ny,i), x(i), t_np1)
                            Tuy_m = tbc1(u_m(ny,i), x(i), t_np1)
                            n1 = 2.0_rp*hy*Tu_m/Tuy_m
                            n2 = 2.0_rp*hy*( f_top(x(i),t_np1) - T_m )/Tuy_m
                            n3   =      (b1(x(i), y(ny), t_np1))/( rho*cp(u_m(ny,i)) )
                            n4   =    (b2_y(x(i), y(ny), t_np1))*kt(u_m(ny,i))
                            n5 = 2.0_rp*(b2(x(i), y(ny), t_np1))*(kt_u(u_m(ny,i)))*u_y(ny,i,u_m(:,i),x,y,t_np1)
                            IF (coord_flag <= 2) THEN           ! FOR USER DEFINED ANALYTIC SYSTEM, CARTESIAN AND CYLINDRICAL COORDINATES.
                                    IF (i /= nx) THEN       ! Right-Top Corner Point Already Computed under Right BC.
                                            Nuy_m = n3*(n4+n5)
                                            Nuyy_m = n3*(b2(x(i), y(ny), t_np1))*(kt(u_m(ny,i)))

                                            n6 = hy*Nuy_m/(2.0_rp*Nuyy_m)
                                            cf(ny,i,1) = 2.0_rp
                                            cf(ny,i,2) = cf(ny,i,2) - n1*(1.0_rp+n6)
                                            cf(ny,i,3) = 0.0_rp
                                            r(ny,i) = r(ny,i) - n2*(1.0_rp+n6)
                                    END IF
                            ELSE                                ! FOR SPHERICAL COORDINATES. Includes Right-Top Corner point.
                                    NSuy_m = n3*n5
                                    NSuyy_m = 2.0_rp*n3*(b2(x(i), y(ny), t_np1))*(kt(u_m(ny,i)))

                                    n6 = ( 4.0_rp - k*NSu_m(2,i) )/(4.0_rp*ry*NSuyy_m)
                                    n7 = hy*NSuy_m/(2.0_rp*NSuyy_m)
                                    cf(ny,i,1) = 2.0_rp
                                    cf(ny,i,2) = -(2.0_rp + n6) - n1*(1.0_rp+n7)
                                    cf(ny,i,3) = 0.0_rp
                                     r(ny,i) = -(u(ny,i))/(ry*NSuyy_m) - n2*(1.0_rp+n7)
                            END IF
                    END DO
            END IF
    END IF

!           STEP 3. FINALLY, CONVERT the 3-D CF array into the 2-D COEFF array AND
!           CONVERT the 2-D R array into the RHS vector:
!           IMPORTANT NOTE:         stage_flag = 1: CONVERT BY ROWS TO MAINTAIN TRI-DIAGONALITY.
!                                   stage_flag = 2: CONVERT BY COLUMNS TO MAINTAIN TRI-DIAGONALITY.
    l = 1
    IF (stage_flag == 1) THEN                       ! STAGE 1: CONVERT BY ROWS (Lines // x-direction).
            DO j = 1, ny
                    DO i = 1, nx
                            coeff(l,1) = cf(j,i,1)
                            coeff(l,2) = cf(j,i,2)
                            coeff(l,3) = cf(j,i,3)
                            rhs(l) = r(j,i)
                            IF (linear_flag /= 1) THEN
                                    residual(l) = r1(j,i)   ! For NON-LINEAR PROBLEMS, calculate Residual here.
                            END IF
                            l = l + 1
                    END DO
            END DO
```

```
            ELSE                                                                        ! STAGE 2: CONVERT BY COLUMNS (Lines // y-direction).
                        DO i = 1, nx
                                    DO j = 1, ny
                                                coeff(l,1) = cf(j,i,1)
                                                coeff(l,2) = cf(j,i,2)
                                                coeff(l,3) = cf(j,i,3)
                                                rhs(l) = r(j,i)
                                                l = l + 1
                                    END DO
                        END DO
            END IF

END SUBROUTINE qldgts_coeff_rhs

!-------------------------------------------------------------------------------------------------------------------
SUBROUTINE delta_qlin_dgts(x, y, t, k, u, en_est, dn, srad)
IMPLICIT NONE
!-------------------------------------------------------------------------------------------------------------------
!           This routine computes the solution at A SINGLE TIME STEP, of a generalized heat conduction equation of the form:
!           Ut = {1/(rho*cp)}*
!           [a1*{kt*(a2_x*U_x + a2*U_xx) + a2*kt_u*(U_x)^2} + b1*{kt*(b2_y*U_y + b2*U_yy) + b2*kt_u*(U_y)^2} + f(U,x,y,t)],
!           where the "_" denotes partial differentiation, obtained by expanding a general ADJOINT form of the Conduction
!           Equation. The values of functions a1, a2, b1, b2, kt(u) and cp(u) can be changed to match any regular, closed
!           domain. This routine uses the DELTA-FORM of QUASILINIARIZATION (NEWTON-KANTOROVICH PROCEDURE) in conjunction
!           with the DELTA-FORM of DOUGLAS-GUNN TIME SPLITTING SCHEME (2-STEP). Here, for each iteration of the
!           quasilinearization process, the "imporved" iterate is constructed using two stages corresponding to the
!           2-step Douglas-Gunn scheme.  It uses the initial guess, U, provided by the MAIN PROGRAM for EACH time step, to
!           iterate to a converged value for that time step. It outputs the grid function values for the input time
!           step, u(x,y,t(n)), back to the main program. The grid function at each time stage of a SINGLE iteration is
!           related to the previous one by the compact time-split matrix formulae:
!           {coeff_1(n+1)}*dv(1) = {rhs_1(n+1, n)}  &
!           {coeff_2(n+1)}*dv(2) = dv(1)
!           where n denotes the time step, and "coeff_i" are tridiagonal matrices of dimension nx*ny (=n).
!           Only the band diagonal elements of the tridiagonal systems are computed & stored in this program (in the
!           subroutine "qldgts_coeff_rhs", to minimize storage. They are stored in the form of n X 3 matrices, where the
!           three columns are, respectively, the sub-diagonal, diagonal, and super-diagonal elements of the original n X n
!           system matrix. The program calls the LU decomposition routine to compute the grid-functions, u, after each time
!           step. It also stores the grid function values at the last time step, as they are required for Newton-Kantorovich
!           iterations. IF SMOOTHING FLAG IS NON-ZERO, APPROPRIATE SMOOTHING OF GRID FUNCTION VALUES IS CARRIED OUT. HOWEVER,
!           THIS IS HIGHLY CASE-SPECIFIC AND THE SUB-SET OF U VALUES TO BE SMOOTHED WILL BE DIFFERENT FOR EACH PROBLEM-BC COMBO.
!-------------------------------------------------------------------------------------------------------------------

            !           Arguments
            REAL(KIND=rp), DIMENSION(:), INTENT(IN) :: x, y
            REAL(KIND=rp), INTENT(IN) :: k,t
            REAL(KIND=rp), DIMENSION(:,:), INTENT(INOUT) :: u
            REAL(KIND=rp), DIMENSION(:,:), INTENT(OUT), OPTIONAL :: en_est
            REAL(KIND=rp), DIMENSION (:), INTENT(OUT), OPTIONAL :: dn, srad

            !           Local Variable Declarations
            REAL(KIND=rp), DIMENSION(quasi_iterations) :: dn_norm, rs_norm
            REAL(KIND=rp) :: hx, hy
            INTEGER(KIND=ip) :: i, iter, j, l, n, nx, ny, stage_flag

            nx = SIZE(x)
            ny = SIZE(y)
            n  = nx*ny
            hx = (x_right - x_left)/(nx - 1)
            hy = (y_top - y_bottom)/(ny - 1)

            !           Save incoming value of u at last time step, as well as set the initial guess u_old to u at the
            !           last time step.
            u_n = u                 ! This is necessary since u_n WILL BE NEEDED AT EVERY ITERATION, for the first D-G stage.
            IF (linear_flag /= 1) THEN
                        u_old = u_n ! This assignes the first guess of the iterations as the value at the last time step.
            END IF

            !           Start the iteration loop. The convergence limit, "quasi_epsilon", is calculated in the MAIN Program
            !           based on the value of the time step, k, and generally assigned a value of at least k^3, to ensure
            !           that DELTA_u is less than the maximum truncation error. This variable is, however, defined globally
            !           in the module "const_params" above. The maximum number of iterations should also be set in the
            !           globally defined variable "quasi_iterations", in the Main Program. NOTE: These iterations solve for
            !           the solution to the fully LINEARIZED (Frechet-Taylor) form of the original non-linear PDE. If k is
            !           sufficiently small, the value of the first iterate can be reasonably taken to be the value at the
            !           previous time step.

            DO iter = 1, quasi_iterations
                        IF (iter > 1) THEN
                                    IF (linear_flag /= 1) THEN
                                    ! If the PDE is non-linear, then beginning with the second iteration, check for convergence before
                                    ! starting any calculations for this iteration. Can use either residual or Dn for testing convergence.
                                    ! Simply remove the comment symbol, and CHANGE between "rs_norm" and "dn_norm" in the PRINT statement.
                                                IF (exact_sol_flag == 0) THEN
!                                                           IF (rs_norm(iter-1) < quasi_epsilon) THEN
                                                            IF (dn(iter-1) < quasi_epsilon) THEN
                                                                        en_est = (dn(iter-1))/(1.0_rp - srad(iter-1))
                                                                        !           OPTIONAL PRINT STATEMENTS:
                                                                        IF (verbose_flag == 1) THEN
                                                                            PRINT '(A7,ES9.3,A48,I2,A43,ES12.6, A1)', "TIME = ",t, &
                                                                            & ". Newton-Kantorovich Iterations Converged after ",iter-1,  &
                                                                            & " iterations. Final value of L2-norm of Dn: ", dn(iter-1),"."
!                                                                           & " iterations. Final value of L2 residual:   ", rs_norm(iter-1),"."
                                                                        END IF
                                                                        RETURN
                                                            END IF
                                                ELSE
!                                                           IF (rs_norm(iter-1) < quasi_epsilon) THEN
                                                            IF (dn_norm(iter-1) < quasi_epsilon) THEN
                                                                        !           OPTIONAL PRINT STATEMENTS:
                                                                        IF (verbose_flag == 1) THEN
                                                                            PRINT '(A7,ES9.3,A48,I2,A43,ES12.6, A1)', "TIME = ",t, &
                                                                            & ". Newton-Kantorovich Iterations Converged after ",iter-1,  &
                                                                            & " iterations. Final value of L2-norm of Dn: ", dn_norm(iter-1),"."
!                                                                           & " iterations. Final value of L2 residual:   ", rs_norm(iter-1),"."
                                                                        END IF
                                                                        RETURN
                                                            END IF
                                                END IF
                                    END IF
                        END IF
```

```
!           If convergence is not observed, proceed to the next iteration:
!           Every DOUGLAS-GUNN time step for the LINEARIZED NON-LINEAR PDE has two stages, each corresponding
!           to one of the spatial directions. START THE STAGE LOOP FOR THE TWO STAGES AT EACH ITERATION, m.
DO stage_flag = 1, 2
            !           Call Subroutine "qldgts_coeff_rhs" to calculate POST-DISCRETIZATION BC-ADJUSTED PDE
            !           Coefficient matrix and PDE RHS vector, as defined by the above PDE. The stage flag
            !           determines which of the two sets of coefficients will be computed by the
            !           "dgts_coeff_rhs" routine. If stage_flag = 1, grid function values from BOTH the previous
            !           time step (u_n), and the last iteration (u_old) are used to calculate the RHS vector,
            !           and COEFF matrix is computed at time n+1. If stage_flag = 2, the intermediate values,
            !           dv(1)/F(m) becomes the RHS vector, and the COEFF matrix is again computed at time n+1 &
            !           using m-th iterate. The solution is updated at the end of the SECOND STAGE for each iteration,
            !           and allows for testing the convergence of the Newton-Kantorovich procedure. Also, the OPTIONAL
            !           rs vector computed at the end of the first stage gives the residual at the current iteration.
            !           L2 NORMS are used  to estimate its magnitude and test for convergence at the beginning of the
            !           next iteration. If the PDE is linear, then in order to use the functional notation and setup
            !           of subroutine "qldgts_coeff_rhs", the value of "u_old" is set to "u_n" or "u", for each stage
            !           respectively.

            !PRINT *, "ITERATION, STAGE: ", Iter, stage_flag
            IF (stage_flag == 1) THEN
                        IF (linear_flag == 1) THEN
                                    CALL qldgts_coeff_rhs(x, y, t, k, stage_flag, iter, u_n, u_n)
                        ELSE
                                    CALL qldgts_coeff_rhs(x, y, t, k, stage_flag, iter, u_n, u_old, rs)
                                    rs_norm(iter) = 0.0_rp
                                    DO i = 1, n
                                                rs_norm(iter) = rs_norm(iter) + (rs(i))*rs(i)
                                    END DO
                                    rs_norm(iter) = SQRT(rs_norm(iter))
                        END IF
            ELSE
                        IF (linear_flag == 1) THEN
                                    CALL qldgts_coeff_rhs(x, y, t, k, stage_flag, iter, u, u_n)
                        ELSE
                                    CALL qldgts_coeff_rhs(x, y, t, k, stage_flag, iter, u, u_old)
                        END IF
            END IF

            !           Call LU-Decomposition Routine to compute the grid function at the current time
            !           STAGE. These INTERMEDIATE grid function DELTAs, dU*, are returned in the "rhs" vector.
            CALL lud_trid(coeff, rhs)

            !           NOTE: In this case, the values obtained in the RHS vector above are the dU (DELTA_U)
            !           values after each stage (dV after first stage and dU after second, notationally).
            !           Since the direction of evaluation changes from column-wise (for each row) in the 1st
            !           stage, to row-wise (for each column) in the 2nd stage, we have to RE-ORDER the RHS vector
            !           output by the lud_trid routine into the Grid Function deltas (dUs) at EACH STAGE of the
            !           time step m. In order to MINIMIZE STORAGE, the intermediate values are overwritten in the
            !           second stage of this time step, to give the FINAL grid function delta values at the end of
            !           the current iteration, m.
            !           IMPORTANT NOTE:
            !           stage_flag = 1: CONVERT BY ROWS (column-wise evaluations) TO MAINTAIN SYSTEM TRI-DIAGONALITY.
            !           stage_flag = 2: CONVERT BY COLUMNS (row-wise evaluations) TO MAINTAIN SYSTEM TRI-DIAGONALITY.
            l = 1
            IF (stage_flag == 1) THEN           ! 1st STAGE: Convert RHS= dv(1)= dV by Rows(Lines//x-axis)
                        DO j = 1, ny
                                    DO i = 1, nx
                                                u(j,i) = rhs(l)
                                                l = l + 1
                                    END DO
                        END DO
            ELSE                                ! 2nd STAGE: Convert RHS= dv(2)= dU, by Columns(Lines//y-axis)
                        DO i = 1, nx
                                    DO j = 1, ny
                                                u(j,i) = rhs(l)
                                                l = l + 1
                                    END DO
                        END DO
            END IF
END DO                  ! Stage Loop.

!           Now update the grid functions to the the value at the CURRENT time step: u(m) = U (=dU) + u_old.
!           Here, dU (U stored in the rhs vector, OR dv(2)) is the reordered form of the last RHS vector
!           output from the lud_trid subroutine, corresponding the output for stage_flag = 2.
DO i = 1, nx
            DO j = 1, ny
                        IF (linear_flag == 1) THEN
                                    u(j,i) = u(j,i) + u_n(j,i)
                        ELSE
                                    u(j,i) = u(j,i) + u_old(j,i)
                        END IF
            END DO
END DO

!           For NON-LINEAR PROBLEMS, if exact solution is NOT available, compute the iteration error norm,
!           temporarily storing the iteration error in the estimated error array, en_est. If the exact solution
!           is known, compute the relative error Dn = u_(n) - u_(n-1) to use for convergence tests. Finally, save the
!           current iteration grid function values for use in the next iteration.
IF (linear_flag /= 1) THEN
            IF (exact_sol_flag == 0) THEN
                        en_est = u - u_old
                        dn(iter) = 0.0_rp
                        DO j = 1, ny
                                    DO i = 1, nx
                                                dn(iter) = dn(iter) + (en_est(j,i))*en_est(j,i)
                                    END DO
                        END DO
                        dn(iter) = SQRT(dn(iter))
                        IF (verbose_flag == 1) PRINT *, "Dn = ", dn(iter)           ! OPTIONAL
                        IF (iter > 1) THEN
                                    srad(iter) = dn(iter)/dn(iter-1)
                        END IF
                        IF (iter == quasi_iterations) THEN  ! Final iteration Warning.
                                    en_est = (dn(iter))/(1.0_rp - srad(iter))
                                    !           OPTIONAL PRINT STATEMENTS:
                                    IF (verbose_flag == 1) THEN
                                        PRINT '(A20,ES9.3,A1,A38,I2,A30,ES12.6)', "*** WARNING: TIME = ",t,".", &
                                        & "N-K Iterations DID NOT Converge after ",iter," iterations. L2-norm of Dn: ", dn(iter)
!                                     & "N-K Iterations DID NOT Converge after ",iter," iterations. LAST L2 residual:",rs_norm(iter)
                                    END IF
                        END IF
```

```fortran
                                        ELSE                    ! Compute Dn to assess the decrease in relative error with iterations.
                                                u_old = u - u_old
                                                dn_norm(iter) = 0.0_rp
                                                DO j = 1, ny
                                                        DO i = 1, nx
                                                                dn_norm(iter) = dn_norm(iter) + (u_old(j,i))*u_old(j,i)
                                                        END DO
                                                END DO
                                                dn_norm(iter) = SQRT(dn_norm(iter))
!       WHEN USING "rs_norm" >          IF (verbose_flag == 1) PRINT *, "Dn = ", dn_norm(iter)
                                                IF (iter == quasi_iterations) THEN  ! Final iteration Warning.
                                                        !       OPTIONAL PRINT STATEMENTS:
                                                        IF (verbose_flag == 1) THEN
                                                          PRINT '(A20,ES9.3,A1,A38,I2,A30,ES12.6)', "*** WARNING: TIME = ",t,".", &
                                                          & "N-K Iterations DID NOT Converge after ",iter," iterations. L2-norm of Dn: ", dn_norm(iter)
!                                                         & "N-K Iterations DID NOT Converge after ",iter," iterations. LAST L2 residual:",rs_norm(iter)
                                                        END IF
                                                END IF
                                        END IF
                                        u_old = u
                                END IF
!                               IF (exact_sol_flag == 0) THEN
!                                       IF (iter == 1) PRINT *, "--------------------------------------------------------------------------------"
!                                       PRINT *, "Newton-Kantorovich Iteration# ",iter,". L2-norm of Dn = ", dn(iter)

!                               ELSE
!                                       IF (iter == 1) PRINT *, "--------------------------------------------------------------------------------"
!                                       PRINT *, "Newton-Kantorovich Iteration# ",iter,". L2-norm of Dn = ", dn_norm(iter)
!                               END IF
!                               IF (iter == 1) PRINT *, "--------------------------------------------------------------------------------"
!                               PRINT *, "Newton-Kantorovich Iteration# ",iter,". L2 RESIDUAL= ", rs_norm(iter)
                        END DO  ! Iterations Loop

        END SUBROUTINE delta_qlin_dgts

        !-----------------------------------------------------------------------------------------------------------------

END MODULE solver_routines

!-----------------------------------------------------------------------------------------------------------------------
PROGRAM nonlin_parabolic_pde
USE const_params
USE fault_params
USE pde_routines
USE solver_routines
IMPLICIT NONE

!-----------------------------------------------------------------------------------------------------------------------
!       Program for the solution of a GENERAL NON-LINEAR, 2D, TIME DEPENDENT HEAT CONDUCTION EQUATION (in Cartesian/
!       Cylindrical/Spherical coordinates OR in ANY USER DEFINED ANALYTIC SYSTEM), with general NON-LINEAR BOUNDARY CONDITIONS
!       USING DELTA-FORM OF QUASILINIARIZATION (NEWTON-KANTOROVICH PROCEDURE) IN CONJUNCTION WITH THE DELTA-FORM OF THE
!       DOUGLAS-GUNN TIME SPLITTING SCHEME (2-STEP). THIS CODE CAN ALSO BE USED FOR LINEAR PROBLEMS WITHOUT ANY CHANGES TO
!       THE CORE ALGORITHM IMPLEMENTED HERE. This code was written as part of the development of an "Asperity scale frictional
!       melting model" for my M.S. Thesis Research. This work was supported by NSF grant: XXXXX-XXXXX. - Ravi Kanda (November, 2002).
!       This program solves an equation of the form:
!       Ut = {1/(rho*cp)}*
!            [a1*{kt*(a2_x*U_x + a2*U_xx) + a2*kt_u*(U_x)^2} + b1*{kt*(b2_y*U_y + b2*U_yy) + b2*kt_u*(U_y)^2} + f(U,x,y,t)],
!       where the "_" denotes partial differentiation, obtained by expanding the ADJOINT form of the linear, but very general
!       Pure Conduction Equation. The values of functions a1, a2, b1, b2, kt(U) and cp(U) can be changed to match any
!       "regular closed domain" (i.e.. Cartesian, Cylindrical, Spherical, Elliptical or ANY USER DEFINED ANALYTIC SYSTEM domains),
!       in either of the three coordinate systems mentioned above. In addition, the treatment of the boundary conditions is very
!       general in that any type of convective/conductive/radiative heat transfer boundary condition can be applied at any of the
!       boundaries. The code adjusts the form of the equation in Spherical AND Cylindrical coordinates as r -> 0 ("left boundary"
!       in an equivalent cartesian grid representation), and in Spherical coordinates, as THETA -> 0 or PI. In these cases, the
!       coefficients of U_x (or U_y) in the generalized equation above (i.e., a2_x*a1 and b2_y*b1) are not ANALYTIC. The form
!       of the coordinate system can be specified using a "coord_flag" in the module "const_params". This program computes
!       the number of points in the spatial and time domains based on user supplied values of hx, hy & k, and computes the
!       "evloution" of the grid functions, Uji, for each "grid node" with time.
!
!       NOTE: IF A USER DEFINED SYSTEM IS CHOSEN, with NON-ANALYTIC {a1, a2, b1, b2}, THESE FUNCTIONS AND THEIR DERIVATIVES MUST
!       BE DEFINED CORRECTLY IN THE SUBROUTINES OF THE MODULE "pde_routines". CARE MUST ALSO BE TAKEN TO APPROPRIATELY
!       IMPLEMENT THE "INTERIOR" LOOP AND ALL THE "BOUNDARY CONDITION" LOOPS, IN THE SUBROUTINE "qldgts_coeff_rhs".
!
!       NOTE: For use with highly non-linear problems, a smoothing flag and parameter can be prescribed by the user, in the command
!       line, following the executable name. Either 1D or 2D Smoothing can be carried out using the simple Shuman filter, a low-pass
!       filter, that basically smooths out gradients in the domain at the end of each time step, at points (determined explicitly by
!       the user). IF SMOOTH FLAG IS NON-ZERO, THEN APPROPRIATE CHANGES NEED TO BE MADE BELOW, IN THE MAIN PROGRAM, TO MODIFY APPROPRIATE
!       GRID VALUES OF U.
!
!       This program computes  the number of points in the spatial and time domains based on user supplied values of hx, hy
!       & k, and computes the "evloution" of the grid functions, Uji, for each "grid node" with time. It allocates arrays,
!       prior to these computations. The boundary conditions are specified in separate functions, as are the forcing
!       function, f_rhs, as well as the exact solution (if known). IN THIS VERSION, boundary condition flags HAVE TO BE DEFINED in
!       the module CONST_PARAMS, but SPECIFIED in the MAIN PROGRAM. This allows for SEVERAL changes in Boundary Condition types,
!       with time [as when an Initial Neumann BC changes later to a Dirichlet BC]. Further details of boundary condition implementation
!       are presented under the subroutine "qldgts_coeff_rhs", above. The initial condition is specified under a separate function,
!       and is passed on to the "qldgts" subroutine. Time stepping is controlled in the main program, which outputs data at selected
!       time levels to the output files. The latter subroutine outputs the values of the grid function u, at each time step, in
!       a two dimensional array in y(j), and x(i). The plot data are printed out in separate output files to facilitate easy
!       post-processing, for each of  the time steps specified by the user. The number of time steps to be plotted or gridded
!       and the number of output files, along with their names can be changed by changing the "out" parameter array size, and
!       the array's elements, in the "const_params" module. EXTENSIVE checks have been added to all algorithms to improve
!       ERROR TRAPPING. The program allows the output of grid function and plot data at any resolution that the user chooses,
!       with the maximum ALLOWED resolution, of course, being hx*hy. If lower resolutions of hx and hy than allowed by the
!       machine array limitations are needed, the code can be modified later to completely eliminate storage in large arrays,
!       and instead, directly print out only the required plot data to output files. Evolution of maximum temperature is output to the
!       screen at a few specified time levels.
!-----------------------------------------------------------------------------------------------------------------------

        !-------------------------------------------------------------------------------------------------------------------------
        !       MAIN PROGRAM DECLARATIONS.
        !-------------------------------------------------------------------------------------------------------------------------

        REAL(KIND=rp), ALLOCATABLE, DIMENSION(:, :, :) :: u_errg, u_grid
        REAL(KIND=rp), ALLOCATABLE, DIMENSION(:, :) :: en, error_maxevol, u, u_evol, u_maxevol, u_minevol, u_xsnap, u_ysnap
        REAL(KIND=rp), ALLOCATABLE, DIMENSION(:) :: dn, srad, t_max_evol, x, y
        REAL(KIND=rp), DIMENSION( SIZE(grid_conv,1) ) :: u_conv
        REAL(KIND=rp), DIMENSION( SIZE(t_snap) ) :: u_grid_norm
        REAL(KIND=rp) :: dtdec, en_max, en_norm, global_max_error, global_max_error_u, global_max_u, global_max_u_error, &
                     & globalmax_u_norm, hx, hy, jr, k, lr, lsx, lsy, maxdec, max_error, max_error_u, max_u, max_u_error, min_u, &
                     & min_u_error, mr, nr, steps, t, t_evol, t_global_max, t_global_max_error, tm, t_out, t_steps, u_norm, x_steps, &
                     & y_steps, y1, y2
        INTEGER(KIND=ip), ALLOCATABLE, DIMENSION(:) :: i_grid, i_xsnap, j_grid, j_ysnap, nt_evol, nt_max_evol
```

```
INTEGER(KIND=ip), DIMENSION(SIZE(t_snap)) :: nt_snap
INTEGER(KIND=ip), DIMENSION(SIZE(grid_conv,1)) :: nt_gridconv, nx_gridconv, ny_gridconv
INTEGER(KIND=ip) :: alloc_error, bcout_flag, close_status, dealloc_error, decsteps, evol_count, i, i_evol, ifg1, ifg2, ifg3, &
                         & ifg4, i_max, i_max_global, int_res_flag_1, int_res_flag_2, int_smflag, i_tmax, i_tmax_global, i_tmin, &
                         & i_tmin_global, i_ysnap, j, j_evol, j_max, j_max_global, j_tmax, j_tmax_global, j_tmin, j_xsnap, l, lk, &
                         & lt, lx, ly, m, maxintt, mt0, n, n_c_r, ne, n_evol, nk, norm_flag, n_t, n_tout, nt_globalmax, &
                         & nt_globalmax_error, nt_xsnap, nt_ysnap, num_tmaxevol, n_xgrid, n_xout, n_xsnap, n_ygrid, n_yout, n_ysnap, &
                         & open_status, out_count, out_time_steps, p, s1, tevol_count, t_points, x_points, xsnap_count, &
                         & ysnap_count, y_points
CHARACTER(LEN=1) :: res_flag_1, res_flag_2, smflag
CHARACTER(LEN=6) :: smfact

!           Program Screen Header.
PRINT *, " "
PRINT *, "Program to compute the solution of a GENERAL NON-LINEAR, 2D, HEAT CONDUCTION EQUATION (in Cartesian/ "
PRINT *, "Cylindrical/Spherical coordinates), with general NON-LINEAR BOUNDARY CONDITIONS USING THE DELTA-FORM "
PRINT *, "OF QUASILINIARIZATION (NEWTON-KANTOROVICH PROCEDURE) IN CONJUNCTION WITH THE DELTA-FORM OF THE "
PRINT *, "DOUGLAS-GUNN TIME SPLITTING SCHEME (2-STEP). THIS CODE CAN ALSO BE USED FOR LINEAR PROBLEMS WITHOUT "
PRINT *, "ANY CHANGES TO THE CORE SUBROUTINES OF THIS IMPLEMENTATION. - by RAVI KANDA (November, 2002)."
PRINT *, "---------------------------------------------------------------------------------------------------"
PRINT *, " "

!------------------------------------------------------------------------------------------------------------------------------------
!           READ COMMAND LINE ARGUMENTS AND CHECK THAT ARGUMENT SIZES & VALUES ARE IN THE REQUIRED RANGES. COMPUTE DEPENDENT RUN VARIABLES.
!------------------------------------------------------------------------------------------------------------------------------------

!           SPATIAL Resolution Flag can be any number between 1 and 6. Each higher integer halves the grid spacing in space, in equal proportions.
!           EXAMPLE: If x_right = 1.0,
!           res_flag_1 = 1: int_res_flag_1 = 49: hx = hy = 0.1
!           res_flag_1 = 2: int_res_flag_2 = 50: hx = hy = 0.05
!           res_flag_1 = 3: int_res_flag_3 = 51: hx = hy = 0.025
!           .............. And so on ..................   ifg1 = IGETARG(1,res_flag_1,1)
IF (ifg1 < 0) THEN
            PRINT *, "Error Reading FIRST ARGUMENT: SPATIAL Resolution Flag! Check that the program executable is followed by FOUR "
            PRINT *, "arguments, SEPARATED BY SPACES. The first argument (1-9) specifies the SPATIAL resolution. The second argument (1-5)"
            PRINT *, "specifies the TEMPORAL resolution. The third argument (0-2) specifies SMOOTHING FLAG. The fourth (000000-999999) "
            PRINT *, "specifies the SMOOTHING FACTOR, if smoothing flag is NON-ZERO. The SPATIAL resolutions are determined as follows: "
            PRINT *, "------------------ FIRST ARGUMENT --------------------------"
            PRINT *, "FIRST ARGUMENT =  1: RES 1:           hx1 = hx_max,           hy1 = hy_max "
            PRINT *, "FIRST ARGUMENT =  2: RES 2:           hx2 = hx1/2,            hy2 = hy1/2 "
            PRINT *, "FIRST ARGUMENT =  3: RES 3:           hx3 = hx2/2,            hy3 = hy2/2 "
            PRINT *, "........AND SO ON"
            PRINT *, "------------------------------------------------------------"
            STOP
ELSE
            int_res_flag_1 = ICHAR(res_flag_1)
            IF ( ((int_res_flag_1 - 48) == 0) .OR. ((int_res_flag_1 - 48) > 9) ) THEN
                        PRINT *, "***** ERROR: Due to MACHINE LIMITATIONS, the FIRST argument must be between 1 and 6! EXITING PROGRAM."
                        STOP
            END IF
            hx = hx_max/(2.0_rp**(int_res_flag_1 - 49))
            hy = hy_max/(2.0_rp**(int_res_flag_1 - 49))
            !           Check that the output grid spacings are reasonable.
            IF (out_x_grid_spacing < hx) THEN
                        PRINT *, "WARNING: Grid output has been requested at a higher resolution than hx! Setting this to equal hx."
                        out_x_grid_spacing = hx
            END IF
            IF (out_y_grid_spacing < hy) THEN
                        PRINT *, "WARNING: Grid output has been requested at a higher resolution than hy! Setting this to equal hy."
                        out_y_grid_spacing = hy
            END IF
END IF

!           TEMPORAL Resolution Flag can be any number between 1 and 5. Each higher integer cuts the time resolution by a 10th.
!           res_flag_2 = 1: int_res_flag_2 = 49: k = MIN(hx,hy)                   = 0.1                For the above SPATIAL resolution example
!           res_flag_2 = 2: int_res_flag_2 = 50: k = MIN(hx,hy)/10      = 0.01      For the above SPATIAL resolution example
!           res_flag_2 = 3: int_res_flag_2 = 51: k = MIN(hx,hy)/100     = 0.001     For the above SPATIAL resolution example
!           res_flag_2 = 4: int_res_flag_2 = 52: k = MIN(hx,hy)/1000    = 0.0001    For the above SPATIAL resolution example
!           res_flag_2 = 5: int_res_flag_2 = 52: k = MIN(hx,hy)/10000   = 0.00001   For the above SPATIAL resolution example
ifg2 = IGETARG(2,res_flag_2,1)
IF (ifg2 < 0) THEN
            PRINT *, "Error Reading SECOND ARGUMENT: TEMPORAL Resolution Flag! Check that the program executable is followed by FOUR "
            PRINT *, "arguments, SEPARATED BY SPACES. The first argument (1-9) specifies the SPATIAL resolution. The second argument (1-5)"
            PRINT *, "specifies the TEMPORAL resolution. The third argument (0-2) specifies SMOOTHING FLAG. The fourth (000000-999999) "
            PRINT *, "specifies the SMOOTHING FACTOR, if smoothing flag is NON-ZERO. The TEMPORAL resolutions are determined as follows: "
            PRINT *, "------------------ SECOND ARGUMENT -------------------------"
            PRINT *, "SECOND ARGUMENT = 1: TIME RES 1: k = MIN(hx,hy)"
            PRINT *, "SECOND ARGUMENT = 2: TIME RES 2: k = MIN(hx,hy)/10"
            PRINT *, "SECOND ARGUMENT = 3: TIME RES 3: k = MIN(hx,hy)/100"
            PRINT *, "SECOND ARGUMENT = 4: TIME RES 4: k = MIN(hx,hy)/1000"
            PRINT *, "SECOND ARGUMENT = 5: TIME RES 5: k = MIN(hx,hy)/10000"
            PRINT *, "------------------------------------------------------------"
            STOP
ELSE
            int_res_flag_2 = ICHAR(res_flag_2)
            IF ( ((int_res_flag_2 - 48) == 0) .OR. ((int_res_flag_2 - 48) > 5) ) THEN
                        PRINT *, "***** ERROR: Due to MACHINE LIMITATIONS, the SECOND argument must be between 1 and 5! EXITING PROGRAM."
                        STOP
            END IF
            k  = ( MIN(hx, hy) )/( 10.0_rp**(int_res_flag_2 - 49) )
            !           Set Tolerance for Non-Linear Iterations.
            IF (linear_flag /= 1) THEN
                        quasi_epsilon = k*k*k
                        quasi_iterations = 25
            ELSE
                        quasi_epsilon = 1.0E30
                        quasi_iterations = 1
            END IF
END IF

!           SMOOTHING FLAG: THIRD ARGUMENT AFTER THE PROGRAM EXECUTABLE. For highly non-linear problems, this smooths out the gradients
!           in the domain, at the end of each time step, at points (determined explicitly by the user) using either 1D or 2D Shuman Filter.
!           IF THIS VALUE IS NON-ZERO, THEN APPROPRIATE CHANGES NEED TO BE MADE BELOW, IN THE MAIN PROGRAM, TO MODIFY THE APPROPRIATE
!           GRID VALUES OF U. Values for this flag are:
!           smooth_flag = 0, no smoothing
!           smooth_flag = 1, 1D smoothing
!           smooth_flag = 2, 2D smoothing.
ifg3 = IGETARG(3,smflag,1)
IF (ifg3 < 0) THEN
            PRINT *, "Error Reading THIRD ARGUMENT: SMOOTHING Flag! Check that the program executable is followed by FOUR arguments,"
            PRINT *, "SEPARATED BY SPACES. The first argument (1-9) specifies the SPATIAL resolution. The second argument (1-5) "
            PRINT *, "specifies the TEMPORAL resolution. The third argument (0-2) specifies SMOOTHING FLAG. The fourth (000000-999999) "
            PRINT *, "specifies the SMOOTHING FACTOR, if smoothing flag is NON-ZERO. The SMOOTHING FLAGS are as follows: "
            PRINT *, "------------------ THIRD ARGUMENT -------------------------"
```

```
            PRINT *, "THIRD ARGUMENT = smooth_flag = 0, No smoothing"
            PRINT *, "THIRD ARGUMENT = smooth_flag = 1, 1D smoothing"
            PRINT *, "THIRD ARGUMENT = smooth_flag = 2, 2D smoothing"
            PRINT *, "-------------------------------------------------------------"
            STOP
ELSE
            smooth_flag = ICHAR(smflag) - 48
            IF (smooth_flag > 2) THEN
                        PRINT *, "***** ERROR: This program can handle only 1D or 2D problems. The THIRD argument must be between 0 and 2!"
                        PRINT *, "EXITING PROGRAM."
                        STOP
            END IF
END IF

!           NOTE: If the THIRD argument, SMOOTHING FLAG, is NON-ZERO, then specify a degree of smoothing between {2 or 4} to 9999
!           as the last argument for the executable file, for 1D or 2D SMOOTHING, respectively.
ifg4 = IGETARG(4,smfact,6)
IF (ifg4 < 0) THEN
            PRINT *, "Error Reading FOURTH ARGUMENT: SMOOTHING FACTOR! Check that the program executable is followed by FOUR arguments, "
            PRINT *, "SEPARATED BY SPACES. The first argument (1-9) specifies the SPATIAL resolution. The second argument (1-5) "
            PRINT *, "specifies the TEMPORAL resolution. The third argument (0-2) specifies SMOOTHING FLAG. The fourth (000000-999999) "
            PRINT *, "specifies the SMOOTHING FACTOR, if smoothing flag is NON-ZERO. SMOOTHING FACTOR has a range of 0-999999, and "
            PRINT *, "MUST BE 6 characters long. FORMAT: 000002, 000038, 000125, 001525, 085792, & 850000."
            PRINT *, "IF NO SMOOTHING IS NEEDED, make sure that the THIRD ARGUMENT, SMOOTHING FLAG, is ZERO, & SET this value to 000000!"
            PRINT *, " "
            PRINT *, "Check also that the program executable is followed by THREE 1-digit arguments, SEPARATED BY SPACES, prior to this one."
            PRINT *, "-------------------------------------------------------------------------------------------------------------"
            STOP
ELSE IF (ifg4 < 6) THEN
            PRINT *, "Error Reading FOURTH Input! This argument specifies the SMOOTHING FACTOR FOR NON-LINEAR TEMPERATURE CORRECTIONS."
            PRINT *, "This argument has a range of 0-999999, and MUST be 6 characters long. FORMAT: "
            PRINT *, "000002, 000038, 000125, 001525, 085792, & 850000."
            PRINT *, "IF NO SMOOTHING IS NEEDED, make sure that the THIRD ARGUMENT, SMOOTHING FLAG, is 0 (ZERO), & SET this value to 000000!"
            PRINT *, " "
            PRINT *, "Check also that the program executable is followed by THREE 1-digit, and ONE 6-digit arguments, SEPARATED BY SPACES."
            PRINT *, "-------------------------------------------------------------------------------------------------------------"
            STOP
ELSE
            smooth_factor =     (ICHAR(smfact(1:1)) - 48)*100000.0_rp + (ICHAR(smfact(2:2)) - 48)*10000.0_rp &
                                & + (ICHAR(smfact(3:3)) - 48)*1000.0_rp   + (ICHAR(smfact(4:4)) - 48)*100.0_rp &
                                & + (ICHAR(smfact(5:5)) - 48)*10.0_rp     + (ICHAR(smfact(6:6)) - 48)*1.0_rp
            IF ( (smooth_flag == 0) .AND. (smooth_factor > 0) ) THEN
                        PRINT *, "SMOOTHING FLAG = 0: For NO smoothing, SMOOTHING FACTOR MUST BE 0000 (ZERO)!"
                        PRINT *, "EXITING PROGRAM."
                        STOP
            END IF
END IF
!-------------------------------------------------------------------------------------------------------------------------------
!           END OF READING COMMAND LINE ARGUMENTS AND INPUT ERROR CHECKS.
!-------------------------------------------------------------------------------------------------------------------------------

!-------------------------------------------------------------------------------------------------------------------------------
!           HEADER INFORMATION FOR OUTPUT FILES
!-------------------------------------------------------------------------------------------------------------------------------

!           Connect to output files, and type in the headings. Report opening errors.
DO m = 1, SIZE(out)
            OPEN (UNIT=out(m), FILE=outfile(m), STATUS="REPLACE", IOSTAT=open_status)
            DO i = 1,3
                        IF (open_status==0) THEN
                                    EXIT        ! Exit on successful opening/connection
                        ELSE
                                    PRINT *, "Unable to open file - ", outfile(m), ". Trying again."
                        ENDIF
                        PRINT *, outfile(m), " cannot be opened! Check your source directory contents."
                        STOP
            END DO
            WRITE (UNIT=out(m), FMT='("% Program to compute the solution evolution of a GENERALIZED NON-LINEAR, 2D "/&
                                    & "% HEAT CONDUCTION PDE, with GENERALIZED NON-LINEAR BCs, using the DELTA-FORM of"/&
                                    & "% QUASILINEARIZATION (NEWTON-KANTOROVICH PROCEDURE) WITH DOUGLAS-GUNN TIME&
                                    & SPLITTING SCHEME:"/"%  - by RAVI KANDA (July, 2002).")')
            WRITE (UNIT=out(m), FMT='("% Precision: KIND = ",I2," for FORTRAN90 Compiler v2.4 for HP-UX 11i on HP-SuperDome.")') rp
            WRITE (UNIT=out(m), FMT='("% ---------------------------------------------------------------------------"/"% ")')
            WRITE (UNIT=out(m), FMT='("X-Limits: (x_left, x_right) = (",ES14.8,",",ES14.8,")")') x_left, x_right
            WRITE (UNIT=out(m), FMT='("Y-Limits: (y_bottom, y_top) = (",ES14.8,",",ES14.8,")")') y_bottom, y_top
            WRITE (UNIT=out(m), FMT='("t-Limits: (t_initial, t_final) = (",ES14.8,",",ES14.8,")")') t_initial, t_final
            WRITE (UNIT=out(m), FMT='("The value of x-step, hx = ", ES14.8)') hx
            WRITE (UNIT=out(m), FMT='("The value of y-step, hy = ", ES14.8)') hy
            WRITE (UNIT=out(m), FMT='("The value of t-step,  k = ", ES14.8)') k
            WRITE (UNIT=out(m), FMT='("% ---------------------------------------------------------------------------")')
            IF (linear_flag /= 1) THEN
                        WRITE (UNIT=out(m), FMT='("% This problem is indicated to be NON-LINEAR. Newton-Kantorovich "/&
                                    & "% iterations will be performed up to a convergence tolerance of ", ES12.6,"."/&
                                    & "% The maximum number of iterations, max_iter, was set to: ",I2,".")') quasi_epsilon,quasi_iterations
                        WRITE (UNIT=out(m), FMT='("% ---------------------------------------------------------------------")')
                        IF (smooth_flag == 0) THEN
                                    WRITE (UNIT=out(m), FMT='("% SMOOTHING FLAG = 0: NO SMOOTHING WILL BE PERFORMED.")')
                                    WRITE (UNIT=out(m), FMT='("% ---------------------------------------------------")')
                        ELSE IF (smooth_flag == 1) THEN
                                    WRITE (UNIT=out(m), FMT='("% SMOOTHING FLAG = 1: 1D SMOOTHING WILL BE PERFORMED, with SMOOTHING FACTOR = ", &
                                                                                                & F7.2,".")') smooth_factor
                                    WRITE (UNIT=out(m), FMT='("% -------------------------------------------------------------------------------")')
                        ELSE
                                    WRITE (UNIT=out(m), FMT='("% SMOOTHING FLAG = 2: 2D SMOOTHING WILL BE PERFORMED, with SMOOTHING FACTOR = ", &
                                                                                                & F7.2,".")') smooth_factor
                                    WRITE (UNIT=out(m), FMT='("% -------------------------------------------------------------------------------")')
                        END IF
            ELSE
                        WRITE (UNIT=out(m), FMT='("% This problem is indicated to be LINEAR. No iterations need to be "/&
                                                & "% performed. Douglas-Gunn Time splitting will be directly implemented.")')
                        WRITE (UNIT=out(m), FMT='("% ---------------------------------------------------------------------")')
            END IF
            IF (coord_flag == 1) THEN
                        WRITE (UNIT=out(m), FMT='("% COORDINATE SYSTEM: CARTESIAN.")')
                        WRITE (UNIT=out(m), FMT='("% -------------------------------")')
            ELSE IF (coord_flag == 2) THEN
                        WRITE (UNIT=out(m), FMT='("% COORDINATE SYSTEM: CYLINDRICAL.")')
                        WRITE (UNIT=out(m), FMT='("% -------------------------------")')
            ELSE
                        WRITE (UNIT=out(m), FMT='("% COORDINATE SYSTEM: SPHERICAL.")')
                        WRITE (UNIT=out(m), FMT='("% -------------------------------")')
            END IF
END DO
```

```
IF (exact_sol_flag == 0) THEN                  ! No exact solution available.
           IF (linear_flag /= 1) THEN
                      PRINT *, "Exact solution not available for Non-Linear Problem. Using Error Estimates."
                      WRITE (UNIT=out(2), FMT='("% NON-LINEAR PDE: Exact solution not available. Using Error Estimates.")')
                      WRITE (UNIT=out(2), FMT='("% ---------------------------------------------------------------------")')
           ELSE
                      PRINT *, "Exact solution not available for Linear Problem. No Error Estimate available."
                      WRITE (UNIT=out(2), FMT='("% LINEAR PDE: Exact solution not available. No Error Estimates Available.")')
                      WRITE (UNIT=out(2), FMT='("% LINEAR PDE: ERRORS WILL BE ARBITRARILY SET TO 1000.0_rp!")')
                      WRITE (UNIT=out(2), FMT='("% ---------------------------------------------------------------------")')
           END IF
END IF

!          Compute the RADIAN MEASURE of the RADIUS OF ASPERITY CONTACT AREA. OUTPUT FAULT DATA TO ALL FILES & SCREEN. This cannot be computed
!          under FAULT PARAMS because the Parameter statement does not accept any intrinsic function evaluations.
y0 = ATAN(rc_by_r0)

!          Print out all the Fault Parameters Being used for this run:
DO m = 1, SIZE(out)
           WRITE (UNIT=out(m), FMT='("Ambient Temperature,                                 U0 = 300 K.")')
           WRITE (UNIT=out(m), FMT='("Asperity Radius,                                     r0 = ", F6.3, " m.")') x_right
           WRITE (UNIT=out(m), FMT='("Young''s Modulus,                                      E = ", F6.2, " GPa.")') e_y
           WRITE (UNIT=out(m), FMT='("Poisson''s Ratio,                                     nu = ", F4.2, " (dimensionless).")') nu_ps
           WRITE (UNIT=out(m), FMT='("Coefficient of Friction,                             mu = ", F4.2, " (dimensionless).")') mu
           WRITE (UNIT=out(m), FMT='("Density of asperity material,                       rho = ", F7.2, " kg/m**3.")') rho
           WRITE (UNIT=out(m), FMT='("Ambient average shear stress,                       TAU = ", ES8.2," Pa.")') tau
           WRITE (UNIT=out(m), FMT='("Asperity slip velocity,                               U = ", F6.3, " m/sec.")') slip_v
           WRITE (UNIT=out(m), FMT='("The ratio,                                        rc/r0 = ", ES14.8," (dimensionless).")') rc_by_r0
           WRITE (UNIT=out(m), FMT='("Maximum radius of circular asperity contact area,    rc = ", ES9.3, " m.")') rc
           WRITE (UNIT=out(m), FMT='("Asperity slip duration,                              T0 = ", ES9.3, " sec.")') t0
           WRITE (UNIT=out(m), FMT='("Maximum Asperity contact,                       THETA_0 = ", F10.8," Radians.")') y0
           IF (linear_flag == 1) THEN
              WRITE (UNIT=out(m), FMT='("Thermal Conductivity,                                 kt = ", ES8.2, " W/(m**2.K).")') kt_const
              WRITE (UNIT=out(m), FMT='("Specific Heat,                                        Cp = ", ES8.2, " J/kg")') cp_const
              WRITE (UNIT=out(m), FMT='("Thermal Conductivity,                              KAPPA = ", ES8.2, " m**2/sec.")') &
                      &
kt_const/(rho*cp_const)
           ELSE
              WRITE (UNIT=out(m), FMT='("Specific Heat, Cp & Coeff. of Thermal Conductivity, k are NON-LINEAR FUNCTIONS OF TEMPERATURE.")')
           END IF
           WRITE (UNIT=out(m), FMT='("% -----------------------------------------------------------------------------------------")')
END DO
PRINT *, "X-Limits: (x_left, x_right)    = (",x_left,",",x_right,")"
PRINT *, "Y-Limits: (y_bottom, y_top)    = (",y_bottom,",",y_top,")"
PRINT *, "t-Limits: (t_initial, t_final) = (",t_initial,",",t_final,")"
PRINT *, "The value of x-step, hx        = ", hx
PRINT *, "The value of y-step, hy        = ", hy
PRINT *, "The value of t-step,  k        = ", k
PRINT *, "Smoothing Flag                 = ", smooth_flag
PRINT *, "Smoothing Factor               = ", smooth_factor
PRINT *, "---------------------------------------------------------------------------------"
PRINT *, "Ambient Temperature,                             U0 =  300 K."
PRINT *, "Asperity Radius                                  r0 = ", x_right, " m."
PRINT *, "Young''s Modulus,                                  E = ", e_y,     " GPa."
PRINT *, "Poisson's Ratio,                                 nu = ", nu_ps,   " (dimensionless)."
PRINT *, "Coefficient of Friction,                         mu = ", mu,      " (dimensionless)."
PRINT *, "Density of asperity material,                   rho = ", rho,     " kg/m**3."
PRINT *, "Ambient average shear stress,                   TAU = ", tau,     " Pa."
PRINT *, "Asperity slip velocity,                           U = ", slip_v,  " m/sec."
PRINT *, "The ratio,                                    rc/r0 = ", rc_by_r0," (dimensionless)."
PRINT *, "Maximum radius of circular asperity contact area, rc = ", rc,      " m."
PRINT *, "Asperity slip duration,                          T0 = ", t0,      " sec."
PRINT *, "Maximum Asperity contact                    THETA_0 = ", y0,      " Radians."
IF (linear_flag == 1) THEN
           PRINT *, "Thermal Conductivity,                           kt = ", kt_const, " W/(m**2.K)."
           PRINT *, "Specific Heat,                                  Cp = ", cp_const, " J/kg"
           PRINT *, "Thermal Conductivity,                        KAPPA = ", kt_const/(rho*cp_const), " m**2/sec."
ELSE
           PRINT *, "Specific Heat, Cp & Coeff. of Thermal Conductivity, k are NON-LINEAR FUNCTIONS OF TEMPERATURE."
END IF
PRINT *, "---------------------------------------------------------------------------------"
IF (t0 < k) THEN
           PRINT *, " "
           PRINT *, "WARNING: T0, the asperity separation time, is LESS THAN THE TEMPORAL RESOLUTION FOR THIS RUN!!"
           PRINT *, " "
END IF
IF (t0 > t_final) THEN
           PRINT *, " "
           PRINT *, "WARNING: TIME RANGE for this run is LESS THAN the asperity separation time, T0!!"
           PRINT *, " "
END IF
!-------------------------------------------------------------------------------------------------------------------------------------
!          END OF HEADER INFORMATION FOR OUTPUT FILES & SCREEN
!-------------------------------------------------------------------------------------------------------------------------------------

!-------------------------------------------------------------------------------------------------------------------------------------
!          OUTPUT FILE PARAMETERS:
!-------------------------------------------------------------------------------------------------------------------------------------

!          OUTPUT FILES #1 & 2: GRID FUNCTIONS at times corresponding to those defined in the array t_snap in the MODULE CONST_PARAMS.
!          Convert time levels for outputting GRID FUNCTIONS and ERRORS into time step numbers for the given value of k, the step size.
!          Also compute the output grid size, and the grid indices for outputting to these files, given hx and hy.
DO n = 1, SIZE(t_snap)
           nr = (t_snap(n) - t_initial)/k + 1.0_rp
           IF ( ABS(nr-INT(nr)) > 0.5_rp ) THEN
                      nt_snap(n) = INT(nr) + 1

           ELSE
                      nt_snap(n) = INT(nr)
           END IF
END DO
lsx = (x_right - x_left)/(out_x_grid_spacing) + 1.0_rp
IF ( ABS(lsx-INT(lsx)) > 0.5_rp ) THEN
           n_xgrid = INT(lsx) + 1
ELSE
           n_xgrid = INT(lsx)
END IF
lsy = (y_top - y_bottom)/(out_y_grid_spacing) + 1.0_rp
IF ( ABS(lsy-INT(lsy)) > 0.5_rp ) THEN
           n_ygrid = INT(lsy) + 1
ELSE
           n_ygrid = INT(lsy)
END IF
```

```
!           Allocate grid 1D index arrays i_grid, and j_grid. At the same time, allocate the 3D arrays, u_grid and u_errg.
ALLOCATE (i_grid(n_xgrid), j_grid(n_ygrid), u_errg(n_ygrid,n_xgrid,SIZE(t_snap)), u_grid(n_ygrid,n_xgrid,SIZE(t_snap)), &
            & STAT=alloc_error)
IF (alloc_error /=0) THEN
            PRINT *, "ERROR: Some/All GRID arrays could not be allocated! Not enough storage space."
            STOP
ELSE
            PRINT *, "ALL grid ARRAYS SUCCESSFULLY ALLOCATED."
END IF
y1 = x_left
DO i = 1,n_xgrid
            lr = (y1 - x_left)/hx + 1.0_rp
            IF ( ABS(lr-INT(lr)) > 0.5_rp ) THEN
                        i_grid(i) = INT(lr) + 1
            ELSE
                        i_grid(i) = INT(lr)
            END IF
            y1 = y1 + out_x_grid_spacing
END DO
y1 = y_bottom
DO j = 1,n_ygrid
            mr = (y1 - y_bottom)/hy + 1.0_rp
            IF ( ABS(mr-INT(mr)) > 0.5_rp ) THEN
                        j_grid(j) = INT(mr) + 1
            ELSE
                        j_grid(j) = INT(mr)
            END IF
            y1 = y1 + out_y_grid_spacing
END DO
!--------------------------------------------------------

!           OUTPUT FILE #3: SNAPSHOTS.
!           OUTPUT FILE #3a: SNAPSHOT OF PROFILE ALONG A LINE PARALLEL TO x-axis - Convert to time level, i, in t(i):
nr = (t_xsnap - t_initial)/k + 1.0_rp
IF ( ABS(nr-INT(nr)) > 0.5_rp ) THEN
            nt_xsnap = INT(nr) + 1
ELSE
            nt_xsnap = INT(nr)
END IF
!           Compute y-index of snap along x-axis:
mr = (y_xsnap - y_bottom)/hy + 1.0_rp
IF ( ABS(mr-INT(mr)) > 0.5_rp ) THEN
            j_xsnap = INT(mr) + 1
ELSE
            j_xsnap = INT(mr)
END IF
!           Also compute/specify the number of x spatial steps for output generation. ALLOCATE i_xsnap array, along with u_xsnap.
!           Compute the index contents of i_xsnap:
n_xsnap = n_xgrid
ALLOCATE (i_xsnap(n_xsnap), u_xsnap(n_xsnap,2), STAT=alloc_error)
IF (alloc_error /=0) THEN
            PRINT *, "ERROR: All/Some XSNAP arrays could not be allocated! Not enough storage space."
            STOP
ELSE
            PRINT *, "ALL xsnap ARRAYS SUCCESSFULLY ALLOCATED."
END IF
y1 = x_left
DO i = 1, n_xsnap
            lr = (y1 - x_left)/hx + 1.0_rp
            IF ( ABS(lr-INT(lr)) > 0.5_rp ) THEN
                        i_xsnap(i) = INT(lr) + 1
            ELSE
                        i_xsnap(i) = INT(lr)
            END IF
            y1 = y1 + out_x_grid_spacing
END DO
!--------------------------------------------------------

!           OUTPUT FILE #3b: SNAPSHOT OF PROFILE ALONG A LINE PARALLEL TO y-axis - Convert to time level, i, in t(i):
nr = (t_ysnap - t_initial)/k + 1.0_rp
IF ( ABS(nr-INT(nr)) > 0.5_rp ) THEN
            nt_ysnap = INT(nr) + 1
ELSE
            nt_ysnap = INT(nr)
END IF
!           Compute x-index of snap along y-axis:
lr = (x_ysnap - x_left)/hx + 1.0_rp
IF ( ABS(lr-INT(lr)) > 0.5_rp ) THEN
            i_ysnap = INT(lr) + 1
ELSE
            i_ysnap = INT(lr)
END IF
!           Also compute/specify the number of y spatial steps for output generation. ALLOCATE j_ysnap array, along with u_ysnap.
!           Compute the index contents of j_ysnap:
n_ysnap = n_ygrid
ALLOCATE (j_ysnap(n_ysnap), u_ysnap(n_ysnap,2), STAT=alloc_error)
IF (alloc_error /=0) THEN
            PRINT *, "ERROR: All/Some YSNAP arrays could not be allocated! Not enough storage space."
            STOP
ELSE
            PRINT *, "ALL ysnap ARRAYS SUCCESSFULLY ALLOCATED."
END IF
y1 = y_bottom
DO j = 1, n_ysnap
            mr = (y1 - y_bottom)/hy + 1.0_rp
            IF ( ABS(mr-INT(mr)) > 0.5_rp ) THEN
                        j_ysnap(j) = INT(mr) + 1
            ELSE
                        j_ysnap(j) = INT(mr)
            END IF
            y1 = y1 + out_y_grid_spacing
END DO
!--------------------------------------------------------

!           OUTPUT FILE #4: TEMPERATURE & ERROR EVOLUTION OUTPUT.
!           Compute the x- and y- indices for the point at which grid function temporal evolution is being output. Also, compute the number of
!           time evolution output steps based on the value for t_evol_spacing defined in the MODULE
!           CONST_PARAMS:
lr = (x_time - x_left)/hx + 1.0_rp
IF ( ABS(lr-INT(lr)) > 0.5_rp ) THEN
            i_evol = INT(lr) + 1
ELSE
            i_evol = INT(lr)
END IF
```

```
mr = (y_time - y_bottom)/hy + 1.0_rp
IF ( ABS(mr-INT(mr)) > 0.5_rp ) THEN
            j_evol = INT(mr) + 1
ELSE
            j_evol = INT(mr)
END IF
nr = (t_final - t_initial)/t_evol_spacing + 1.0_rp
IF ( ABS(nr-INT(nr)) > 0.5_rp ) THEN
            n_evol = INT(nr) + 1
ELSE
            n_evol = INT(nr)
END IF
!           Allocate the time evolution index array, nt_evol, as well as u_evol. Compute the index elements of nt_evol.
ALLOCATE (nt_evol(n_evol), u_evol(n_evol,2), STAT=alloc_error)
IF (alloc_error /=0) THEN
            PRINT *, "ERROR: All/Some T_EVOL arrays could not be allocated! Not enough storage space."
            STOP
ELSE
            PRINT *, "ALL t_evol ARRAYS SUCCESSFULLY ALLOCATED."
END IF
t_evol = t_initial
DO m = 1, n_evol
            nr = (t_evol - t_initial)/k + 1.0_rp
            IF ( ABS(nr-INT(nr)) > 0.5_rp ) THEN
                        nt_evol(m) = INT(nr) + 1
            ELSE
                        nt_evol(m) = INT(nr)
            END IF
            t_evol = t_evol + t_evol_spacing
END DO
!           MAX. TEMPERATURE & ERROR EVOLUTION.
!           Determine Maximum Temperature (and Maximum Error, if applicable) Evolution time levels. The time levels are distributed at
!           equidistant points on a log-scale - i.e., appropriate points in the decades containing the time step, k, and the final time,
!           t_final, and 10 points in each of the intermediate decades.
lt = INT(LOG10(t_final))
lk = INT(LOG10(k))
maxdec = 10.0_rp**( lt )
IF (t_final == maxdec) THEN                                                         ! Determine the number of terms in the decade containing t_final
            maxintt = 0
ELSE
            maxintt= INT( t_final/maxdec )
END IF
dtdec = 10.0_rp**( lk )

!     Determine the number of terms in the decade containing time step size, k: Exclude the last value, which falls into the next higher decade.
nk = INT(dtdec/k) - 1

!           For the intermediate time range (between the decades contiaining t_final and k), each decade will have 9 points.
decsteps = lt - lk
num_tmaxevol = 1 + nk + 9*decsteps + maxintt + 1

!           Allocate all TEMPERATURE EVOLUTION arrays.
ALLOCATE (error_maxevol(num_tmaxevol,7), t_max_evol(num_tmaxevol), nt_max_evol(num_tmaxevol), u_maxevol(num_tmaxevol,7), &
                                                                                                                        & u_minevol(num_tmaxevol,7), STAT=alloc_error)
IF (alloc_error /=0) THEN
            PRINT *, "ERROR: All/Some TEMPERATURE EVOLUTION arrays could not be allocated! Not enough storage space."
            STOP
ELSE
            PRINT *, "ALL temperature evolution ARRAYS SUCCESSFULLY ALLOCATED."
END IF

!           Fill the t_max_evol array with appropriate output time levels.
m = 0
i = 1
DO j = 1, num_tmaxevol
            IF (j <= 1+nk) THEN
                        IF (j == 1) THEN
                                    t_max_evol(j) = t_initial
                        ELSE
                                    t_max_evol(j) = (j-1)*k                         ! If nk = 0, there are no terms in this block.
                        END IF
            ELSE IF (j <= 1+nk+9*decsteps ) THEN
                                    t_max_evol(j) = i*dtdec*(10.0_rp**m)
                                    i = i + 1
                                    IF (i > 9) THEN
                                                m = m + 1
                                                i = 1
                                    END IF


            ELSE IF (maxintt /= 0) THEN                                 ! If maxintt = 0, t_final corresponds to a decadal "margin", then no terms here.
                        IF ( j <= (num_tmaxevol - 1) ) THEN
                                    t_max_evol(j) = ( j - (1 + nk + 9*decsteps) )*maxdec
                        END IF
            ELSE                                                                                                    ! j = num_tmaxevol
                        t_max_evol(j) = t_final
            END IF
END DO

!           Now convert the maximum temperature evolution time levels to the corresponding integral time steps, for the given k.
DO m = 1, num_tmaxevol
            tm = (t_max_evol(m) - t_initial)/k + 1.0_rp
            IF ( ABS(tm-INT(tm)) > 0.5_rp ) THEN
                        nt_max_evol(m) = INT(tm) + 1
            ELSE
                        nt_max_evol(m) = INT(tm)
            END IF
END DO
!---------------------------------------------------------

!           OUTPUT FILE #5: POINT GRID CONVERGENCE TEST LOCATIONS - 8 points, at different space & time coordinates:
!           DEFINE THIS ARRAY IN THE MODULE "const_params" WITH THE REQUIRED DIMENSION! Convert grid convergence time levels
!           into time levels for the given value of k, the step size. Also, print out all the grid convergence data points.
DO n = 1, SIZE(grid_conv,1)
            lr = (grid_conv(n,1) - x_left)/hx + 1.0_rp
            IF ( ABS(lr-INT(lr)) > 0.5_rp ) THEN
                        nx_gridconv(n) = INT(lr) + 1
            ELSE
                        nx_gridconv(n) = INT(lr)
            END IF
            mr = (grid_conv(n,2) - y_bottom)/hy + 1.0_rp
```

```
            IF ( ABS(mr-INT(mr)) > 0.5_rp ) THEN
                        ny_gridconv(n) = INT(mr) + 1
            ELSE
                        ny_gridconv(n) = INT(mr)
            END IF
            nr = (grid_conv(n,3) - t_initial)/k + 1.0_rp
            IF ( ABS(nr-INT(nr)) > 0.5_rp ) THEN
                        nt_gridconv(n) = INT(nr) + 1
            ELSE
                        nt_gridconv(n) = INT(nr)
            END IF
            WRITE (UNIT=out(5), FMT='(1X,"x = ",F4.2,1X,"y = ",F4.2,1X,"t = ",F4.2)') grid_conv(n,1), grid_conv(n,2), grid_conv(n,3)
END DO
WRITE (UNIT=out(5), FMT='(/"----------------------------------------------------------------------------")')
!--------------------------------------------------------

!-------------------------------------------------------------------------------------------------------------------------------------
!           COMPUTE/INITIALIZE RUN PARAMETERS, AND ALLOCATE ALL OTHER ARRAYS NEEDED FOR THIS RUN:
!-------------------------------------------------------------------------------------------------------------------------------------

!           Calculate the Number of Points in the space and time domains. Check that the number of points do not
!           exceed machine limitations.
x_steps = 1.0_rp + (x_right - x_left)/hx
y_steps = 1.0_rp + (y_top - y_bottom)/hy
t_steps = 1.0_rp + (t_final - t_initial)/k
IF ( ABS(x_steps-INT(x_steps)) > 0.5_rp ) THEN
            x_points = INT(x_steps) + 1
ELSE
            x_points = INT(x_steps)
END IF
IF (x_points > max_points) THEN
            PRINT*, "******** ERROR: Number of x grid points exceeds maximum allowed grid points, ", max_points
            PRINT*, "ABORTING PROGRAM!"
            STOP
END IF
IF ( ABS(y_steps-INT(y_steps)) > 0.5_rp ) THEN
            y_points = INT(y_steps) + 1
ELSE
            y_points = INT(y_steps)
END IF
IF (y_points > max_points) THEN
            PRINT*, "******** ERROR: Number of y grid points exceeds maximum allowed grid points, ", max_points
            PRINT*, "ABORTING PROGRAM!"
            STOP
END IF
!           Unlike the x and y grid points above, "t_points" has a maximum value determined only by the machine DO LOOP counter limit.*****
IF ( ABS(t_steps-INT(t_steps)) > 0.5_rp ) THEN
            t_points = INT(t_steps) + 1
ELSE
            t_points = INT(t_steps)
END IF

n_c_r = x_points*y_points           ! This is used in for defining the coeff & rhs arrays in the ALLOCATE statement below.

!           Allocate arrays and vectors. Arrays coeff, NSu_m, Nu_m, rhs, rs, u_n, u_old are used in other modules, and MUST BE
!           DEFINED GLOBALLY, in the module "const_params" above.
IF (linear_flag /= 1) THEN
            ALLOCATE (coeff(n_c_r,3), dn(quasi_iterations), en(y_points,x_points), NSu_m(2,x_points), Nu_m(y_points,x_points), &
                        & srad(quasi_iterations), rhs(n_c_r), rs(n_c_r), u(y_points,x_points), u_n(y_points,x_points), &
                        & u_old(y_points,x_points), x(x_points), y(y_points), STAT=alloc_error)
ELSE
            ALLOCATE (coeff(n_c_r,3), en(y_points,x_points), NSu_m(2,x_points), Nu_m(y_points,x_points), rhs(n_c_r), &
                        & u(y_points,x_points), u_n(y_points,x_points), x(x_points), y(y_points), STAT=alloc_error)
END IF
IF (alloc_error /=0) THEN
            PRINT *, "ERROR: All/Some NON-OUTPUT-FILE arrays could not be allocated! Not enough storage space."
            STOP
ELSE
            PRINT *, "ALL non-output-file ARRAYS SUCCESSFULLY ALLOCATED."
            PRINT *, "-----------------------------------------------------------------------------------------------------------"
            PRINT *, " "
END IF


!           Initialize all arrays that are not being used in the MAIN Program.
coeff = 0.0_rp
Nu_m = 0.0_rp                       ! This array is used for the Non-Linear/Linear Functional in "qlindgts_coeff_rhs" routine.
NSu_m = 0.0_rp                      ! This array is used for the bottom boundary Non-Linear/Linear Functional in "qlindgts_coeff_rhs" routine.
rhs = 0.0_rp
rs = 0.0_rp
u_n = 0.0_rp
u_old = 0.0_rp

!           Compute the spatial grid coordinate vectors, X & Y, and assign the initial time:
x = (/ ((x_left    + (i-1)*hx), i = 1, x_points) /)
y = (/ ((y_bottom  + (i-1)*hy), i = 1, y_points) /)

!           INITIALIZE time and other flags/counters.
t = t_initial
t_evol = t_initial
ne = 1
out_count = 1
global_max_error = 0.0_rp
global_max_u = 0.0_rp
evol_count = 1                                  ! Screen output time level index
bcout_flag = 0                                  ! For outputting BC types each time there is a change.
norm_flag = 1                                   ! For saving U_norm each time global maximum temperature is updated.
xsnap_count = 1                                 ! Count for output level for FILE #3a
ysnap_count = 1                                 ! Count for output level for FILE #3b
tevol_count = 1                                 ! Count for output level for FILE #4

!---------------------------------------------------------------------------------------------------------------------------------------------
!     MAIN COMPUTAIONAL LOOP: Contains BC TYPE definitions as a function of time, if applicable. OUTPUT DATA IS ALSO STORED WITHIN THIS LOOP.
!---------------------------------------------------------------------------------------------------------------------------------------------

!           START THE TIME STEPPING LOOP, with m=1 as the initial time.
DO n_t = 1, t_points
            IF (n_t == 1) THEN
            !           Compute the initial values for the problem, and output them.
                        u = f_initial(x,y)
                        max_u = 0.0_rp
                        min_u = 1/epsilon
```

```fortran
		DO j = 1, y_points
			DO i = 1, x_points
				IF (ABS(u(j,i)) > max_u) THEN
					max_u       = ABS(u(j,i))
					max_u_error = en(j,i)
					i_tmax      = i
					j_tmax      = j
				END IF
				IF (ABS(u(j,i)) < min_u) THEN
					min_u       = ABS(u(j,i))
					min_u_error = en(j,i)
					i_tmin      = i
					j_tmin      = j
				END IF
				IF (ABS(u(j,i)) > global_max_u) THEN
					global_max_u       = ABS(u(j,i))
					global_max_u_error = en(j,i)
					i_tmax_global      = i
					j_tmax_global      = j
					t_global_max       = t
					nt_globalmax       = n_t
				END IF
			END DO
		END DO

!		DEFINE THE INITIAL BC FLAGS.
!		PDE BOUNDARY CONDITION FLAGS: These are being moved here from CONST_PARAMS to offer flexibility in terms of
!		time varying BC TYPES (for instance a change from an initial Neumann BC to a subsequent Dirichlet BC. The BC
!		type change can happen any number of times, and the case-specific handling of these changes will be dealt with
!		different DO LOOPS for each BC set.
!		NOTATION FOR BOUNDARY CONDITION FLAGS:		0 for DIRICHLET {i.e., Bbc(U) = B2bc(U)},
!								1 for NEUMANN {i.e., Bbc(U) = U_x*B1bc(U)},
!								2 for ROBIN {i.e., Bbc(u) = U_x*B1bc(U)}.
!		All BCs are represented in the generalized non-linear forms encountered in heat conduction problems:
!		Bbc(U) = U_x*B1bc(U)+ B2bc(U) or U_y*B1bc(U)+ B2bc(U). This form can be used to represent either NON-LINEAR or
!		LINEAR BCs. PROVIDE ALL BOUNDARY OPERATORS, B, in this SPLIT FORM, using separate functions for B1 and B2, for
!		EACH BC. These classifications and their implementations are discussed under the separate functions in the module
!		"pde_routines", below, and ESPECIALLY UNDER THE SUBROUTINE "qldgts_coeff_rhs", where they are used:

		left_bc_flag  = 1
		right_bc_flag  = 1
		bottom_bc_flag  = 1
		top_bc_flag  = 1

!		OPTIONAL Linear Robin Parameters, ALPHA_x & ALPHA_y for each of the two directions. Eg., in: L = U_x + alpha_x * U
!		alpha_x = 0.0_rp
!		alpha_y = 0.0_rp
!
!		BOUNDARY CONDITION LINEARITY FLAGS: 1 if linear, 0 if non-linear.
!		These will affect the forms and values of the corresponding boundary condition functionals (lbc1, bbc1, tbc2, etc.)
!		below. If any of these flags is 0 (non-linear BC) then the forms of these functionals have to be defined in the
!		respective subroutines in MODULE "pde_routines":

		left_lin_flag = 1
		right_lin_flag = 0
		bottom_lin_flag = 1
		top_lin_flag = 1

!		Confirm ALL BC types for this time range.
		DO m = 1, SIZE(out)
			WRITE (UNIT=out(m), FMT='("% For time <= To = ",ES8.2, ": ")', ADVANCE="NO") t0
			IF (left_lin_flag == 1) THEN
				IF (left_bc_flag == 0) THEN
					WRITE (UNIT=out(m), FMT='("LEFT BC = Linear Dirichlet; ")', ADVANCE="NO")
				ELSE IF (left_bc_flag == 1) THEN
					WRITE (UNIT=out(m), FMT='("LEFT BC = Linear Neumann; ")', ADVANCE="NO")
				ELSE
					WRITE (UNIT=out(m), FMT='("LEFT BC = Linear Robin; ")', ADVANCE="NO")
				END IF
			ELSE
				IF (left_bc_flag == 0) THEN
					WRITE (UNIT=out(m), FMT='("LEFT BC = Non-Linear Dirichlet; ")', ADVANCE="NO")
				ELSE IF (left_bc_flag == 1) THEN
					WRITE (UNIT=out(m), FMT='("LEFT BC = Non-Linear Neumann; ")', ADVANCE="NO")
				ELSE
					WRITE (UNIT=out(m), FMT='("LEFT BC = Non-Linear Robin; ")', ADVANCE="NO")
				END IF
			END IF
			IF (right_lin_flag == 1) THEN
				IF (right_bc_flag == 0) THEN
					WRITE (UNIT=out(m), FMT='("RIGHT BC = Linear Dirichlet; ")', ADVANCE="NO")
				ELSE IF (right_bc_flag == 1) THEN
					WRITE (UNIT=out(m), FMT='("RIGHT BC = Linear Neumann; ")', ADVANCE="NO")
				ELSE
					WRITE (UNIT=out(m), FMT='("RIGHT BC = Linear Robin; ")', ADVANCE="NO")
				END IF
			ELSE
				IF (right_bc_flag == 0) THEN
					WRITE (UNIT=out(m), FMT='("RIGHT BC = Non-Linear Dirichlet; ")', ADVANCE="NO")
				ELSE IF (right_bc_flag == 1) THEN
					WRITE (UNIT=out(m), FMT='("RIGHT BC = Non-Linear Neumann; ")', ADVANCE="NO")
				ELSE
					WRITE (UNIT=out(m), FMT='("RIGHT BC = Non-Linear Robin; ")', ADVANCE="NO")
				END IF
			END IF
			IF (bottom_lin_flag == 1) THEN
				IF (bottom_bc_flag == 0) THEN
					WRITE (UNIT=out(m), FMT='("BOTTOM BC = Linear Dirichlet; ")', ADVANCE="NO")
				ELSE IF (bottom_bc_flag == 1) THEN
					WRITE (UNIT=out(m), FMT='("BOTTOM BC = Linear Neumann; ")', ADVANCE="NO")
				ELSE
					WRITE (UNIT=out(m), FMT='("BOTTOM BC = Linear Robin; ")', ADVANCE="NO")
				END IF
			ELSE
				IF (bottom_bc_flag == 0) THEN
					WRITE (UNIT=out(m), FMT='("BOTTOM BC = Non-Linear Dirichlet; ")', ADVANCE="NO")
				ELSE IF (bottom_bc_flag == 1) THEN
					WRITE (UNIT=out(m), FMT='("BOTTOM BC = Non-Linear Neumann; ")', ADVANCE="NO")
				ELSE
					WRITE (UNIT=out(m), FMT='("BOTTOM BC = Non-Linear Robin; ")', ADVANCE="NO")
				END IF
			END IF
```

```
                IF (top_lin_flag == 1) THEN
                    IF (top_bc_flag == 0) THEN
                        WRITE (UNIT=out(m), FMT='("TOP BC = Linear Dirichlet; ")')
                        WRITE (UNIT=out(m), FMT='("% ------------------------------------------------------------&
                                        &------------------------------------------------------------------
")')
                    ELSE IF (top_bc_flag == 1) THEN
                        WRITE (UNIT=out(m), FMT='("TOP BC = Linear Neumann; ")')
                        WRITE (UNIT=out(m), FMT='("% ------------------------------------------------------------&
                                        &------------------------------------------------------------------
")')
                    ELSE
                        WRITE (UNIT=out(m), FMT='("TOP BC = Linear Robin; ")')
                        WRITE (UNIT=out(m), FMT='("% ------------------------------------------------------------&
                                        &------------------------------------------------------------------
")')
                    END IF
                ELSE
                    IF (top_bc_flag == 0) THEN
                        WRITE (UNIT=out(m), FMT='("TOP BC =  Non-Linear Dirichlet.")')
                        WRITE (UNIT=out(m), FMT='("% ------------------------------------------------------------&
                                        &------------------------------------------------------------------
")')
                    ELSE IF (top_bc_flag == 1) THEN
                        WRITE (UNIT=out(m), FMT='("TOP BC = Non-Linear Neumann.")')
                        WRITE (UNIT=out(m), FMT='("% ------------------------------------------------------------&
                                        &------------------------------------------------------------------
")')
                    ELSE
                        WRITE (UNIT=out(m), FMT='("TOP BC = Non-Linear Robin.")')
                        WRITE (UNIT=out(m), FMT='("% ------------------------------------------------------------&
                                        &------------------------------------------------------------------
")')
                    END IF
                END IF
            END DO

        ELSE

            !           CALL DOUGLAS-GUNN ROUTINE TO COMPUTE THE EVOLUTION OF GRID FUNCTIONS. If exact solution is not
            !           available, request error estimation from the quasilinear Douglas-Gunn routine.      Compute exact errors,
            !           if known, otherwise, use the error estimate obtained from "delta_qlin_dgts".
            !
            IF ( (t > t0) .AND. (bcout_flag == 0) ) THEN    ! Set BCs & Print to Output files on the FIRST PASS post the time of BC change.

            !           DEFINE BC TYPE VARIATIONS FOR SUBSEQUENT TIME(S).
            !           NOTE: In the case of hemispherical asperity frictional melting, if time is less than or equal to the duration of
            !           asperity separation, the right BC is the frictional heat flux (Neumann) into the asperity. Otherwise, the asperity
            !           is surrounded by air at ambient temperature (Dirichlet).
            !           PDE BOUNDARY CONDITION FLAGS: These are being moved here from CONST_PARAMS to offer flexibility in terms of
            !           time varying BC TYPES (for instance a change from an initial Neumann BC to a subsequent Dirichlet BC. The BC
            !           type change can happen any number of times, and the case-specific handling of these changes will be dealt with
            !           different DO LOOPS for each BC set.
            !           NOTATION FOR BOUNDARY CONDITION FLAGS:          0 for DIRICHLET {i.e., Bbc(U) = B2bc(U)},
            !                                                                           1 for NEUMANN {i.e., Bbc(U) = U_x*B1bc(U)},
            !                                                                           2 for ROBIN {i.e., Bbc(u) = U_x*B1bc(U)}.
            !           All BCs are represented in the generalized non-linear forms encountered in heat conduction problems:
            !           Bbc(U) = U_x*B1bc(U)+ B2bc(U) or U_y*B1bc(U)+ B2bc(U). This form can be used to represent either NON-LINEAR or
            !           LINEAR BCs. PROVIDE ALL BOUNDARY OPERATORS, B, in this SPLIT FORM, using separate functions for B1 and B2, for
            !           EACH BC. These classifications and their implementations are discussed under the separate functions in the module
            !           "pde_routines", below, and ESPECIALLY UNDER THE SUBROUTINE "qldgts_coeff_rhs", where they are used:

                left_bc_flag  = 1
                right_bc_flag  = 1
                bottom_bc_flag  = 1
                top_bc_flag  = 1

            !           OPTIONAL Linear Robin Parameters, ALPHA_x & ALPHA_y for each of the two directions. Eg., in: L = U_x + alpha_x * U
!               alpha_x = 0.0_rp
!               alpha_y = 0.0_rp
            !
            !           BOUNDARY CONDITION LINEARITY FLAGS: 1 if linear, 0 if non-linear.
            !           These will affect the forms and values of the corresponding boundary condition functionals (lbc1, bbc1, tbc2, etc.)
            !           below. If any of these flags is 0 (non-linear BC) then the forms of these functionals have to be defined in the
            !           respective subroutines in MODULE "pde_routines":

                left_lin_flag = 1
                right_lin_flag = 0
                bottom_lin_flag = 1
                top_lin_flag = 1

            !           Confirm ALL BC types for this time range.
                DO m = 1, SIZE(out)
                    WRITE (UNIT=out(m), FMT='("% For time > To = ",ES8.2, ": ")', ADVANCE="NO") t0
                    IF (left_lin_flag == 1) THEN
                        IF (left_bc_flag == 0) THEN
                           WRITE (UNIT=out(m), FMT='("LEFT BC = Linear Dirichlet; ")', ADVANCE="NO")
                        ELSE IF (left_bc_flag == 1) THEN
                           WRITE (UNIT=out(m), FMT='("LEFT BC = Linear Neumann; ")', ADVANCE="NO")
                        ELSE
                           WRITE (UNIT=out(m), FMT='("LEFT BC = Linear Robin; ")', ADVANCE="NO")
                        END IF
                    ELSE
                        IF (left_bc_flag == 0) THEN
                           WRITE (UNIT=out(m), FMT='("LEFT BC = Non-Linear Dirichlet; ")', ADVANCE="NO")
                        ELSE IF (left_bc_flag == 1) THEN
                           WRITE (UNIT=out(m), FMT='("LEFT BC = Non-Linear Neumann; ")', ADVANCE="NO")
                        ELSE
                           WRITE (UNIT=out(m), FMT='("LEFT BC = Non-Linear Robin; ")', ADVANCE="NO")
                        END IF
                    END IF
                    IF (right_lin_flag == 1) THEN
                        IF (right_bc_flag == 0) THEN
                           WRITE (UNIT=out(m), FMT='("RIGHT BC = Linear Dirichlet; ")', ADVANCE="NO")
                        ELSE IF (right_bc_flag == 1) THEN
                           WRITE (UNIT=out(m), FMT='("RIGHT BC = Linear Neumann; ")', ADVANCE="NO")
                        ELSE
                           WRITE (UNIT=out(m), FMT='("RIGHT BC = Linear Robin; ")', ADVANCE="NO")
                        END IF
                    ELSE
                        IF (right_bc_flag == 0) THEN
                           WRITE (UNIT=out(m), FMT='("RIGHT BC = Non-Linear Dirichlet; ")', ADVANCE="NO")
                        ELSE IF (right_bc_flag == 1) THEN
                           WRITE (UNIT=out(m), FMT='("RIGHT BC = Non-Linear Neumann; ")', ADVANCE="NO")
```

```
                                        ELSE
                                          WRITE (UNIT=out(m), FMT='("RIGHT BC = Non-Linear Robin; ")', ADVANCE="NO")
                                        END IF
                                END IF


                                IF (bottom_lin_flag == 1) THEN
                                        IF (bottom_bc_flag == 0) THEN
                                          WRITE (UNIT=out(m), FMT='("BOTTOM BC = Linear Dirichlet; ")', ADVANCE="NO")
                                        ELSE IF (bottom_bc_flag == 1) THEN
                                          WRITE (UNIT=out(m), FMT='("BOTTOM BC = Linear Neumann; ")', ADVANCE="NO")
                                        ELSE
                                          WRITE (UNIT=out(m), FMT='("BOTTOM BC = Linear Robin; ")', ADVANCE="NO")
                                        END IF
                                ELSE
                                        IF (bottom_bc_flag == 0) THEN
                                          WRITE (UNIT=out(m), FMT='("BOTTOM BC = Non-Linear Dirichlet; ")', ADVANCE="NO")
                                        ELSE IF (bottom_bc_flag == 1) THEN
                                          WRITE (UNIT=out(m), FMT='("BOTTOM BC = Non-Linear Neumann; ")', ADVANCE="NO")
                                        ELSE
                                          WRITE (UNIT=out(m), FMT='("BOTTOM BC = Non-Linear Robin; ")', ADVANCE="NO")
                                        END IF
                                END IF
                                IF (top_lin_flag == 1) THEN
                                        IF (top_bc_flag == 0) THEN
                                          WRITE (UNIT=out(m), FMT='("TOP BC = Linear Dirichlet; ")')
                                          WRITE (UNIT=out(m), FMT='("% ------------------------------------------------------------&
                                                           &-------------------------------------------------------------
")')
                                        ELSE IF (top_bc_flag == 1) THEN
                                          WRITE (UNIT=out(m), FMT='("TOP BC = Linear Neumann; ")')
                                          WRITE (UNIT=out(m), FMT='("% ------------------------------------------------------------&
                                                           &-------------------------------------------------------------
")')
                                        ELSE
                                          WRITE (UNIT=out(m), FMT='("TOP BC = Linear Robin; ")')
                                          WRITE (UNIT=out(m), FMT='("% ------------------------------------------------------------&
                                                           &-------------------------------------------------------------
")')
                                        END IF
                                ELSE
                                        IF (top_bc_flag == 0) THEN
                                          WRITE (UNIT=out(m), FMT='("TOP BC =  Non-Linear Dirichlet.")')
                                          WRITE (UNIT=out(m), FMT='("% ------------------------------------------------------------&
                                                           &-------------------------------------------------------------
")')
                                        ELSE IF (top_bc_flag == 1) THEN
                                          WRITE (UNIT=out(m), FMT='("TOP BC = Non-Linear Neumann.")')
                                          WRITE (UNIT=out(m), FMT='("% ------------------------------------------------------------&
                                                           &-------------------------------------------------------------
")')
                                        ELSE
                                          WRITE (UNIT=out(m), FMT='("TOP BC = Non-Linear Robin.")')
                                          WRITE (UNIT=out(m), FMT='("% ------------------------------------------------------------&
                                                           &-------------------------------------------------------------
")')
                                        END IF
                                END IF
                        END DO
                        bcout_flag = 1

                END IF

                IF (exact_sol_flag == 1) THEN
                        CALL delta_qlin_dgts(x, y, t, k, u)

        !   SMOOTHING: If smooth_flag is non-zero, then apply appropriate smoothing to grid functions.      NOTE: SMOOTHING IS PROBLEM SPECIFIC
        !   AND THE GRID FUNCTIONS TO BE SMOOTHED HAVE TO BE DETERMINED, SOMETIMES THROUGH MANUAL ITERATIONS OF WHAT WORKS BEST. Below, two
        !   smoothing functions are provided for the case of a steep gradient at the right boundary of the problem domain. The smoothing
        !   factor is defined globally in the MODULE CONST_PARAMS, and is SPECIFIED on the command line along with the executable file.
                        IF (smooth_flag == 1) THEN
!                               DO i = x_points-2, x_points-1       ! 1D smoothing: DIRICHLET BC - Smooth columns nx-2 to nx-1.
                                DO i = x_points-3, x_points         ! 1D smoothing: NEUMANN BC - Smooth columns nx-3 to nx, i.e., INCLUDE BDRY. NODE.
                                        IF (i == x_points) u(j,i+1) = 2.0_rp*hx*f_right(y(j),t) + u(j, i-1)     ! For NEUMANN RIGHT BC.
                                        DO j = 1, y_points                              ! Since TOP & BOTTOM BCs are NEUMANN.
                                                u(j,i) = ( u(j,i-1) + smooth_factor*u(j,i) + u(j,i+1) )/(2.0_rp + smooth_factor)
                                        END DO
                                END DO
                        ELSE IF (smooth_flag == 2) THEN
!                               DO i = x_points-s1+1, x_points-1                ! 2D smoothing: DIRICHLET BC - Smooth columns nx-2 to nx-1.
                                DO i = x_points-s1, x_points        ! 2D smoothing: NEUMANN BC - Smooth columns nx-1 to nx, i.e., INCLUDE BDRY. NODE.
                                        s1 = 3 + INT(y0/hy)     ! Location of the flux-RBC input edge with respect to the current grid.
                                        IF (i == x_points) u(j,i+1) = 2.0_rp*hx*f_right(y(j),t) + u(j,i-1)      ! For NEUMANN RIGHT BC.
                                        DO j = 1, s1                                            ! Since the BOTTOM BC is NEUMANN.
                                                ! For NEUMANN BOTTOM BC.
                                                IF (j == 1) u(j-1,i) = u(j+1,i) - 2.0_rp*hy*f_bottom(x(i),t)
                                                u(j,i) = ( u(j,i-1) + u(j-1,i) + smooth_factor*u(j,i) + u(j,i+1) + u(j+1,i) )
                                                u(j,i) = u(j,i)/(4.0_rp + smooth_factor)
                                        END DO
                                END DO
                        END IF

                        max_u = 0.0_rp
                        min_u = 1/epsilon
                        max_error = 0.0_rp
                        DO j = 1, y_points
                                DO i = 1, x_points
                                        en(j,i) = ABS( f_exact(x(i),y(j),t) - u(j,i) )
                                        IF (verbose_flag == 1) THEN
                                                IF (en(j,i) > max_error) THEN
                                                        max_error = en(j,i)
                                                        max_error_u = u(j,i)
                                                        i_max = i
                                                        j_max = j
                                                END IF
                                                IF (u(j,i) > max_u) THEN
                                                        max_u       = u(j,i)
                                                        max_u_error = en(j,i)
                                                        i_tmax      = i
                                                        j_tmax      = j
                                                END IF
```

```
                                IF (u(j,i) < min_u) THEN
                                    min_u       = u(j,i)
                                    min_u_error = en(j,i)
                                    i_tmin      = i
                                    j_tmin      = j
                                END IF
                            END IF
                            IF (u(j,i) > global_max_u) THEN
                                global_max_u       = u(j,i)
                                global_max_u_error = en(j,i)
                                i_tmax_global      = i
                                j_tmax_global      = j
                                t_global_max       = t
                                nt_globalmax       = n_t
                            END IF
                            IF (en(j,i) > global_max_error) THEN
                                global_max_error   = en(j,i)
                                global_max_error_u = u(j,i)
                                i_max_global       = i
                                j_max_global       = j
                                t_global_max_error = t
                                nt_globalmax_error = n_t
                            END IF
                        END DO
                    END DO
                ELSE
                    IF (linear_flag /= 1) THEN
                        CALL delta_qlin_dgts(x, y, t, k, u, en, dn, srad)

            !   SMOOTHING: If smooth_flag is non-zero, then apply appropriate smoothing to grid functions.      NOTE: SMOOTHING IS PROBLEM SPECIFIC
            !   AND THE GRID FUNCTIONS TO BE SMOOTHED HAVE TO BE DETERMINED, SOMETIMES THROUGH MANUAL ITERATIONS OF WHAT WORKS BEST. Below, two
            !   smoothing functions are provided for the case of a steep gradient at the right boundary of the problem domain. The smoothing
            !   factor is defined globally in the MODULE CONST_PARAMS, and is SPECIFIED on the command line along with the executable file.
                        IF (smooth_flag == 1) THEN
!                           DO i = x_points-2, x_points-1               ! 1D smoothing: DIRICHLET BC - Smooth columns nx-2 to nx-1.
                            DO i = x_points-3, x_points                 ! 1D smoothing: NEUMANN BC - INCLUDE BOUNDARY NODES.
                                IF (i == x_points) u(j,i+1) = 2.0_rp*hx*f_right(y(j),t) + u(j, i-1)     ! For NEUMANN RIGHT BC.
                                DO j = 1, y_points                            ! Since both the TOP & BOTTOM BCs are NEUMANN.
                                    u(j,i) = ( u(j,i-1) + smooth_factor*u(j,i) + u(j,i+1) )/(2.0_rp + smooth_factor)
                                END DO
                            END DO
                        ELSE IF (smooth_flag == 2) THEN
!                           DO i = x_points-s1+1, x_points-1            ! 2D smoothing: DIRICHLET BC - Smooth columns nx-2 to nx-1.
                            DO i = x_points-s1, x_points                ! 2D smoothing: NEUMANN BC - INCLUDE BOUNDARY NODES.
                                s1 = 3 + INT(y0/hy)     ! Location of the flux-RBC input edge with respect to the current grid.
                                IF (i == x_points) u(j,i+1) = 2.0_rp*hx*f_right(y(j),t) + u(j,i-1)      ! For NEUMANN RIGHT BC.
                                DO j = 1, s1                                                ! Since the BOTTOM BC is NEUMANN.
                                    ! For NEUMANN BOTTOM BC.
                                    IF (j == 1) u(j-1,i) = u(j+1,i) - 2.0_rp*hy*f_bottom(x(i),t)
                                    u(j,i) = ( u(j,i-1) + u(j-1,i) + smooth_factor*u(j,i) + u(j,i+1) + u(j+1,i) )
                                    u(j,i) = u(j,i)/(4.0_rp + smooth_factor)
                                END DO
                            END DO
                        END IF

                        max_u = 0.0_rp
                        min_u = 1/epsilon
                        max_error = 0.0_rp
                        DO j = 1, y_points
                            DO i = 1, x_points
                                IF (verbose_flag == 1) THEN
                                    IF (en(j,i) > max_error) THEN
                                        max_error = en(j,i)
                                        max_error_u = u(j,i)
                                        i_max = i
                                        j_max = j
                                    END IF
                                    IF (u(j,i) > max_u) THEN
                                        max_u       = u(j,i)
                                        max_u_error = en(j,i)
                                        i_tmax      = i
                                        j_tmax      = j
                                    END IF
                                    IF (u(j,i) < min_u) THEN
                                        min_u       = u(j,i)
                                        min_u_error = en(j,i)
                                        i_tmin      = i
                                        j_tmin      = j
                                    END IF
                                END IF
                                IF (u(j,i) > global_max_u) THEN
                                    global_max_u       = u(j,i)
                                    global_max_u_error = en(j,i)
                                    i_tmax_global      = i
                                    j_tmax_global      = j
                                    t_global_max       = t
                                    nt_globalmax       = n_t
                                END IF
                                IF (en(j,i) > global_max_error) THEN
                                    global_max_error   = en(j,i)
                                    global_max_error_u = u(j,i)
                                    i_max_global       = i
                                    j_max_global       = j
                                    t_global_max_error = t
                                    nt_globalmax_error = n_t
                                END IF
                            END DO
                        END DO
                    ELSE
                        CALL delta_qlin_dgts(x, y, t, k, u)
                        max_u = 0.0_rp
                        min_u = 1/epsilon
                        DO j = 1, y_points
                            DO i = 1, x_points
                                IF (verbose_flag == 1) THEN
                                    IF (u(j,i) > max_u) THEN
                                        max_u  = u(j,i)
                                        i_tmax = i
                                        j_tmax = j
                                    END IF
```

```
                                                IF (u(j,i) < min_u) THEN
                                                            min_u  = u(j,i)
                                                            i_tmin = i
                                                            j_tmin = j
                                                END IF
                                    END IF
                                    IF (u(j,i) > global_max_u) THEN
                                                global_max_u  = u(j,i)
                                                i_tmax_global = i
                                                j_tmax_global = j
                                                t_global_max  = t
                                                nt_globalmax  = n_t
                                    END IF
                        END DO
                  END DO
            END IF
      END IF
END IF

!---------------------------------------------------------------
!           STORE OUTPUT FILE & SCREEN OUTPUT DATA IN DATA ARRAYS.
!---------------------------------------------------------------

!           SCREEN OUTPUT:
!           -------------
IF ( (verbose_flag /= 1) .AND. (n_t == nt_max_evol(evol_count)) ) THEN
!Optional            PRINT *, "STARTING EVOLUTION DATA PROCESSING FOR t(",n_t,") = ",t,"."
END IF

!           OUTPUT FILES 1 & 2: Output data if this is the correct time level.
!           -----------------------------------------------
IF ( n_t == nt_snap(out_count) ) THEN
            u_grid_norm(out_count) = 0.0_rp
            DO j = 1, y_points
                        DO i = 1, x_points
                                    IF (exact_sol_flag == 1) THEN
                                                en(j,i) = ABS( f_exact(x(i),y(j),t) - u(j,i) )
                                    ELSE
                                                IF (linear_flag == 1) en(j,i) = 1.0E30_rp
                                    END IF
                                    u_grid_norm(out_count) = u_grid_norm(out_count) + (u(j,i))*u(j,i)
                        END DO
            END DO
            u_grid_norm(out_count) = SQRT( u_grid_norm(out_count) )
            DO j = 1,n_ygrid
                        DO i = 1,n_xgrid
                                    u_grid(j,i,out_count) = u(j_grid(j),i_grid(i))
                                    IF (exact_sol_flag == 1) THEN
                                                u_errg(j,i,out_count) = en(j_grid(j),i_grid(i))/u_grid_norm(out_count)
                                    ELSE
                                                IF (linear_flag == 1) THEN
                                                            u_errg(j,i,out_count) = en(j_grid(j),i_grid(i))
                                                ELSE
                                                            u_errg(j,i,out_count) = en(j_grid(j),i_grid(i))/u_grid_norm(out_count)
                                                END IF
                                    END IF
                        END DO
            END DO
            out_count = out_count + 1
END IF

!           OUTPUT FILE 3: Snapshot data, if this is the correct time level.
!           ----------------------------------------------------------------
IF (n_t == nt_xsnap) THEN
            DO i = 1, n_xsnap
                        u_xsnap(i,1) =  x(i_xsnap(i))
                        u_xsnap(i,2) =  u(j_xsnap,i_xsnap(i))
            END DO
END IF
IF (n_t == nt_ysnap) THEN
            DO j = 1, n_ysnap
                        u_ysnap(j,1) =  y(j_ysnap(j))
                        u_ysnap(j,2) =  u(j_ysnap(j),i_ysnap)
            END DO
END IF

!           OUTPUT FILE 4: TEMPERATURE EVOLUTION AT A SINGLE (x,y) grid point in the problem domain; EVOLUTION OF MAXIMUM DOMAIN
!           TEMPERATURE & MAXIMUM DOMAIN ERROR :
!           -------------------------------------------------------------------------------------------------------------------
DO m = 1, n_evol
            IF (n_t == nt_evol(m)) THEN
                        u_evol(m,1) =  t
                        u_evol(m,2) =  u(j_evol,i_evol)
            END IF
END DO
IF (n_t == nt_max_evol(evol_count)) THEN
    IF (exact_sol_flag == 1) THEN
            max_error = 0.0_rp
            max_u = 0.0_rp
            min_u = 1/epsilon
            en_norm = 0.0_rp
            u_norm = 0.0_rp
            DO j = 1, y_points
                        DO i = 1, x_points
                                    en(j,i) = ABS( f_exact(x(i),y(j),t) - u(j,i) )
                                    en_norm = en_norm + (en(j,i))*en(j,i)
                                    u_norm = u_norm + (u(j,i))*u(j,i)
                                    IF (en(j,i) > max_error) THEN
                                                max_error   = en(j,i)
                                                max_error_u = u(j,i)
                                                i_max = i
                                                j_max = j
                                    END IF
                                    IF (u(j,i) > max_u) THEN
                                                max_u       = u(j,i)
                                                max_u_error = en(j,i)
                                                i_tmax      = i
                                                j_tmax      = j
                                    END IF
```

```
                        IF (u(j,i) < min_u) THEN
                                min_u       = u(j,i)
                                min_u_error = en(j,i)
                                i_tmin      = i
                                j_tmin      = j
                        END IF
                END DO
        END DO
        en_norm = SQRT(en_norm)
        u_norm = SQRT(u_norm)
        en = en/u_norm
        max_error = max_error/u_norm
        max_u_error = max_u_error/u_norm
        min_u_error = min_u_error/u_norm
        u_maxevol(evol_count, 1) = n_t
        u_maxevol(evol_count, 2) = t
        u_maxevol(evol_count, 3) = j_tmax
        u_maxevol(evol_count, 4) = i_tmax
        u_maxevol(evol_count, 5) = u(j_tmax, i_tmax)
        u_maxevol(evol_count, 6) = max_u_error
        u_maxevol(evol_count, 7) = u_norm
        u_minevol(evol_count, 1) = n_t
        u_minevol(evol_count, 2) = t
        u_minevol(evol_count, 3) = j_tmin
        u_minevol(evol_count, 4) = i_tmin
        u_minevol(evol_count, 5) = u(j_tmin, i_tmin)
        u_minevol(evol_count, 6) = min_u_error
        u_minevol(evol_count, 7) = u_norm
        error_maxevol(evol_count, 1) = n_t
        error_maxevol(evol_count, 2) = t
        error_maxevol(evol_count, 3) = j_max
        error_maxevol(evol_count, 4) = i_max
        error_maxevol(evol_count, 5) = max_error
        error_maxevol(evol_count, 6) = u(j_max, i_max)
        error_maxevol(evol_count, 7) = u_norm
ELSE
        IF (linear_flag /= 1) THEN
                max_error = 0.0_rp
                max_u = 0.0_rp
                min_u = 1/epsilon
                en_norm = 0.0_rp
                u_norm = 0.0_rp

                DO j = 1, y_points
                        DO i = 1, x_points
                                en_norm = en_norm + (en(j,i))*en(j,i)
                                u_norm = u_norm + (u(j,i))*u(j,i)
                                IF (en(j,i) > max_error) THEN
                                        max_error = en(j,i)
                                        max_error_u = u(j,i)
                                        i_max = i
                                        j_max = j
                                END IF
                                IF (u(j,i) > max_u) THEN
                                        max_u       = u(j,i)
                                        max_u_error = en(j,i)
                                        i_tmax      = i
                                        j_tmax      = j
                                END IF
                                IF (u(j,i) < min_u) THEN
                                        min_u       = u(j,i)
                                        min_u_error = en(j,i)
                                        i_tmin      = i
                                        j_tmin      = j
                                END IF
                        END DO
                END DO
                en_norm = SQRT(en_norm)
                u_norm = SQRT(u_norm)
                en = en/u_norm
                max_error = max_error/u_norm
                max_u_error = max_u_error/u_norm
                min_u_error = min_u_error/u_norm
                u_maxevol(evol_count, 1) = n_t
                u_maxevol(evol_count, 2) = t
                u_maxevol(evol_count, 3) = j_tmax
                u_maxevol(evol_count, 4) = i_tmax
                u_maxevol(evol_count, 5) = u(j_tmax, i_tmax)
                u_maxevol(evol_count, 6) = max_u_error
                u_maxevol(evol_count, 7) = u_norm
                u_minevol(evol_count, 1) = n_t
                u_minevol(evol_count, 2) = t
                u_minevol(evol_count, 3) = j_tmin
                u_minevol(evol_count, 4) = i_tmin
                u_minevol(evol_count, 5) = u(j_tmin, i_tmin)
                u_minevol(evol_count, 6) = min_u_error
                u_minevol(evol_count, 7) = u_norm
                error_maxevol(evol_count, 1) = n_t
                error_maxevol(evol_count, 2) = t
                error_maxevol(evol_count, 3) = j_max
                error_maxevol(evol_count, 4) = i_max
                error_maxevol(evol_count, 5) = max_error
                error_maxevol(evol_count, 6) = u(j_max, i_max)
                error_maxevol(evol_count, 7) = u_norm
        ELSE
                max_u = 0.0_rp
                min_u = 1/epsilon
                DO j = 1, y_points
                        DO i = 1, x_points
                                IF (u(j,i) > max_u) THEN
                                        max_u  = u(j,i)
                                        i_tmax = i
                                        j_tmax = j
                                END IF
                                IF (u(j,i) < min_u) THEN
                                        min_u  = u(j,i)
                                        i_tmin = i
                                        j_tmin = j
                                END IF
                        END DO
                END DO
                u_maxevol(evol_count, 1) = n_t
                u_maxevol(evol_count, 2) = t
                u_maxevol(evol_count, 3) = j_tmax
```

```
                        u_maxevol(evol_count, 4) = i_tmax
                        u_maxevol(evol_count, 5) = u(j_tmax, i_tmax)
                        u_maxevol(evol_count, 6) = 1.0E30_rp
                        u_maxevol(evol_count, 7) = 1.0E30_rp
                        u_minevol(evol_count, 2) = t
                        u_minevol(evol_count, 3) = j_tmin
                        u_minevol(evol_count, 4) = i_tmin
                        u_minevol(evol_count, 5) = u(j_tmin, i_tmin)
                        u_minevol(evol_count, 6) = 1.0E30_rp
                        u_minevol(evol_count, 7) = 1.0E30_rp
                END IF
            END IF
            evol_count = evol_count + 1
        END IF

        !           OUTPUT FILE 5: GRID FUNCTION CONVERGENCE DATA:
        !           ----------------------------------------------
        DO j = 1, SIZE(grid_conv,1)
                IF (n_t == nt_gridconv(j)) THEN
                        u_conv(j) = u(ny_gridconv(j),nx_gridconv(j))
                END IF
        END DO


        IF (verbose_flag == 1) THEN
            PRINT *, "t(", n_t,") = ",t,":"
            PRINT *, "             row=",j_tmax,", col=",i_tmax,": DOMAIN MAXIMUM TEMPERATURE = ", u(j_tmax, i_tmax)
            PRINT *, "             row=",j_tmin,", col=",i_tmin,": DOMAIN MINIMUM TEMPERATURE = ", u(j_tmin, i_tmin)
            IF (linear_flag /= 1) THEN
                PRINT *, "             row=",j_max, ", col=",i_max," : DOMAIN MAXIMUM ERROR       = ",max_error,", &
                                                                                &TEMPERATURE = ", u(j_max, i_max),"."
            END IF
        END IF


        ! UPDATE TIME TO NEXT STEP.
        t = t + k

END DO

!--------------------------------------------------------------------------------------------------------------------------
!           END OF MAIN COMPUTAIONAL LOOP.
!--------------------------------------------------------------------------------------------------------------------------

!--------------------------------------------------------------------------------------------------------------------------
!           OUTPUT STORED RUN DATA TO ALL OUTPUT FILES.
!--------------------------------------------------------------------------------------------------------------------------

!           --------------------------------------------------------------------------------------------------------------------
!           OUTPUT FILE 1: Output grid functions at the resolution required for convergence tests.
!           --------------------------------------------------------------------------------------------------------------------
DO m = 1, SIZE(t_snap)
        WRITE (UNIT=out(1), FMT='("TIME STEP",I6,": The solution u(x,y) at time = ", F10.6)') nt_snap(m),t_snap(m)
        WRITE (UNIT=out(1), FMT='("% ------------------------------------------------------------------")')
        WRITE (UNIT=out(1), FMT='("% x = ")', ADVANCE="NO")
        DO i = 1,n_xgrid                                                            ! Print out X-coordinate headings.
                IF (i == n_xgrid) THEN
                        WRITE (UNIT=out(1), FMT='(1X, F9.6)') x(i_grid(i))
                        EXIT
                END IF
                WRITE (UNIT=out(1), FMT='(1X, F9.6,",")', ADVANCE="NO") x(i_grid(i))
        END DO
        DO j = 1,n_ygrid                                                            ! Print out each row vector (y-row) of the solution.
                IF ( (coord_flag == 2) .OR. (coord_flag == 3) )THEN
                        IF (j == n_ygrid) THEN  ! If y_top = PI, then repeat last value within domain for top boundary point
                                WRITE (UNIT=out(1), FMT='("y(",I5,")= ",F9.6,",")', ADVANCE="NO") j_grid(j), &
                                                                            & ( y(j_grid(j-1)) + out_y_grid_spacing )
                        ELSE
                                WRITE (UNIT=out(1), FMT='("y(",I5,")= ",F9.6,",")', ADVANCE="NO") j_grid(j), y(j_grid(j))
                        END IF
                ELSE
                        WRITE (UNIT=out(1), FMT='("y(",I5,")= ",F9.6,",")', ADVANCE="NO") j_grid(j), y(j_grid(j))
                END IF
                DO i = 1,n_xgrid
                        IF (i == n_xgrid) THEN
                                IF ( (coord_flag == 2) .OR. (coord_flag == 3) )THEN
                                        IF (j == n_ygrid) THEN  ! If y_top = PI, then repeat last value in domain for top bdry. point
                                                WRITE (UNIT=out(1), FMT='(1X,ES18.8)') u_grid(j-1,i,m)
                                        ELSE
                                                WRITE (UNIT=out(1), FMT='(1X,ES18.8)') u_grid(j,i,m)
                                        END IF
                                ELSE
                                        WRITE (UNIT=out(1), FMT='(1X,ES18.8)') u_grid(j,i,m)
                                END IF
                                EXIT
                        END IF
                        IF ( (coord_flag == 2) .OR. (coord_flag == 3) )THEN
                                IF (j == n_ygrid) THEN  ! If y_top = PI, then repeat last value within domain for top boundary point
                                        WRITE (UNIT=out(1), FMT='(1X,ES18.8,",")', ADVANCE="NO") u_grid(j-1,i,m)
                                ELSE
                                        WRITE (UNIT=out(1), FMT='(1X,ES18.8,",")', ADVANCE="NO") u_grid(j,i,m)

                                END IF
                        ELSE
                                WRITE (UNIT=out(1), FMT='(1X,ES18.8,",")', ADVANCE="NO") u_grid(j,i,m)
                        END IF
                END DO
        END DO
        WRITE (UNIT=out(1), FMT='("% -------------------------------------------------------------------------------------"/)')
END DO

!           --------------------------------------------------------------------------------------------------------------------
!           OUTPUT FILE 2: Print ERROR data at required resolution.
!           Exact error distribution, en(yj,xi), at the current time step if exact solution is known;
!           ESTIMATED Error distribution, en_est(yj,xi), at the current time step when exact solution is not available.
!           --------------------------------------------------------------------------------------------------------------------
DO m = 1, SIZE(t_snap)
        IF (exact_sol_flag == 1) THEN
                WRITE (UNIT=out(2), FMT='("TIME STEP",I6,": The RELATIVE error en(x,y) at time = ", F10.6,&
                                                            & ": U_norm = ",ES18.8)') nt_snap(m), t_snap(m), u_grid_norm(m)
                WRITE (UNIT=out(2), FMT='("% ------------------------------------------------------------------------")')
        ELSE
                WRITE (UNIT=out(2), FMT='("TIME STEP",I6,": The ESTIMATED RELATIVE error en_est(x,y) at time = ", F10.6, &
                                                            & ": U_norm = ",ES18.8)') nt_snap(m), t_snap(m), u_grid_norm(m)
```

```
                WRITE (UNIT=out(2), FMT='("% -------------------------------------------------------------------------------")')
        END IF
        WRITE (UNIT=out(2), FMT='("% x = ")', ADVANCE="NO")
        DO i = 1,n_xgrid                                                        ! Print out X-coordinate headings.
                IF (i == n_xgrid) THEN
                        WRITE (UNIT=out(2), FMT='(1X, F9.6)') x(i_grid(i))
                        EXIT
                END IF
                WRITE (UNIT=out(2), FMT='(1X, F9.6,",")', ADVANCE="NO") x(i_grid(i))
        END DO
        DO j = 1,n_ygrid                                                        ! Print out each row vector (y-row) of the solution.
                IF ( (coord_flag == 2) .OR. (coord_flag == 3) )THEN
                        IF (j == n_ygrid) THEN  ! If y_top = PI, then repeat last value within domain for top boundary point
                                WRITE (UNIT=out(2), FMT='("y(",I5,")= ",F9.6,",")', ADVANCE="NO") j_grid(j), &
                                                                                & ( y(j_grid(j-1)) + out_y_grid_spacing )
                        ELSE
                                WRITE (UNIT=out(2), FMT='("y(",I5,")= ",F9.6,",")', ADVANCE="NO") j_grid(j), y(j_grid(j))
                        END IF
                ELSE
                        WRITE (UNIT=out(1), FMT='("y(",I5,")= ",F9.6,",")', ADVANCE="NO") j_grid(j), y(j_grid(j))
                END IF
                DO i = 1,n_xgrid
                        IF (i == n_xgrid) THEN
                                IF ( (coord_flag == 2) .OR. (coord_flag == 3) )THEN
                                        IF (j == n_ygrid) THEN  ! If y_top = PI, then repeat last value in domain for top bdry. point
                                                WRITE (UNIT=out(2), FMT='(1X,ES18.8)') u_errg(j-1,i,m)
                                        ELSE
                                                WRITE (UNIT=out(2), FMT='(1X,ES18.8)') u_errg(j,i,m)
                                        END IF
                                ELSE
                                        WRITE (UNIT=out(2), FMT='(1X,ES18.8)') u_errg(j,i,m)
                                END IF
                                EXIT
                        END IF
                        IF ( (coord_flag == 2) .OR. (coord_flag == 3) )THEN
                                IF (j == n_ygrid) THEN  ! If y_top = PI, then repeat last value within domain for top boundary point
                                        WRITE (UNIT=out(2), FMT='(1X,ES18.8,",")', ADVANCE="NO") u_errg(j-1,i,m)
                                ELSE
                                        WRITE (UNIT=out(2), FMT='(1X,ES18.8,",")', ADVANCE="NO") u_errg(j,i,m)
                                END IF
                        ELSE
                                WRITE (UNIT=out(2), FMT='(1X,ES18.8,",")', ADVANCE="NO") u_errg(j,i,m)
                        END IF
                END DO
        END DO
        WRITE (UNIT=out(2), FMT='("% ------------------------------------------------------------------------------------"/)')
END DO

!       ------------------------------------------------------------------------------------------------------------------------
!       OUTPUT FILE 3: Snapshot data.
!       ------------------------------------------------------------------------------------------------------------------------
!       3a. Plot the profile parallel to x-axis (corresponding to y_xsnap and t_xsnap).
!       ---------------------------------------------------------------------------
WRITE (UNIT=out(3), FMT='(/"SNAPSHOT at y = ",F9.6," & t = ",F9.6,":")') y_xsnap, t_xsnap
WRITE (UNIT=out(3), FMT='("---------------------------------------------------------------------"/)')
WRITE (UNIT=out(3), FMT='(5X,"x",8X,"U_xsnap(x)")')
DO i = 1, n_xsnap
        WRITE (UNIT=out(3), FMT='(3X,F4.2,3X,ES17.10)') u_xsnap(i,1), u_xsnap(i,2)
END DO
WRITE (UNIT=out(3), FMT='("---------------------------------------------------------------------"/)')

!       3b. Plot the profile parallel to y-axis (corresponding to x_ysnap and t_ysnap).
!       ---------------------------------------------------------------------------
WRITE (UNIT=out(3), FMT='("SNAPSHOT at x = ",F9.6," & t = ",F9.6,":")') x_ysnap, t_ysnap
WRITE (UNIT=out(3), FMT='("---------------------------------------------------------------------"/)')
WRITE (UNIT=out(3), FMT='(5X,"y",8X,"U_ysnap(y)")')
DO j = 1, n_ysnap
        IF (j == n_ysnap) THEN
                WRITE (UNIT=out(3), FMT='(3X,F4.2,3X,ES17.10)') ( u_ysnap(j-1,1) + out_y_grid_spacing ), u_ysnap(j-1,2)
        ELSE
                WRITE (UNIT=out(3), FMT='(3X,F4.2,3X,ES17.10)') u_ysnap(j,1), u_ysnap(j,2)
        END IF
END DO
WRITE (UNIT=out(3), FMT='("---------------------------------------------------------------------"/)')

!       ------------------------------------------------------------------------------------------------------------------------
!       OUTPUT FILE 4: TEMPERATURE EVOLUTION AT A SINGLE (x,y) grid point, MAX. & MIN. TEMPERATURE, and MAX. ERROR in the problem domain:
!       ------------------------------------------------------------------------------------------------------------------------

!       First, output the time lag between the maximum temperature and the time of separation.
WRITE (UNIT=out(4), FMT='("TIME LAG BETWEEN TIME CORRESPONDING TO U_max AND TIME AT ASPERITY SEPARATION = ",ES13.6)') &
                                                                                & (t_global_max - t0)
WRITE (UNIT=out(4), FMT='("RELATIVE TIME LAG (w.r.t. T0) BETWEEN TIME CORRESPONDING TO U_max AND TIME AT ASPERITY &
                                                                &SEPARATION = ",ES13.6)') (t_global_max - t0)/t0
WRITE (UNIT=out(4), FMT='(/"-------------------------------------------------------------------------------------------")')

!       4a. Grid Function Evolution at (x_time, y_time):
!       ------------------------------------------------
WRITE (UNIT=out(4), FMT='("Grid Function evolution at grid point: (",F9.6,", ",F9.6,").")') x_time, y_time
WRITE (UNIT=out(4), FMT='("---------------------------------------------------------------------------------")')
WRITE (UNIT=out(4), FMT='(5X,"t",5X,"U(x_time, y_time)")')
DO m = 1, n_evol
        WRITE (UNIT=out(4), FMT='(3X,F4.2,3X,ES15.8)') u_evol(m,1), u_evol(m,2)
END DO
WRITE (UNIT=out(4), FMT='("---------------------------------------------------------------------"/)')

!       4b. Maximum Temperature Evolution:
!       ----------------------------------
WRITE (UNIT=out(4), FMT='("Domain Maximum Temperature evolution:")')
WRITE (UNIT=out(4), FMT='("-------------------------------------------------------------------------------------------")')
IF (exact_sol_flag == 1) THEN
        WRITE (UNIT=out(4), FMT='(4X,"Step #",8X," t",8X,"    j  ",3X,"   i  ",3X,"     U_max     ",3X,"Relative Error",&
                                                                                & 3X,"     U_norm    ")')
ELSE
        WRITE (UNIT=out(4), FMT='(4X,"Step #",8X," t",8X,"    j  ",3X,"   i  ",3X,"     U_max     ",3X,"Est. Relative Error",&
                                                                                & 3X,"     U_norm    ")')
END IF
DO m = 1, num_tmaxevol
        WRITE (UNIT=out(4), FMT='(3X,I7,3X,ES12.6,3X,2(I6,3X),3(ES15.8,3X))') u_maxevol(m,1), u_maxevol(m,2), u_maxevol(m,3), &
                                                & u_maxevol(m,4), u_maxevol(m,5), u_maxevol(m,6), u_minevol(m,7)
END DO
WRITE (UNIT=out(4), FMT='(/"TEMPORAL GLOBAL TEMPERATURE MAXIMA: ")')
WRITE (UNIT=out(4), FMT='("-----------------------------------")')
```

```fortran
        IF (exact_sol_flag == 1) THEN
                WRITE (UNIT=out(4), FMT='(3X,I7,3X,F4.2,3X,2(I6,3X),2(ES15.8,3X))') nt_globalmax, t_global_max, j_tmax_global, &
                                                                                    & i_tmax_global, global_max_u, global_max_u_error
        ELSE
            IF (linear_flag /= 1) THEN
                WRITE (UNIT=out(4), FMT='(3X,I7,3X,F4.2,3X,2(I6,3X),2(ES15.8,3X))') nt_globalmax, t_global_max, j_tmax_global, &
                                                                                    & i_tmax_global, global_max_u, global_max_u_error
            ELSE
                WRITE (UNIT=out(4), FMT='(3X,I7,3X,F4.2,3X,2(I6,3X),2(ES15.8,3X))') nt_globalmax, t_global_max, j_tmax_global, &
                                                                                    & i_tmax_global, global_max_u, 1.0E30_rp
            END IF
        END IF
        WRITE (UNIT=out(4), FMT='("-----------------------------------------------------------------------------------------------"/)')

        !           4c. Maximum Error Evolution:
        !           ----------------------------------
        WRITE (UNIT=out(4), FMT='("Domain Maximum Error evolution:")')
        WRITE (UNIT=out(4), FMT='("------------------------------------------------------------------------------------------------")')
        IF (exact_sol_flag == 1) THEN
                WRITE (UNIT=out(4), FMT='(4X,"Step #",8X," t",8X,"    j  ",3X,"   i  ",1X,"  Max. Rel. Error  ",1X,"       U       ",&
                                                                                                  & 3X,"     U_norm    ")')

        ELSE
                WRITE (UNIT=out(4), FMT='(4X,"Step #",8X," t",8X,"    j  ",3X,"   i  ",1X,"Max. Est. Rel. Err.",1X,"       U       ",&
                                                                                                  & 3X,"     U_norm    ")')
        END IF
        DO m = 1, num_tmaxevol
                WRITE (UNIT=out(4), FMT='(3X,I7,3X,ES12.6,3X,2(I6,3X),3(ES15.8,3X))') error_maxevol(m,1), error_maxevol(m,2), &
                                                & error_maxevol(m,3), error_maxevol(m,4), error_maxevol(m,5), error_maxevol(m,6), u_minevol(m,7)
        END DO
        WRITE (UNIT=out(4), FMT='(/"TEMPORAL GLOBAL ABSOLUTE ERROR MAXIMA: ")')
        WRITE (UNIT=out(4), FMT='("-----------------------------")')
        IF (exact_sol_flag == 1) THEN
                WRITE (UNIT=out(4), FMT='(3X,I7,3X,F4.2,3X,2(I6,3X),2(ES15.8,3X))') nt_globalmax_error, t_global_max_error, j_max_global, &
                                                                                    & i_max_global, global_max_error, global_max_error_u
        ELSE
            IF (linear_flag /= 1) THEN
                WRITE (UNIT=out(4), FMT='(3X,I7,3X,F4.2,3X,2(I6,3X),2(ES15.8,3X))') nt_globalmax_error, t_global_max_error, j_max_global, &
                                                                                    & i_max_global, global_max_error, global_max_error_u
            ELSE
                WRITE (UNIT=out(4), FMT='(3X,I7,3X,F4.2,3X,2(I6,3X),2(ES15.8,3X))') nt_globalmax_error, t_global_max_error, j_max_global, &
                                                                                    & i_max_global, global_max_error, 1.0E30_rp
            END IF
        END IF
        WRITE (UNIT=out(4), FMT='("-----------------------------------------------------------------------------------------------"/)')

        !           4d. Minimum Temperature Evolution:
        !           ----------------------------------
        WRITE (UNIT=out(4), FMT='(/"Domain Minimum Temperature evolution:")')
        WRITE (UNIT=out(4), FMT='("------------------------------------------------------------------------------------------------")')
        IF (exact_sol_flag == 1) THEN
                WRITE (UNIT=out(4), FMT='(4X,"Step #",8X," t",8X,"   j  ",3X,"   i  ",3X,"     U_min     ",3X," Relative Error", 3X,"     U_norm    ")')
        ELSE
                WRITE (UNIT=out(4), FMT='(4X,"Step #",8X," t",8X,"   j  ",3X,"   i  ",3X,"     U_min     ",3X,"Est. Relative Error", 3X,"     U_norm    ")')
        END IF
        DO m = 1, num_tmaxevol
                WRITE (UNIT=out(4), FMT='(3X,I7,3X,ES12.6,3X,2(I6,3X),3(ES15.8,3X))') u_minevol(m,1), u_minevol(m,2), u_minevol(m,3), &
                                                                        & u_minevol(m,4), u_minevol(m,5), u_minevol(m,6), u_minevol(m,7)
        END DO
        WRITE (UNIT=out(4), FMT='("-----------------------------------------------------------------------------------------------"/)')

        !           ---------------------------------------------------------------------------------------------------------------------
        !           OUTPUT FILE 5: GRID FUNCTION CONVERGENCE DATA.
        !           ---------------------------------------------------------------------------------------------------------------------
        WRITE (UNIT=out(5), FMT='(/"Grid Function Convergence Data at the following grid points: ")')
        WRITE (UNIT=out(5), FMT='("--------------------------------------------------------------------------------")')
        WRITE (UNIT=out(5), FMT='(1X,"k",1X,"hx",1X,"hy",1X,"U1",1X,"U2",1X,"U3",1X,"U4",1X,"U5",1X,"U6",1X,"U7",1X,"U8",)')
        WRITE (UNIT=out(5), FMT='(3(1X,F8.6))', ADVANCE="NO") k, hx, hy
        DO m = 1, SIZE(grid_conv,1)
                WRITE (UNIT=out(5), FMT='(ES17.10,1X)', ADVANCE="NO") u_conv(m)
        END DO
        WRITE (UNIT=out(5), FMT='(/"-----------------------------------------------------------------------------------------------"/)')

        PRINT *, " "
        PRINT *, "-----------------------------------------------------------------------------------------------------------"
        !------------------------------------------------------------------------------------------------------------------------
        !           END OF INPUT TO OUTPUT FILES.
        !------------------------------------------------------------------------------------------------------------------------

        !------------------------------------------------------------------------------------------------------------------------
        !           PROGRAM CLOSING SEQUENCE: Deallocate arrays, and lose all files.
        !------------------------------------------------------------------------------------------------------------------------
!           Deallocate ALL arrays.
        IF (linear_flag /= 1) THEN
                DEALLOCATE (coeff, dn, en, error_maxevol, i_grid, i_xsnap, j_grid, j_ysnap, NSu_m, Nu_m, nt_evol, nt_max_evol, srad, &
                              & rhs, rs, t_max_evol, u, u_errg, u_evol, u_grid, u_n, u_old, u_maxevol, u_minevol, u_xsnap, u_ysnap, x, y, &
                              & STAT=dealloc_error)
        ELSE
                DEALLOCATE (coeff, en, error_maxevol, i_grid, i_xsnap, j_grid, j_ysnap, NSu_m, Nu_m, nt_evol, nt_max_evol, &
                              & rhs, t_max_evol, u, u_errg, u_evol, u_grid, u_n, u_maxevol, u_minevol, u_xsnap, u_ysnap, x, y, STAT=dealloc_error)
        END IF
        IF (dealloc_error /=0) THEN
                PRINT *, "WARNING: SOME OR ALL Arrays could not be DEALLOCATED!"
        END IF
        PRINT *, "FINISHED DEALLOCATING ALL ARRAYS."

!           Close output files.
        DO m = 1, SIZE(out)
                CLOSE (UNIT=out(m), STATUS="KEEP", IOSTAT=close_status)
                IF (close_status==0) THEN
                        PRINT *, "OUTPUT FILE, ",outfile(m),", CLOSED"
                ELSE
                        PRINT *, "WARNING: The file, ",outfile(m), ", could not be disconnected!"
                END IF
        END DO
        PRINT *, "Program execution completed successfully. EXITING."
        !------------------------------------------------------------------------------------------------------------------------
        !           END OF PROGRAM CLOSING SEQUENCE
        !------------------------------------------------------------------------------------------------------------------------
END PROGRAM nonlin_parabolic_pde
!--------------------------------------------------------------------------------------------------------------------------------
!                                                            END OF MAIN PROGRAM
!--------------------------------------------------------------------------------------------------------------------------------
```

APPENDIX C: PROPERTIES OF ROCKS & MINERALS: TABLES AND FIGURES

Table C- 1. Data relavant to frictional melting from literature survey.

<table>
<tr><th></th><th colspan="2">Parameter</th><th>Data (Units, Comments, Reference)</th></tr>
<tr><th>#</th><th>Symbol</th><th>Definition</th><th></th></tr>
<tr><td>1</td><td>η</td><td>Viscocity of frictional melt</td><td>• <u>Scholz (1990):</u> for dry granitic frictional melt: 10^7-10^8 poise (p. 137).
• <u>Sibson (1975):</u> For basaltic andesite at 1100 ^{0}C and 1 bar: 10^3-10^4 poise (p. 783).</td></tr>
<tr><td>2</td><td>κ</td><td>Thermal diffusivity of host rock</td><td>• <u>Killick & Roering (1998):</u> $1x10^{-6}$ – $1.9x10^{-6}$ m^2/sec for most rock materials (p. 254)
• <u>Sibson (1975):</u> $7.0x10^{-7}$ m^2/sec (p. 784)
• <u>Lachenbruch & Sass (1980):</u> $1x10^{-7}$ m^2/sec (p. 6187).</td></tr>
<tr><td>3</td><td>μ</td><td>Coefficient of friction between fault/slip surfaces</td><td>• <u>Killick & Roering (1998):</u> 0.6 – 0.85; 0.85, for σ_n < *200 MPa* (p. 253) – : <u>Byrelee's (1978)</u> results.</td></tr>
<tr><td>4</td><td>ν</td><td>Poisson's Ratio</td><td>• <u>Jaegar & Cook (1979):</u> W. Granite: 0.11 (Table 6.2.1, p. 146);
• <u>Wang & Scholz (1994):</u> 0.21 (p. 6793)
• <u>Touloukinan et. al. (1981):</u> In Gpa (Table 6.1, p. 135)
<table>
<tr><td>Quartzite</td><td>0.10-0.30</td></tr>
<tr><td>Granite/</td><td>0.09-0.48</td></tr>
<tr><td>Diorite</td><td>0.05-0.29</td></tr>
<tr><td>Gneiss</td><td>0.06-0.13</td></tr>
<tr><td>Schist</td><td>0.01-0.15</td></tr>
</table></td></tr>
<tr><td>5</td><td>ρ</td><td>Density of host rock</td><td>• <u>Killick & Roering (1998):</u> 2700-2820 kg/m^3.(p. 254)
• <u>Sibson (1975):</u> 2800 kg/m^3 (p. 786)
• <u>Cardwell et al. (1978):</u> 2800 kg/m^3 (p. 527)
• <u>McKinzie and Brune (1972):</u> 3000 kg/m^3 (p. 74)</td></tr>
<tr><td>6</td><td>$\Delta\sigma_{s/d}$</td><td>Stress Drop (static/dynamic)</td><td>• <u>Kanamori (1994):</u>
  ▫ <u>Static drop:</u> 30-100 bars (p. 209);
  ▫ <u>Static drop:</u> 10-100 bars over large scales (or profile lengths, p. 215); 300-2000 bars for the 1990 Pasadena, CA earthquake, over a profile length of about 0.5 km; 150-300 bars for the Sierra Madre, CA earthquake, over a profile length of about 4 km (p. 218).
  ▫ <u>Dynamic drop:</u> Average over whole quake area: 12-40 bars; Local range: 22-84 bars; point range: 40-200 bars.
• <u>Lachenbruch & Sass (1980):</u> Stress drop based on heat flow calculations and seismic observations: 0-100 bars (p. 6206).</td></tr>
</table>

(CONTINUED)

Table C-1. Data relavant to frictional melting from literature survey. (CONTINUED)

#	Parameter: Symbol	Parameter: Definition	Data (Units, Comments, Reference)	
7	σ_c	Compressive strength	*Scholz (1990):*– Uniaxial compressive strength: Quartz: 2200 MPa; Calcite: 200 MPa (p. 61).	
8	σ_n, σ_v	Normal or vertical stresses on the fault (average)	• *Sibson (1975):* $\sigma_n = 1.6\ \rho gz$, for optimal thrust faulting; Differential stress approx. = 3.2 kbar (p. 790) • *McKinzie and Brune (1972):* >10 bars for frictional melting (p. 74) • Kanamori (1994): About 200 bars or even less than 100 bars for the San Andreas Fault system. • Turcotte & Tag (1980): About 100 bars.	
9	τ_y	Yield strength in shear, for host rock. **$H=(nM)^3=3\sigma_y = 6\tau_y$** **n=1.3 - 1.6**	*Spray (1992):* In MPa (Table 1, p. 210):	
			Micas (Muscovite & Biotite)	167-333
			Serpentine (lizardite & chrysotile)	200
			Amphiboles: actinolite & tremolite horneblende & parg	567-833 750
			Pyroxenes: clinopyroxene orthopyroxene	750-1083 567-833
			Feldspar: Orthoclase Albite & Anorthite	833 833-1083
			Silicon Dioxide (Quartz):	1400
			Olivine (Forsterite)	1083-1400
			Zircon	1667
			Soda-Lime glass	1800
			Rutile	1083
			Corundum	3333
			Diamond	25,000
			Titanium	85
10	σ_y	Yield strength in tension, for host rock **$H=(nM)^3=3\sigma_y = 6\tau_y$** **n=1.3 - 1.6**	• *Sibson (1975):* For Gneiss, ANISOTROPIC strengths: σ_{45} = 4.2 kbar; σ_{90} = 8.4 kbar (p. 779) • *Spray (1992):* In Mpa (Table 1, p. 210):	
			Micas (Muscovite & Biotite)	333-666
			Serpentine (lizardite & chrysotile)	400
			Amphiboles: actinolite & tremolite horneblende & parg	1133-1666 1500
			Pyroxenes: clinopyroxene orthopyroxene	1500-2166 1133-1666
			Feldspar: Orthoclase Albite & Anorthite	1666 1666-2166
			Silicon Dioxide (Quartz): Natural Synthetic	2800
			Olivine (Forsterite)	2166-2800
			Zircon	3333
			Soda-Lime glass	3600
			Rutile	2166
			Corundum	6666
			Diamond	50,000
			Titanium	170

(CONTINUED)

Table C-1. Data relavant to frictional melting from literature survey. (CONTINUED)

<table>
<tr><th></th><th colspan="2">Parameter</th><th colspan="2">Data (Units, Comments, Reference)</th></tr>
<tr><th>#</th><th>Symbol</th><th>Definition</th><th colspan="2"></th></tr>
<tr><td>11</td><td>ξ</td><td>Strain rate for fault</td><td colspan="2">• <u>Spray (1992):</u> 10^{-2} – 1 for coseismic slip; $>10^{3}$ for meteorite impact (Figure 2, p. 209)
• <u>Kanamori (1994):</u> 10^{-4}</td></tr>
<tr><td>12</td><td>C</td><td>Cohesive strength of the fault</td><td colspan="2"><u>Killick & Roering (1998):</u> 1-54 Mpa (= 2S, Figure 6, p. 256, and also below).</td></tr>
<tr><td>13</td><td>C_P</td><td>Specific heat at constant pressure for host rock</td><td colspan="2">• <u>Killick & Roering (1998):</u> 1200 $JKg^{-1}K^{-1}$ (p. 254).
• <u>Cardwell et al. (1978):</u> 1050 $JKg^{-1}K^{-1}$ (p. 527).
• <u>McKinzie and Brune (1972):</u> 1000 $JKg^{-1}K^{-1}$ (p. 74)</td></tr>
<tr><td>14</td><td>d</td><td>Crustal depth of pseudotachylyte formation</td><td colspan="2">• <u>Swanson (1992):</u> 0-18 km below the surface (Fig.1); Crystalline PT: < 5 km; Glassy PT: >5 km; Mylonitic zone, plastically deformed PT: 10-15 km.
• <u>Killick & Roering (1998):</u> 1.9-6.6 km below paleo land surface. 3.3-6.1 km under lithostatic loading, and 9.3-17.2 km under hydrostatic loading. The values depend on the mole fraction of water and mass fraction of CO_2 in host rock. (p.250-1)
• <u>Sibson (1975):</u> 1-10 km (p. 784); > 2-3 km (p. 786); most likely depth at 4-5 km (p. 791).</td></tr>
<tr><td rowspan="7">15</td><td rowspan="7">E, E_y</td><td rowspan="7">Young's Modulus</td><td colspan="2">• <u>Jaegar & Cook (1979):</u> Quartz Diorite: 3 x 10^{6} psi (= 0.0068915*3000000 Mpa = 20.67 Gpa), Granite: 2 3 x 10^{6} psi (= 13.78 GPa) [p. 188, Sec. 6.14, Fig. 6.15.1]; W. Granite: 8.1 x 10^{6} psi (55.81 Gpa) {Table 6.2.1, p. 146];
• <u>Wang & Scholz (1994):</u> Westerly Granite 69 Gpa (p. 6793)
• <u>http://www.almazoptics.com/homepage/Quartz.htm:</u> Quartz: 76 GPa (perp.), and 97 GPa (para.) – optical quality.
• <u>http://www.tosoh.com/EnglishHomePage/tgg/genprop.htm</u>: Quartz glass: 70-74 GPa.
• <u>Touloukinan et. al. (1981):</u> In Gpa (Table 6.1, p. 135; Fig. 6.27, p. 168)</td></tr>
<tr><td>Quartzite</td><td>14.34-68.95</td></tr>
<tr><td>Granite/ Westerly Granite</td><td>p_{conf} = 0 Mpa: 5.52 – 64.10
p_{conf} = 500 Mpa: 75 (from slope in above figure, at 25^{0} C,), 55 at 300^{0}C, & 40 at 500^{0}C.</td></tr>
<tr><td>Granodiorite</td><td>45.10-70.80</td></tr>
<tr><td>Diorite</td><td>4.09-103.1</td></tr>
<tr><td>Gneiss</td><td>12.68-67.22</td></tr>
<tr><td>Schist</td><td>39.30-80.67</td></tr>
</table>

(CONTINUED)

Table C-1. Data relavant to frictional melting from literature survey. (CONTINUED)

	Parameter		Data (Units, Comments, Reference)	
#	**Symbol**	**Definition**		
16	***H***	Indentation / Penetration hardness **$H=(nM)^3=3\sigma_y = 6\tau_y$** **n=1.3 - 1.6**	• *Scholz (1990):*– Calcite: 600 MPa; Sandstone: 2000 MPa – Logan and Teufel's results (p. 61). • *Spray (1992):* In kg/mm^{-2} (= Mpa) (Table 1, p. 210):	
			Micas (Muscovite & Biotite)	100-200
			Serpentine (lizardite & chrysotile)	120
			Amphiboles: actinolite & tremolite horneblende & parg	340-500 450
			Pyroxenes: clinopyroxene orthopyroxene	450-650 340-500
			Feldspar: Orthoclase Albite & Anorthite	500 500-650
			Silicon Dioxide (Quartz): Natural Synthetic	840
			Olivine (Forsterite)	650-840
			Zircon	1000
			Soda-Lime glass	840
			Rutile	650
			Corundum	2000
			Titanium	50
17	***k***	Thermal conductivity of host rock	• *Sibson (1975):* 2 $W/m^{-1}\ {}^0C^{-1}$ (p. 786) • *McKinzie & Brune (1972):* 2 $W/m^{-1}\ {}^0C^{-1}$ (p. 74). • *Killick & Roering (1998):* Quartzites: 3.45-6.42 $W/m^{-1}\ {}^0C^{-1}$ (p. 254) • *Lachenbruch & Sass (1980):* 2.5 $W/m^{-1}\ {}^0C^{-1}$ (p. 6187). • *Spray (1992):* In $W/m^{-1}\ {}^0C^{-1}$ (Table 1, p. 210):	
			Micas: Muscovite Biotite	1.3 0.8
			Serpentine: lizardite chrysotile	1.34 3.0
			Amphiboles: actinolite tremolite horneblende	1.22 2.78 1.4-1.8
			Pyroxenes: clinopyroxene orthopyroxene	2.4-3.1 2.4-2.86
			Feldspar: Orthoclase Albite Anorthite	1.35 1.35 0.85
			Silicon Dioxide (Quartz): Natural Synthetic	4.3
			Olivine (Forsterite)	2.96
			Zircon	2.6
			Soda-Lime glass	1.0
			Rutile	2.9
			Corundum	13.0
			Diamond	63-93
			Titanium	22

(CONTINUED)

Table C-1. Data relavant to frictional melting from literature survey. (CONTINUED)

#	Parameter: Symbol	Parameter: Definition	Data (Units, Comments, Reference)		
18	K_c	Fracture toughness	*Spray (1992):* In MPa/m$^{-1/2}$ (Table 1, p. 210):		
			Feldspar (orthoclase)	1.3	001
			Natural quartz	2.4	perp. to the <c> direction
			Synnthetic quartz	0.8-1.0	perpendicular to *r* and *z* directions.
			Olivine	0.59 0.73	010 001
			Soda-Lime glass	0.7	
			Corundum	3.0	
			Diamond	3.4-3.9	111
			Titanium	≥50!	
19	L_F	Length of fault veins	• *Swanson (1992):* 1-10 m long • *Curewitz & Karson (1999):* Sometimes >20 m (p. 1695) • *Grocott (1981):* up to 1 km long! (p. 169) • *Sibson (1975):* Approx. 10 cm. (p. 778)		
20	l_I	Length of injection veins	• *Curewitz & Karson (1999):* Typically, 1 m (p. 1695) • *Sibson (1975):* Approx. 0.1-1 cm. (p. 778)		
21	M	Mohs hardness **$H=(nM)^3=3\sigma_y = 6\tau_y$** **$n=1.3 - 1.6$**	*Spray (1992):* (Table 1, p. 210):		
			Micas (Muscovite & Biotite)	2.5-4	
			Serpentine (lizardite & chrysotile)	3	
			Amphiboles: actinolite & tremolite horneblende & parg	5-6 5.5	
			Pyroxenes: clinopyroxene orthopyroxene	5.5-6.5 5-6	
			Feldspar: Orthoclase Albite & Anorthite	6 6-6.5	
			Silicon Dioxide (Quartz): Natural Synthetic	7	
			Olivine (Forsterite)	6.5-7	
			Zircon	7.5	
			Soda-Lime glass	7	
			Rutile	6.5	
			Corundum	9	
			Diamond	10	
			Titanium	2	
22	P	Pressure in pore fluid at pseudotachylyte formation depths	• *Killick & Roering (1998):* $P_{confining} = f(W_{H2O})$; $P_{confining} = g(W_{CO2})$. Based on PT without any vesicles or bubbles, the confining pressures must counter the solubility pressure given by these relations. Depending on water and CO_2 content in local rocks, these pressures were hypothesized to vary between 92 MPa and 142 MPa. (p. 250-251). Also P approx. = 0.335 σ_n (for hydrostatic conditions) and 0.9σ_n (for lithostatic conditions). • *Sibson (1975):* Pore fluid pressure rise = (Temperature rise/47) kbars, for water initially at 140 ^{0}C (close to homogenization), and depth of 4-5 km. 50 ^{0}C rise in temp corresponds to a 1kbar overpressurization.		

(CONTINUED)

Table C-1. Data relavant to frictional melting from literature survey. (CONTINUED)

	Parameter		Data (Units, Comments, Reference)
#	**Symbol**	**Definition**	
23	***Q***	Heat flux	*Scholz (1990):– $Q= \tau_f . U$ =50Mpa x 1 cm/sec; OR 0.016 W/m^2 typically.* Good chunk of W_f (p. 114)
24	***r***	Clast size (radius/major axis)	• *Curewitz & Karson (1999):* 10µm – 1m (p. 1696 & 99) • *Shimamoto & Nagahama (1992):* 5 – 2000 µm (graphs)
25	***S***	Tensile strength of the fault	*Killick & Roering (1998):* 0.5-27 Mpa (Figure 6, p. 256).
26	***t_0***	Time duration for melting/ duration of fault motion	• *Swanson (1992):* Melting duration approx.= 10^4 sec; Rupture duration approx. = 1.2-12 sec. (Figure 2) • *Sibson (1975):* Cooling times = 0.4 – 40 s (p. 778). • *Cardwell et al. (1978):* Duration of faulting approx. = 1 sec (p. 527).
27	***T_{max}***	Maximum frictional melt temperatures	• *Swanson (1992):* (p. 227) ▫ $T_{plastic\ transition}$ (Quartz) = 300 ^{0}C; $T_{plastic\ transition}$ (Feldspar) = 450 ^{0}C ▫ T_{peak} estimate of 1000 ^{0}C from hotrock melt temperatures and theoretical calculations (Cardwell, et al.(1978), and McKinzie & Brune (1972)); ▫ T_{peak} estimate of 1520 ^{0}C from SiO_2 glass compositions; ▫ T_{peak} estimate of 1400 ^{0}C from flash melting during welding; ▫ T_{peak} estimate of 1180 ^{0}C from thermal dye measurements by Logan and Teufel (1986); • *Curewitz & Karson (1999):* $T_{homologous}$ (sintering temperature) = 0.6-0.7 T_{melt}; About 700-900 ^{0}C for granitic melts with rounded clasts in PT (p. 1705); >900 ^{0}C for glassy PT (p. 1707). • *Killick & Roering (1998):* From Carslaw & Jaegar (1959) and Sibson (1975): T_{max}- $T_{ambient}$ = f(Q/t$^{1/2}$) ; and gives, about 1000 ^{0}C (p. 255). • *Sibson (1975):* 1100-1200 ^{0}C, from embayment of plagioclase porphyroclasts (p. 783). • *Cardwell et al. (1978):* $T_{ambient}$ = 400 ^{0}C; T_{melt} = 800 ^{0}C (p. 529). • *McKinzie & Brune (1972):* T_{melt} = 1000 ^{0}C (p. 74).
28	***T_F***	Thickness of fault veins	• *Curewitz & Karson (1999):* < 2 cm; Reservoir zones, > 10 m (p. 1695). • *Grocott (1981):* Distance between paired shears: Typical: .15-1.5 m; Actual, field: 2-3 cm – 3 m. (p. 169 & 171)
29	***t_I***	Thickness of injection veins	*Curewitz & Karson (1999):* About 2 cm (p. 1695).

(CONTINUTED)

Table C-1. Data relavant to frictional melting from literature survey. (CONTINUED)

	Parameter		**Data (Units, Comments, Reference)**	
#	**Symbol**	**Definition**		
30	$\boldsymbol{T_m}$	Mineral melt temperatures	*Spray (1992):* In ^{0}C (Table 1, p. 210):	
			Micas: (Muscovite & Biotite)	650
			Serpentine: (lizardite & chrysotile)	400
			Amphiboles: actinolite tremolite horneblende parg	750 850 750 1000
			Pyroxenes: clinopyroxene orthopyroxene	1400 1425
			Feldspar: Orthoclase Albite Anorthite	1150 1100 1550
			Silicon Dioxide (Quartz): Natural Synthetic	1730
			Olivine (Forsterite)	1890
			Zircon	1695
			Soda-Lime glass	1000
			Rutile	1825
			Corundum	2000
			Diamond	3727
			Titanium	1667
31	$\boldsymbol{U}$	Fault displacement (slip) velocities	• *Swanson (1992):* < 1 m/s (p. 227). • *Curewitz & Karson (1999):* >0.1 m/s for coseismic slip (from Magloughlin & Spray (1992) and Spray (1995)) (p. 1694). • *Spray (1992):* 0.1-2 m/s for coseismic slip (p. 212). • *Sibson (1975):* > 0.1 m/s; typically, .5 m/s (p. 786). • *Grocott (1981):* 0.1-1 m/s (based on Sibson (1975)). • Kanamori (1994): Typically < 1m/s; (1-92 cm/sec observed in field): Maximum about 2 m/s (p. 219). • *Turcotte & Tag (1980):* For San Andreas, plate velocity = 5.0-5.5 cm/yr (!) (p. 6224 & 6229). • *Lachenbruch & Sass (1980):* Plate velocity for San Andreas = 4.0-5.0 cm/yr.	
32	$\boldsymbol{W_{CO2}}$	CO_2 wt % in host rock	*Killick & Roering (1998):* 0.1 % (w/w) (p. 250-251).	
33	$\boldsymbol{W_f}$	Total fault energy	• *Scholz (1990):*– $W_f = W_s + W_g + W_R + Q$ (p. 114)	
34	$\boldsymbol{W_g}$	Gravitational work	• *Scholz (1990):*– Negligible (p. 114)	
35	$\boldsymbol{W_{H2O}}$	H_2O wt %	*Killick & Roering (1998):* 0.48-2.33 % (w/w) (p. 250-1).	
36	$\boldsymbol{W_R}$	Seismic (Reflected) energy	• *Scholz (1990):*– varies from fault to fault (p. 114)	
37	$\boldsymbol{W_s}$	Surface energy	• *Scholz (1990):*– approximately $10^{-3} – 10^{-4}$ of W_f. (p. 114). • *Lachenbruch & Sass (1980):* 10^{-2} of W_f. (p. 6218)	

Figure C- 1. QUARTZ: Specific Heat, *[1 - {C_p(T)/1500}]* as a function of Temperature, T. The value at 300 K, or typical ambient conditions, is marked with the dotted line. In some of the runs illustrated in Table A-7, this cutoff was assumed, resulting in a discontinuity at 300 K.

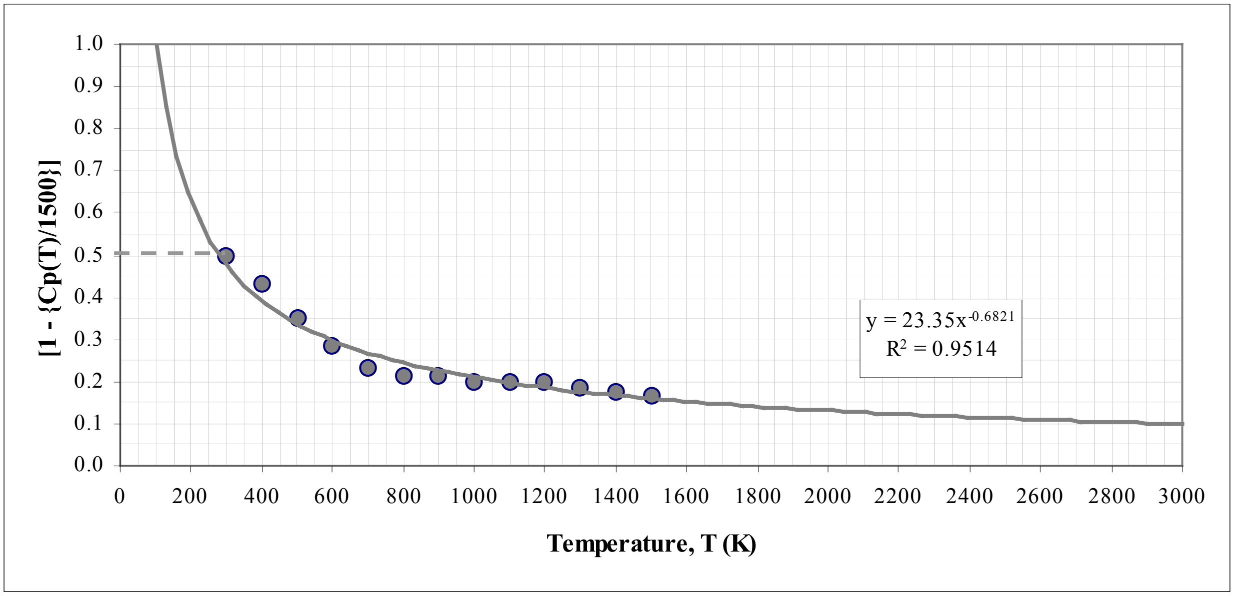


Figure C- 2. QUARTZ: Thermal Conductivity, *(k(T) - 1)* as a function of Temperature, T. The value at 300 K, or typical ambient conditions, is marked with the dotted line. In some of the runs illustrated in Table A-7, this cutoff was assumed, resulting in a discontinuity at 300 K.

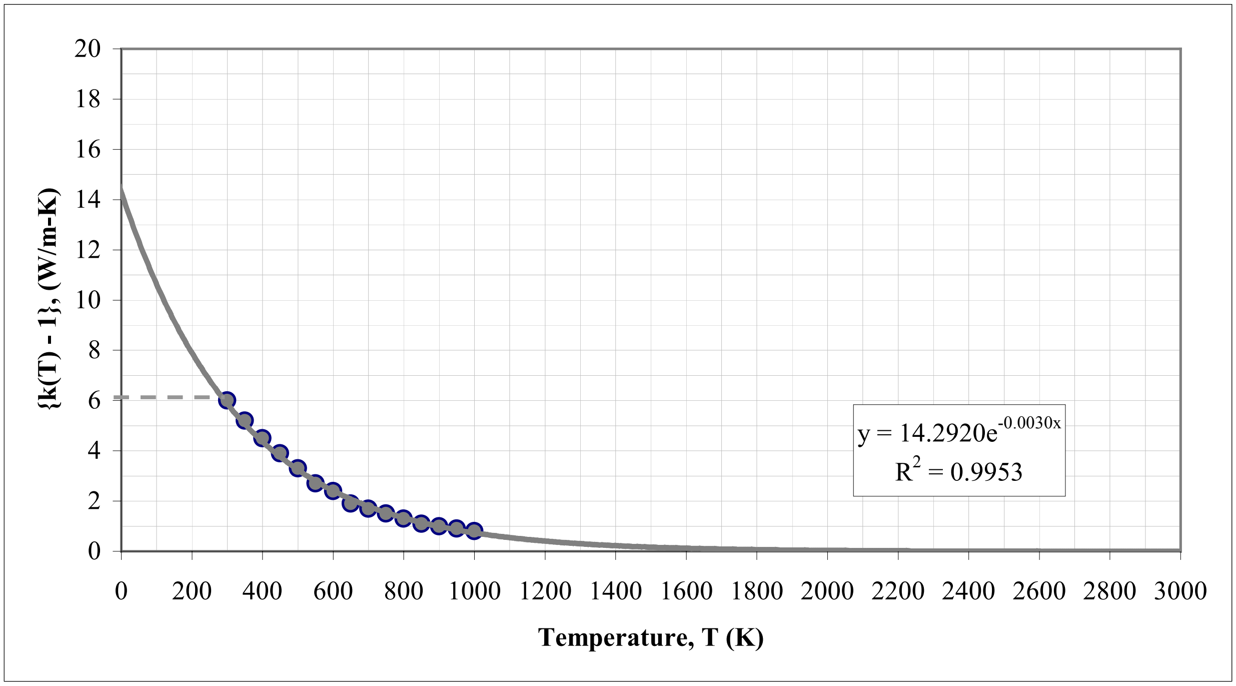


Figure C- 3. FELDSPARS: Specific Heat, *[1 - {$C_p(T)/1000$}]* as a function of Temperature, T. The value at 300 K, or typical ambient conditions, is marked with the dotted line

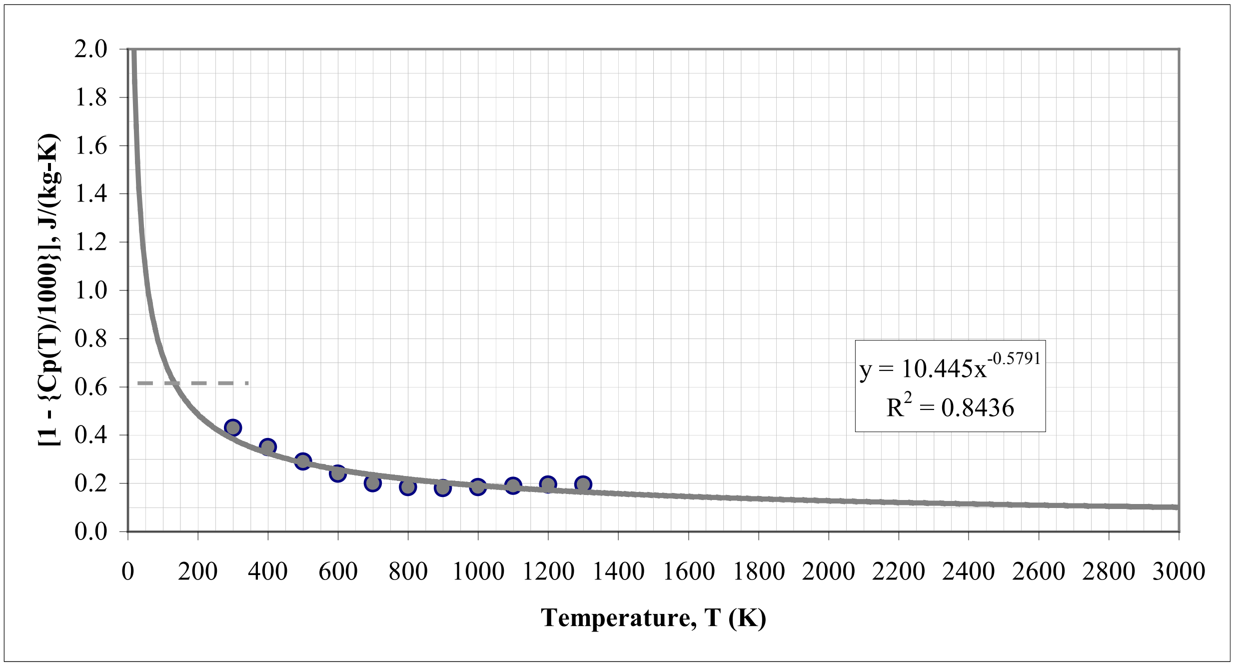
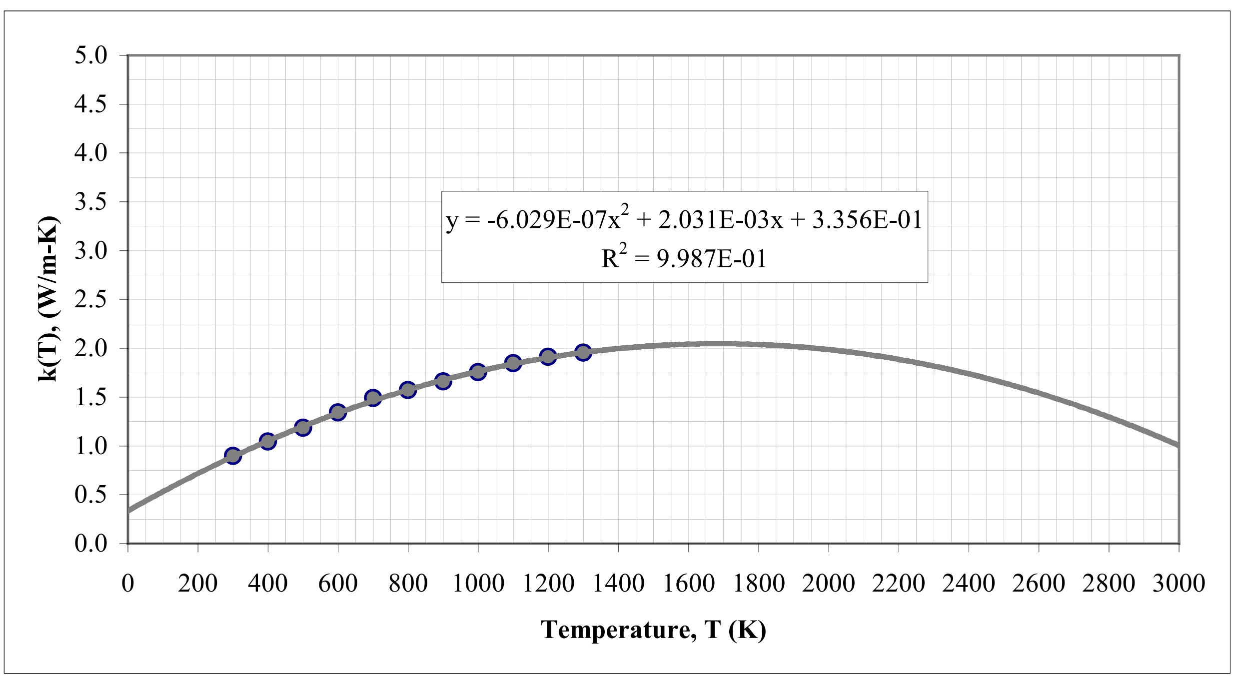


Figure C- 4. FELDSPAR: Thermal Conductivity, *k(T)* as a function of Temperature, T. The value at 300 K, or typical ambient conditions, is marked with the dotted line.



APPENDIX D:MATLAB POST-PROCESSING CODES

FOUR MATLAB POST PROCESSING CODES

FOR FORTRAN 90 OUTPUT FILES (APPENDIX A)

CODE D-1. *DevolRuns.m*: Matlab code for processing FORTRAN 90 output file *DEVOL* (temporal evolution of peak temperatures).

```
% This program reads time evolution data for the specified number of resolutions, and creates a space delimited file for each time
% step. This file can be subsequently for plotting X-Y semi-log scatter plot where the axes for each data set are for the same
% parameters, but of different lengths. The program accomplishes this using string searches for key words, to identify the
% start and end of data at each resolution.

format short e
evoldata = zeros(50,10);
inpfilebeg = 'Devol_';
inpfilemid = '_2_';

% Ask for number of input files:
inpath = input('Type the absolute path to the directory containing the input files (use backslashes):', 's');
firstfile = input('Type the resolution number (1-2, typically) of the FIRST input file for this set of runs:');
lastfile = input('Type the resolution number (3-5, typically) of the LAST input file for this set of runs:');
numfiles = lastfile - firstfile + 1;
inpfileend = input('Type the ending for this set of runs (e.g., qr5T100, fr1T1000, qr1T50_Lin,...):','s');
basetemp = input('Type the value of the ambient temperature - 360, usually, but 330 if D1km:');
for res = firstfile:lastfile                       % FOR LOOP to combine snapution data for all "numfile" resolutions.
    inpfile = [inpath,'\',inpfilebeg,int2str(res),inpfilemid,inpfileend];
    fid = fopen(inpfile, 'r');
    lcount = 0
    while lcount <= 12
        line = fgetl(fid);
        disp(line)
        lcount = lcount + 1;
        if lcount == 8                          % Ignore lines 1-7.
            string = line(32:45);               % Read x-left from line 8.
            string = lower(string);
                xl = str2num(string);
            string = line(47:60);               % Read x-right from line 8.
            string = lower(string);
                xr = str2num(string);
        end
        if lcount == 9
            string = line(32:45);              % Read y-bottom from line 9.
            string = lower(string);
                yb = str2num(string);
            string = line(47:60);               % Read y-top from line 9.
            string = lower(string);
            yt = str2num(string);
        end
        if lcount == 10
            string = line(35:48);               % Read t_initial from line 10.
            string = lower(string);
                ti = str2num(string);
            string = line(50:63);               % Read t_final from line 10.
            string = lower(string);
                tf = str2num(string);
        end
        if lcount == 11
            string = line(27:40);               % Read hx from line 11.
            string = lower(string);
                hx = str2num(string);
        end
        if lcount == 12
            string = line(27:40);               % Read hy from line 12.
            string = lower(string);
                hy = str2num(string);
        end
        if lcount == 13
            string = line(27:40);               % Read k from line 13.
            string = lower(string);
                ht = str2num(string);
        end
    end
    % Compute the total number of time steps based on the time step size and time limits for this run:
    steps = ( (tf - ti)/ht ) + 1.0;
    numsteps = round(steps)
    if abs(numsteps - steps) >= 0.5
        t_steps = numsteps + 1;
    else
        t_steps = numsteps;
    end
% Beginning line 14, start reading each line. If it contains the string "Maximum Temperature", it marks the beginning of EVOL Data.
    while feof(fid) == 0
        line = fgetl(fid);
        if isempty(findstr('Maximum Temperature',line)) == 1
            disp(line)
            lcount = lcount + 1;
        else
            % READ EVOLUTION DATA.
            disp(line)
            lcount = lcount + 1;
            for i = 1:2                                % IGNORE THE NEXT TWO HEADER LINES.
                line = fgetl(fid);
                lcount = lcount + 1;
                disp(line)
            end
            row = 0;                                   % Initialize Data Row counter.
            endstr = [' ',num2str(t_steps),' '];
```

```
            while isempty(findstr(endstr,line)) == 1      % Read evolution data.
                line = fgetl(fid);
                lcount = lcount + 1;
                row = row + 1;
                for i = 1:2                             % Do loop for the two data fields, time and T_max.
                    if i == 1
                        col = 2*res - 1;
                        string = line(14:25);           % Read time field
                        string = lower(string);
                    else
                        col = 2*res;
                        string = line(48:61);           % Read Max. Temperature field
                        string = lower(string);
                    end
                    evoldata(row,col) = str2num(string);
                end
            end
            while isempty(findstr('GLOBAL TEMPERATURE MAXIMA',line)) == 1
                line = fgetl(fid);
                lcount = lcount + 1;
                disp(line)
            end
            line = fgetl(fid);              % Once the Header for the global max. Temperature is found, ignore the next line.
            lcount = lcount + 1;
            disp(line)
            line = fgetl(fid);              % FINAL DATA LINE.
            lcount = lcount + 1;
            row = row + 1;                  % FINAL DATA ROW.
            for i = 1:2                             % Do loop for the two data fields, time and GLOBAL T_max.
                if i == 1
                    col = 2*res - 1;
                    string = line(14:25);           % Read time field
                    string = lower(string);
                else
                    col = 2*res;
                    string = line(48:61);           % Read Max. Temperature field
                    string = lower(string);
                end
                evoldata(row,col) = str2num(string);
            end
        end         % IF LOOP for Data entry into "evoldata" array.
    end             % EOF WHILE LOOP for each resolution file.
end                 % Data assimilation FOR LOOP
% Since the Global maxima is output separately by the FORTRAN 90 Code, evoldata needs to be sorted first before it can be used.
tempcount = 0;
for res = firstfile:lastfile
    for j = 1:size(evoldata,1)
        if evoldata(j,2*res) ~= 0.0                     % COUNT THE NUMBER OF ROWS WITH NON-ZERO TEMPERATURE.
            tempcount = tempcount + 1;
        end
    end
    temp = zeros(tempcount,2);
    tempcount = 0;
    for j = 1:size(evoldata,1)
        if evoldata(j,2*res) ~= 0.0                     % MAKE SURE THE 0 (Zero) ELEMENTS ARE NOT SORTED AND REMAIN AT THE BOTTOM.
            tempcount = tempcount + 1;
            temp(tempcount,1) = evoldata(j,2*res-1);
            temp(tempcount,2) = evoldata(j,2*res  );
        end
    end
    [tmp,idcol] = sort(temp(:,1));
    temp = temp(idcol,:);
    evoldata(:,2*res-1) = zeros(size(evoldata,1),1);
    evoldata(:,2*res) = zeros(size(evoldata,1),1);
    tempcount = 0;
    for j = 1:size(temp,1)
        tempcount = tempcount + 1;
        evoldata(j,2*res-1) = temp(tempcount,1);
        evoldata(j,2*res  ) = temp(tempcount,2);
    end
    tempcount = 0;
end
% FINALLY, SAVE THE SORTED AND INITIAL-TIME "CORRECTED" PEAK TEMPERATURE EVOLUTION DATA IN A SEPARATE FILE. Clear array from
workspace.
eval( ['save ',inpfilebeg,inpfileend,'.dat evoldata -ascii'], ['Error saving EVOLUTION Data file!'] )
eval( 'clear evoldata','Error deleting temporary GRID DATA array from Workspace!')

% Now read in the evoldata file:
inpdata = dlmread([inpfilebeg,inpfileend,'.dat'], ' ');

% Separate into X and Y arrays for ease of sorting, and plotting data:
X = [inpdata(:,1) inpdata(:,3) inpdata(:,5) inpdata(:,7) inpdata(:, 9)];
Y = [inpdata(:,2) inpdata(:,4) inpdata(:,6) inpdata(:,8) inpdata(:,10)];
% Compute maxima and/or minima as required, for determining axes limits:
[ymax,i] = max(Y(:));
[xmax,i] = max(X(:));
%[xmin,i] = min(X(:));
x = X(:);   % Since there are a number of 0 (Zero) valued terms in inpdata array, the minimum CANNOT be found with the "min" function.
xmin = 1;

for j = 1:size(x,1)
    if (x(j) > eps) & (x(j) < xmin)
        xmin = x(j);
    end
end
```

```
xmax = ceil(log10(xmax));        % Round towards +INFINITY.
xmax = 10.0^(xmax);
xmin = floor(log10(xmin)) - 1;   % Round towards -INFINITY.
xmin = 10.0^(xmin);
X(1,:) = xmin;                   % The Fortran 90 code begins at initial time = 0, so for the semi-log plot, set this to a small
value.
% Estimate a y plot grid spacing, based on the current data file.
yincr = (ymax-basetemp)/10.0;
if yincr > 1.0
    order = fix(log10(yincr));        % Round exponent towards 0 (ZERO) (to get the order of magnitude of "yincr") if increment is > 1
else
    order = floor(log10(yincr));      % Round exponent towards -INFINITY (to get the order of magnitude of "yincr") if increment is <
1
end
yincr = (10.0^order)*round(yincr/10.0^order);

% NOW PLOT THE SORTED DATA:
semilogx(X(:,1),Y(:,1),'b:x',X(:,2),Y(:,2),'g:^',X(:,3),Y(:,3),'m-.s',X(:,4),Y(:,4),'k--d',X(:,5),Y(:,5),'r-o');
x_label = 'Time (s)';
y_label = 'Maximum Temperature (K)';
set(gca,'Title',text('String',inpfileend),...
        'GridLineStyle','-',...
        'Layer','top',...
        'XColor',[0.5,0.5,0.5],...
        'YColor',[0.5,0.5,0.5],...
        'XLim',[xmin xmax],...
        'YLim',[basetemp (ymax + yincr)],...
        'YTick',[basetemp:yincr:(ymax + yincr)]);
grid on;
xlabel('Time (s)'), ylabel('Maximum Temperature (K)');
legend('Resolution 1','Resolution 2','Resolution 3','Resolution 4','Resolution 5',-1);

% Save Plots in different formats:
% (1) Save current figure in Matlab readable format:
%         saveas(gcf, [filestart,'_image_',int2str(res),'.fig'])
% (2)Export current figure to uncompressed tiff format (at specified dpi):
        dpi = 300;
        print(['-r',int2str(dpi)], '-dtiff', [inpfilebeg,inpfileend,'.tif'])
% (3) POSTSCRIPT FILES FOR CONVERTING TO PDF FORMAT:
        print(['-r',int2str(dpi)], '-dpsc', [inpfilebeg,inpfileend,'.ps'])

% ------------------------------------------------ END ---------------------------------------------------------------------
```

CODE D-2. ***DevolPlots.m*****: Matlab code for extracting and plotting convergence data from the files generated by the previous code** ***DevolRuns.m.*** **.**

```
% This program reads time evolution data files already created using the MATLAB file "DevolRuns.m", input in the form of a list file,
% and plots and stores convergence rate metrics. These are output to data, ps and tiff files. It also stores data for the highest
% resolution runs in a separate array for the entire list of files, using the specified Thesis run parameters (r and Tau). This
% array is output as a space delimited data file.

format long e
filemax = 25;
filenames = cell(filemax, 1);           % Define cell array for storing input file names.
tmax_out = zeros(5,6);                  % Define and Initialize the array containing max temperature data.

% Ask for number of input files:
dbfile = input('Type the name of the file containing data file names to be processed in this run:', 's');
fid = fopen(dbfile,'r');
filecount = 0;
while feof(fid) == 0                  % Read filenames, count and store them.
    filecount = filecount + 1;
    line = fgetl(fid);
    if filecount == 1
        minrockcode = line(7:7);        % Select mineral/rock code.
    end
    line = deblank(line);               % Remove any trailing or leading blanks from the filename string.
    line = fliplr(line);
    line = deblank(line);
    line = fliplr(line);
    filenames{filecount,1} = line;
    disp(['Processed Filename: ',line])
end
status = fclose(fid);
for file = 1:filecount                 % FOR LOOP for processing input "Devol" files.
    TempRes = dlmread(filenames{file,1}, ' ');

    % First determine the row and column of array "tmax_out" into which the peak temperature value from this file should be input.
    tauend = 0;
    line = filenames{file,1};
    len = length(line);
    run_id = line(7:len-4);
    rstart = 3;
    for i = 1:length(run_id)
        if run_id(i:i) == '_'
            tauend = i-1;
        end
    end
    if tauend == 0
        tauend = length(run_id);
    end
    for i = 1:length(run_id)
        if run_id(i:i) == 'T'
            taustart = i+1;
            rend = i-1;
        end
    end
    len_r = rend - rstart + 1;
    len_tau = tauend - taustart + 1;
    if len_r == 1
        if (str2num(run_id(rstart:rend)) - 1.0) < 1.0e-6
            datrow = 1;
        else
            datrow = 2;
        end
    elseif len_r == 2
        if (str2num(run_id(rstart:rend)) - 10.0) < 1.0e-6
            datrow = 3;
        else
            datrow = 4;
        end
    else
        datrow = 5;
    end
    if len_tau == 2
        if (str2num(run_id(taustart:tauend)) - 10.0) < 1.0e-6
            datcol = 1;
        else
            datcol = 2;
        end
    elseif len_tau == 3
        if (str2num(run_id(taustart:tauend)) - 100.0) < 1.0e-6
            datcol = 3;
        elseif (str2num(run_id(taustart:tauend)) - 200.0) < 1.0e-6
            datcol = 4;
        else
            datcol = 5;
        end
    else
        datcol = 6;
    end

    % Initialize all data arrays and variables.
    numres = (size(TempRes,2))/2;
    Tmax      = zeros(numres,1);
    t_Tmax    = zeros(numres,1);
```

```
    dTmax     = zeros(numres,1);
    dTratio   = zeros(numres,1);
    TmaxOrder = zeros(numres,1);

    % Main calculations for convergence tests.
    for res = 1:numres
        [Tmax(res),i] = max(TempRes(:,2*res));
        t_Tmax(res) = TempRes(i,(2*res-1));
    end
    % Determine the number of resolutions at which output exists for each file.
    res = 2;
    flag = 0;
    while flag == 0
        if Tmax(res) < eps
            numplotdata = res-1;
            flag = 1;
        end
        res = res + 1;
        if (res == numres + 1) & (flag == 0)
            numplotdata = numres;
            flag = 1;
        end
    end
    % Now compute the convergence metrics.
    for res = 1:(numplotdata-1)
        dTmax(res) = abs(Tmax(res+1) - Tmax(res));
    end
    for res = 1:(numplotdata-2)
        dTratio(res) = (dTmax(res))/dTmax(res+1);
        TmaxOrder(res) = (log10(dTratio(res)))/(log10(2.0));
    end

    % Store the above convergence data for each file in its corresponding cell array, and save it to a file.
    tmax_conv_data = [t_Tmax Tmax dTmax dTratio TmaxOrder];

    % SAVE THE PEAK TEMPERATURE CONVERGENCE DATA IN A SEPARATE FILE. Clear array from workspace.
    eval( ['save TmaxConvData_',run_id,'.dat tmax_conv_data -ascii -double'], ['Error saving EVOLUTION Data file!'] )
    eval( 'clear tmax_conv_data','Error deleting temporary CONV DATA array from Workspace!')

    % Save maximum temperature to the tmax_out CELL ARRAY DEFINED ABOVE.
    tmax_out(datrow,datcol) = Tmax(numplotdata,1);
    % 1. PLOT RAW ERROR DATA.
    subplot(2,1,1)
    x = [2:1:numplotdata];
    y = dTmax(1:numplotdata-1);
    y_max = max(y);
    y_min = min(y);
    y_incr = (y_max - y_min)/10.0;
    y_max = y_max + y_incr;
    plot(x,y,'r-o','LineWidth',2)
    set(gca,'Title',text('String',['(a). ',run_id]),...
            'GridLineStyle','-',...
            'Layer','top',...
            'XColor',[0.5,0.5,0.5],...
            'XTick', x,...
            'YColor',[0.5,0.5,0.5],...
            'YLim',[y_min,y_max],...
            'YTick',[y_min:y_incr:y_max]);
    grid on;
    xlabel('Resolution Level, i'), ylabel('dT_{max,i} = T_i - T_{i-1} (K)');

    % 2. PLOT convergence order.
    subplot(2,1,2)
    x = [3:1:numplotdata];
    y = TmaxOrder(1:numplotdata-2);
    if length(y) ~= 1
        y_max = max(y);
        y_min = min(y);
        y_incr = (y_max - y_min)/10.0;
        y_max = y_max + y_incr;
        plot(x,y,'r-o','LineWidth',2)
        set(gca,'Title',text('String',['(b). ',run_id]),...
                'GridLineStyle','-',...
                'Layer','top',...
                'XColor',[0.5,0.5,0.5],...
                'XTick', x,...
                'YColor',[0.5,0.5,0.5],...
                'YLim',[y_min,y_max],...
                'YTick',[y_min:y_incr:y_max]);
        grid on;
        xlabel('Resolution Level, i'), ylabel('Order of Convergence');
    end
    % Save Plots in different formats:
    % (1) Save current figure in Matlab readable format:
    %        saveas(gcf, ['TmaxConvData_',run_id,'.fig'])
    % (2)Export current figure to uncompressed tiff format (at specified dpi):
        dpi = 300;
        print(['-r',int2str(dpi)], '-dtiff', ['TmaxConvData_',run_id,'.tif'])
    % (3) POSTSCRIPT FILES FOR CONVERTING TO PDF FORMAT:
        print(['-r',int2str(dpi)], '-dpsc', ['TmaxConvData_',run_id,'.ps'])
    disp(['Finished Processing Run ID: ',run_id,', file# ',int2str(file),' of ',int2str(filecount),'.'])
    %pause
end
% Finally, save the max temperature data in "tmax_out" into a file.
if tauend < length(run_id)              % Add suffix if the run is linear.
```

```
    fileend = '_Lin'
    eval( ['save TpeakRTauData_',minrockcode,fileend,'.dat tmax_out -ascii -double'], ['Error saving PEAK TEMPERATURE Data file!'] )
else
    eval( ['save TpeakRTauData_',minrockcode,'.dat tmax_out -ascii -double'], ['Error saving PEAK TEMPERATURE Data file!'] )
end

%--------------------------------------------------------- END ----------------------------------------------------------------
```

CODE D-3. *DsnapRuns.m:* Matlab code for processing FORTRAN 90 output file *DSNAP* (temperature profiles along transects parallel to x- and y-axes).

```
% This program reads time x-snap data for the specified number of resolutions, and creates a space delimited file for each time
% step. This file can be subsequently for plotting X-Y scatter plots where the axes for each data set are for the same
% parameters, but of different lengths. The program accomplishes this using string searches for key words, to identify the
% start and end of data at each resolution.

format short e
snapdata = zeros(21,10);
inpfilebeg = 'Dsnap_';
inpfilemid = '_2_';
% Ask for number of input files:
inpath = input('Type the absolute path to the directory containing the input files (use backslashes):', 's');
firstfile = input('Type the resolution number (1-2, typically) of the FIRST input file for this set of runs:');
lastfile = input('Type the resolution number (3-5, typically) of the LAST input file for this set of runs:');
numfiles = lastfile - firstfile + 1;
inpfileend = input('Type the ending for this set of runs (e.g., qr5T100, fr1T1000, qr1T50_Lin,...):','s');
basetemp = input('Type the value of the ambient temperature - 360, usually, but 330 if D1km:');
for res = firstfile:lastfile                       % FOR LOOP to combine snapution data for all "numfile" resolutions.
    inpfile = [inpath,'\',inpfilebeg,int2str(res),inpfilemid,inpfileend];
    fid = fopen(inpfile, 'r');
    lcount = 0
    while lcount <= 12
        line = fgetl(fid);
        disp(line)
        lcount = lcount + 1;
        if lcount == 8                          % Ignore lines 1-7.
            string = line(32:45);               % Read x-left from line 8.
            string = lower(string);
                xl = str2num(string);
            string = line(47:60);               % Read x-right from line 8.
            string = lower(string);
                xr = str2num(string);
        end
        if lcount == 9
            string = line(32:45);              % Read y-bottom from line 9.
            string = lower(string);
                yb = str2num(string);
            string = line(47:60);               % Read y-top from line 9.
            string = lower(string);
            yt = str2num(string);
        end
        if lcount == 10
            string = line(35:48);               % Read t_initial from line 10.
            string = lower(string);
                ti = str2num(string);
            string = line(50:63);               % Read t_final from line 10.
            string = lower(string);
                tf = str2num(string);
        end
        if lcount == 11
            string = line(27:40);               % Read hx from line 11.
            string = lower(string);
                hx = str2num(string);
        end
        if lcount == 12
            string = line(27:40);               % Read hy from line 12.
            string = lower(string);
                hy = str2num(string);
        end
        if lcount == 13
            string = line(27:40);               % Read k from line 13.
            string = lower(string);
                ht = str2num(string);
        end
    end
    % Beginning Line 14, start reading each line until the string "     y        U_ysnap(y)" is found, which marks the beginning
    % of x_snap data.
    while feof(fid) == 0
        line = fgetl(fid);
        if isempty(findstr('     y        U_ysnap(y)',line)) == 1
            disp(line)
            lcount = lcount + 1;
        else
            % READ x_snap DATA.
            row = 0;                                   % Initialize Data Row counter.
            while isempty(findstr('--------------------------',line)) == 1      % Read x_snap data.
                line = fgetl(fid);
                %disp(['PROCESSING THIS LINE:',line])
                lcount = lcount + 1;
                row = row + 1;
                for i = 1:2                            % Do loop for the two data fields, time and T_max.
                    if i == 1
                        col = 2*res - 1;
                        string = line(4:15);           % Read y (THETA) data field
                        string = lower(string);
                    else
                        col = 2*res;
                        string = line(20:35);          % Read Temperature field
                        string = lower(string);
                    end
```

```
                    if row <= 1
                        snapdata(row,col) = str2num(string);
                    elseif (snapdata(row-1,2*res - 1) ~= yt)
                        snapdata(row,col) = str2num(string);
                    end
                end
            end
        end         % IF LOOP for Data entry into "snapdata" array.
    end             % EOF WHILE LOOP for each resolution file.
end                 % Data assimilation FOR LOOP
% FINALLY, SAVE THE SORTED AND INITIAL-TIME "CORRECTED" TEMPERATURE DATA at RIGHT BOUNDARY IN A SEPARATE FILE. Clear array from
workspace.
eval( ['save ',inpfilebeg,inpfileend,'.dat snapdata -ascii'], ['Error saving x_snap Data file!'] )
eval( 'clear snapdata','Error deleting temporary GRID DATA array from Workspace!')

% Now read in the snapdata file:
inpdata = dlmread([inpfilebeg,inpfileend,'.dat'], ' ');

% Separate into X and Y arrays for ease of sorting, and plotting data:
X = [inpdata(:,1) inpdata(:,3) inpdata(:,5) inpdata(:,7) inpdata(:, 9)];
Y = [inpdata(:,2) inpdata(:,4) inpdata(:,6) inpdata(:,8) inpdata(:,10)];

% NOW PLOT THE SORTED DATA:
% Compute maxima and/or minima as required, for determining axes limits:
[ymax,i] = max(Y(:));
% Estimate a y plot grid spacing, based on the current data file.
yincr = (ymax-basetemp)/10.0;
if yincr > 1.0
    order = fix(log10(yincr));        % Round exponent towards 0 (ZERO) (to get the order of magnitude of "yincr") if increment is > 1
else
    order = floor(log10(yincr));      % Round exponent towards -INFINITY (to get the order of magnitude of "yincr") if increment is <
1
end
yincr = (10.0^order)*round(yincr/10.0^order);

xmax = yt;
xmin = yb;
xincr = X(2,1) - X(1,1);

plot(X(:,1),Y(:,1),'b:x',X(:,2),Y(:,2),'g:^',X(:,3),Y(:,3),'m-.s',X(:,4),Y(:,4),'k--d',X(:,5),Y(:,5),'r-o');
set(gca,'Title',text('String',inpfileend),...
        'GridLineStyle','-',...
        'Layer','top',...
        'XColor',[0.5,0.5,0.5],...
        'YColor',[0.5,0.5,0.5],...
        'XLim',[xmin xmax],...
        'XTick',[xmin:xincr:xmax],...
        'YLim',[basetemp (ymax + yincr)],...
        'YTick',[basetemp:yincr:(ymax + yincr)]);
grid on;
xlabel('Theta (radians)'), ylabel('Temperature at Right Boundary (K)');
legend('Resolution 1','Resolution 2','Resolution 3','Resolution 4','Resolution 5',-1);

% Save Plots in different formats:
% (1) Save current figure in Matlab readable format:
%        saveas(gcf, [filestart,'_image_',int2str(res),'.fig'])
% (2)Export current figure to uncompressed tiff format (at specified dpi):
        dpi = 300;
        print(['-r',int2str(dpi)], '-dtiff', [inpfilebeg,inpfileend,'.tif'])
% (3) POSTSCRIPT FILES FOR CONVERTING TO PDF FORMAT:
        print(['-r',int2str(dpi)], '-dpsc', [inpfilebeg,inpfileend,'.ps'])

% ------------------------------------------------ END -------------------------------------------------------------------
```

CODE D-4. Matlab code for processing FORTRAN 90 output file *DGRID* (temperature distribution data at the resolution specified in the FORTRAN 90 code *COND2D*) into 3D temperature surface plots and AVI movies.

```
% FOR HANDLING THE THESIS PROBLEM WHEN hx = hy = 0.1*R:
% This program reads data in the form of grid data over a rectangular domain for each time step, and creates a space delimited file
for each time step.
% This file can be subsequently read by M-Files that can plot the data into Surface/Contour plots. The program accomplishes this using
string searches
% for key words, to identify the start and end of data at each time step.

global z_min

format short e
rowbegin_NumCharIgnore = 19;
inpfilebeg = 'Dgrid_5_2_';

% Ask for input file name:
inpath = input('Type the absolute path to the directory containing the input files (use backslashes): ', 's');
inpfileend = input('Type the ending for this set of runs (e.g., qr5T100, fr1T1000, qr1T50_D1km_Lin...): ','s');
inpfile = [inpath,'\',inpfilebeg,inpfileend];
disp('Next input the z-axis aspect ratio, which determines its relative size w.r.t. the x- and y- axes, and hence the shape')
disp('of the 3D plots. If the z-axis seems scrunched up, keep reducing this value till its size is comarable to the x- and y- axes.')
disp('On the other hand, if z-axis is so big that it dominates that other two yielding a columnar or vertical line plot,')
disp('then do the opposite - Increase this value till the other two axes are restored.');
zaspect = input('Type the aspect ratio for z-axis. Default = 5000000 (5 million). RANGE = 0.01 to 1000000000 (1 billion): ');
log_flag = input('Do you want z data to be converted to log scale (for widely varying orders of magnitude over time)? State y/n: ',
's');
if log_flag == 'y'
    logzero = -10.0;                        % Define how to deal with ZERO or NEGATIVE numbers when using the LOG SCALE for Z-AXIS.
end
max_t = input('Type the maximum number of time levels used in this run. Estimate will do as it is used to initialize the time array:
');
t = zeros(1,max_t);                         % Initialize time level array
fid = fopen(inpfile, 'r');
lcount = 0
while lcount <= 12
    line = fgetl(fid);
    disp(line)
    lcount = lcount + 1;
    if lcount == 8                           % Ignore lines 1-7.
        string = line(32:45);                % Read x-left from line 8.
        string = lower(string);
            xl = str2num(string);
        string = line(47:60);                % Read x-right from line 8.
        string = lower(string);
            xr = str2num(string);
    end
    if lcount == 9
         string = line(32:45);               % Read y-bottom from line 9.
        string = lower(string);
            yb = str2num(string);
        string = line(47:60);                % Read y-top from line 9.
        string = lower(string);
            yt = str2num(string);
    end
    if lcount == 10
        string = line(35:48);                % Read t_initial from line 10.
        string = lower(string);
            ti = str2num(string);
        string = line(50:63);                % Read t_final from line 10.
        string = lower(string);
            tf = str2num(string);
    end
    if lcount == 11
        string = line(27:40);                % Read hx from line 11.
        string = lower(string);
            hx = str2num(string);
    end
    if lcount == 12
        string = line(27:40);                % Read hy from line 12.
        string = lower(string);
            hy = str2num(string);
    end
    if lcount == 13
        string = line(27:40);                % Read k from line 13.
        string = lower(string);
             ht = str2num(string);
    end
end
% Compute the dimensions of the grid data. For these runs, the grid spacings chosen were MAX(hx,hx_max/2) or MAX(hy,hy_max/2). SO,
% for all resolutions except the first one, this results in the latter values being chosen for grid spacing.
x_grid_spacing = 0.5*((xr - xl)/10.0);
y_grid_spacing = 0.5*((yt - yb)/10.0);
steps = ( (yt - yb)/y_grid_spacing ) + 1.0;
nrows = round(steps)
if abs(nrows - steps) >= 0.5
    grid_rows = nrows + 1;
else
    grid_rows = nrows;
end
steps = ( (xr - xl)/x_grid_spacing ) + 1.0;
ncols = round(steps)
```

```
if abs(ncols - steps) >= 0.5
    grid_columns = ncols + 1;
else
    grid_columns = ncols;
end

% Beginning line 14, start reading each line. If it contains the string "TIME STEP", it is the beginning of the next time step grid
data.
tscount = 1;                                    % Initialize time level counter
max_z = -(1.0/eps);                             % Initialize max and min z values. It is optimal to find these values at the same
time
min_z = (1.0/eps);                              % as reading in grid data.
griddata = zeros(grid_rows, grid_columns)
while feof(fid) == 0
    line = fgetl(fid);
    if isempty(findstr('TIME STEP',line)) == 1
        disp(line)
        lcount = lcount + 1;
    else
        % READ TIME LEVEL.
        disp(line)
        lcount = lcount + 1;
        string = line(48:57);
        string = lower(string);
        t(tscount) = str2num(string);           % Compute time level from string.
        tscount = tscount + 1;                  % Update time level counter
        % READ DATA FOR THIS TIME LEVEL.
        for i = 1:2                             % Read HEADER LINES (TWO) for each time step data segment.
            line = fgetl(fid);
            lcount = lcount + 1;
            disp(line)
        end
        for j = 1:grid_rows                     % Read grid data.
            line = fgetl(fid);
            lcount = lcount + 1;
            % THE NEXT TWO STATEMENTS DEAL WITH THE FORTRAN OUTPUT LINES WITH THE FORMAT FMT='(1X,ES18.8,","): So, ignore the blank at
            % the beginning and the "," at the end of each data entry.
            strbegin = rowbegin_NumCharIgnore+2;
            strend = strbegin + 18;
            for i = 1:grid_columns
                string = line(strbegin:strend);
                string = lower(string);
                griddata(j,i) = str2num(string);
                if griddata(j,i) > max_z
                    max_z = griddata(j,i);      % Store current maximum z value
                end
                if griddata(j,i) < min_z
                    min_z = griddata(j,i);      % Store current minimum z value
                end
                strbegin = strend + 2;
                strend = strbegin+18;
            end
        end
        % The FORTAN 90 CODE DOES NOT GENERATE DATA FOR yt = 3.15. So, copy data from y= 3.10 into the last row of griddata.
        if yt == pi
            for i = 1:grid_columns
                griddata(grid_rows, i) = griddata(grid_rows-1, i);
            end
        end
        for i = 1:1           % Read FOOTER LINES for each time step data segment. This includes the last data line which is a repeat.
            line = fgetl(fid);
            lcount = lcount + 1;
            disp(line)
        end
        % SAVE DATA AT THIS TIME LEVEL IN A SEPARATE FILE.
        eval( ['save ',inpfile,'_',int2str(tscount-1),' griddata -ascii'], ['Error saving file for time loop#',int2str(tscount),'!'] )
        eval( 'clear data','Error deleting temporary GRID DATA array from Workspace!')
    end
end

% Now read in the space-delimited GRID data in a rectangular grid representing [r,THETA] space (r = 0-1, THETA=0-PI)
% and (a) extend the data symmetrically over the FULL circle, (b) then plot a POLAR MESH-CONTOUR plot & a COLOR plot of the data, AND
% (c) save the figures in FIG files along with exporting them to TIFF images for word processing applications.
% REQUIRES "MESHC_ZCONTOUR.m", A VARIANT OF THE MATLAB FUNCTION "MESHC", TO CONTROL THE DISTANCE BETWEEN THE MESH PLOT AND
% CONTOUR MAP PLANE. This is accomplished using the global variable, "z_min".

% INPUT FILE SEPCS.
max_files = tscount-1;
filestart = [inpfile,'_'];

% Generate the x,y grid for the POLAR DATA ABOVE, and redefine the lower limit of the y axis to -yt. This will mean redefining
% the number of y grid points on the extended axis.
yb = yb - yt;
grid_rows = grid_rows + (grid_rows - 1);
[th,r] = meshgrid(yb:y_grid_spacing:yt, xl:x_grid_spacing:xr);
[X,Y] = pol2cart(th,r);                 % Convert Polar coordinates to Cartesian Coordinates for creating the plots.
% Define Rows/2 which will be used in reshaping the data array, and in extending the THETA field.
grid_rows_by_2 = (grid_rows + 1)/2      % "grid_rows" is always ODD.

% Compute the limits & tick marks along the x- and y- axes for POLAR plot representation. Define the coordinat limits so they
% completely enclose the segment of the disc being considered: x_left = r_min*COS(Theta_max) & |y_max| = r_max*SIN(Theta_max)
x_right = xr;
epxflag = 0;
epx = 0;
```

```
while epxflag == 0
    if abs(xl*10.0^epx - fix(xl*10.0^epx)) < 1.0e-6
        epxflag = 1;
    else
        epx = epx + 1;
    end
end
x_left = (10.0^-epx)*fix((10.0^epx)*xl*cos(yt));      % Round x_left towards 0.
x_incr = (x_right-x_left)/2.0;
x_tick = [x_left:x_incr:x_right];
if (xr < 0.001)
    epxt = abs( floor(log10(xr)) );                             % Round multiplicative exponent towards -INFINITY.
    x_tick_label = ([x_left:x_incr:x_right]*10.0^epxt)';
elseif abs( log10(xr) - floor(log10(xr)) ) < 1.0e-6
    epxt = abs( floor(log10(xr)) ) + 1;         % Round multiplicative exponent towards -INFINITY. Add 1 if exactly .001, .0001, etc.
    x_tick_label = ([x_left:x_incr:x_right]*10.0^epxt)';
else
    x_tick_label = [x_left:x_incr:x_right]';
end
y_top = xr*sin(yt);      % Round y_top towards INFINITY.
epyflag = 0;
epy = 0;
while epyflag == 0
    if fix(y_top*10.0^epy) >= 1
        epyflag = 1;
    else
        epy = epy + 1;
    end
end
y_top = (10.0^-(epy+1))*floor((10.0^(epy+1))*y_top)     % To ensure representation of y_top (& all y-axis ticks) to 2 significant
digits.
y_bottom = -y_top;
y_incr = (y_top-y_bottom)/2.0;
y_tick = [y_bottom:y_incr:y_top];
if (epy > 3)                      % Use y_top instead of yt here since y_top = r*SIN(yt) ~ r*yt could become small for small yt!
    y_tick_label = ([y_bottom:y_incr:y_top]*10.0^epy)';
else
    y_tick_label = [y_bottom:y_incr:y_top]';
end
x_label_xloc = x_left + (x_right-x_left)/2.0;
x_label_yloc = y_bottom - 0.25*y_incr;
y_label_xloc = x_right + 0.25*x_incr;
y_label_yloc = y_bottom + (y_top-y_bottom)/2.0;

% Compute the limits & tick marks along the z-axis for POLAR MESH-CONTOUR plot representation. It is being assumed that the
temperature decays with time.
% So, for uniformity in representation of plots and colormap at different times, BOTH the axes AND the colormap are scaled with
respect to the earliest time-level,
% corresponding to "t(1)". Also, adjust this z_min value so that the lower z-axis limit is "well" below the minimum value. This is the
z-level (or plane)
% at which contours will be drawn in the 3D plot. ALways set z_max to one z_increment above the max z value. Use "FIX(X)" instead of
"ROUND(X)" to round to the
% lower integer (i.e., round towards 0) always, in determining z_incr to be used for the plots.
if log_flag == 'y'                    % Use log scale when z data varies by orders of magnitude.
    if max_z > 0.0
        z_max = log10(max_z);
    else
        z_max = logzero;                    % Finite Approximation for Log(0) for plotting.
    end
    if min_z > 0.0
        z_min = log10(min_z);
    else
        z_min = logzero;                    % Finite Approximation for Log(0) for plotting.
    end
else
    z_max = 10.0*round(max_z/10.0);                              % Round to nearest 10 K.
    z_min = 10.0*round(min_z/10.0);                              % Round to nearest 10 K.
end
if log_flag == 'y'
    incr = (z_max-z_min)/10.0;
    z_incr = 0.01*fix(incr*100)
    if z_incr >= (log10(max_z))
        z_incr = z_incr/5.0;
    end
else  % If z increment is larger thant he original maximum value, reduce it by a factor of 5.
    z_incr = (z_max-z_min)/10.0;
    if z_incr > 1.0
        order = fix(log10(z_incr));        % Round exponent towards 0 (ZERO) (to get the order of magnitude of "yincr") if increment
is > 1
    else
        order = floor(log10(z_incr));      % Round exponent towards -INFINITY (to get the order of magnitude of "yincr") if increment
is < 1
    end
    z_incr = (10.0^order)*round(z_incr/10.0^order);
    %z_incr = 20.0*fix((z_max-z_min)/100.0);      % Round towards nearest 10 K.
    %if z_incr >= (max_z - min_z)
    %    z_incr = z_incr/5.0;
    %end
end
```

```
if log_flag == 'y'
    z_max = fix(z_max) + z_incr;                     % Use for log scale.
    if z_max < log10(max_z)
        while z_max < log10(max_z)
            z_max = z_max + z_incr;
        end
    end
else                                             % Make sure that the max tick value is one increment above the max z data.
    if z_max < max_z                            % Can happen when z_incr is very small.
        while z_max < max_z
            z_max = z_max + z_incr;
        end
    end
end
if log_flag == 'y'
    z_steps = (fix((z_max - z_min)/z_incr)) + 1;
else
    z_steps = (fix((z_max - z_min)/z_incr)) + 1;
    %z_steps = (fix((max_z - min_z)/z_incr)) + 1;
end
base_steps = z_steps;                            % Use for log scale.
%if mod(z_steps,2) == 0                          % Use when using absolute temperatures.
%   base_steps = z_steps/2.0;
%else
%   base_steps = (z_steps+1)/2.0;
%end
% Save z_min at this stage for use in caxis command below, before changing it to adjust the floor level of the contour map.
z_min_colormap = z_min
z_min = z_min - base_steps*z_incr;
z_tick = [z_min:z_incr:z_max];
%z_tick = [z_min:2*z_incr:z_max];   % Generate z ticks at twice the z_increment for plotting & determining range, IF z_incr is small.
% Use log scale when z data varies by orders of magnitude. In any case, the tick labels can still retain their original values.
if log_flag == 'y'
    %z_tick_label_char = num2str(10.^z_tick);
    z_tick_label_char = num2str(z_tick);
else
    z_tick_label_char = num2str(z_tick);
end
blankpos = findstr(' ',z_tick_label_char);
blanklet = isspace(z_tick_label_char);
% "blanklet" above is an array of the same size as "z_tick_label_char", with 1(ONE)s at blank positions, and 0s at other places.
% "findstr" does not output information about the 1st and last character strings in the z_tick_label_char array, since they do not
% start with a blank. Therefore, separate loops must be used to identify, and later compute, these end values.
% The following loops mark the length of each tick mark label in z_tick_label_char:
k = 1;              % k is the Tick Mark Label Index - the final value of k is the total # of tick mark labels.
if blankpos(1) ~= 1                             % First character string.
    strbegin(1) = 1;
    strend(1) = blankpos(1) - 1;
    len(1) = strend(1) - strbegin(1) + 1;
    k = k + 1;
end
i = 1;
while (i+1) <= size(blankpos,2)
    if (blankpos(i+1) - blankpos(i)) > 1
        strbegin(k) = blankpos(i) + 1;
        strend(k) = blankpos(i+1) - 1;
        len(k) = strend(k) - strbegin(k) + 1;
        k = k + 1;
    end
    i = i + 1                                   % Increment inside array "BLANKPOS".
end
if k == size(z_tick,2)                          % Last character string.
    strbegin(k) = blankpos(size(blankpos,2)) + 1;
    strend(k) = size(z_tick_label_char,2);
    len(k) = strend(k) - strbegin(k) + 1;
else
    disp('WARNING: Number of tick labels does not match the number of ticks!')
end
z_tick_label = cell(1,size(z_tick,2));      % Create and INITIALIZE a CELL ARRAY for storing each of the tick labels (string arrays).
for k = 1:size(z_tick,2)
    disp(['TICK LABEL # ',int2str(k)])
    pos = strbegin(k);
    string = ' ';                           % Initialize string
    for i = 1:len(k)
        string(i:i) = z_tick_label_char(1,pos);
        pos = pos + 1;
    end
    z_tick_label(1,k) = {string};           % ADD each tick mark label string to the Cell array.
end
if log_flag == 'y'
    % Since tick mark increment is 2 times z_incr, the number of tick marks to be erased is only about half as much.
    if mod(base_steps,2) == 0
        base_steps = base_steps/2.0;
    else
        base_steps = (base_steps+1)/2.0;
    end
end
for i = 1:base_steps
    z_tick_label(1,i) = {' '};              % Set the tick marks outside the z data range to blanks, in the Cell Array.
end
z_tick_label = z_tick_label';               % Convert the tick label vector into a column vector for use in Meshc plots.
x_label_zloc = z_min - z_incr;
y_label_zloc = z_min - z_incr;
```

```
if log_flag == 'y'
    z_label_zloc = z_min + (z_max - z_min)/2.0;
else
    z_label_zloc = min_z + (max_z - min_z)/2.0;
end
z_label_xloc = x_left - 0.5*x_incr;
z_label_yloc = y_bottom - 0.5*y_incr;

% FINALLY CREATE MESH-CONTOUR AS WELL AS POLAR COLOR PLOTS FOR EACH FILE.
% First, open an AVI file to store the movie generated. Then create the mesh plots at each time step (one frame), and store each frame
% in the AVI file.
outfile = [inpfilebeg,inpfileend];
aviobj = avifile([outfile,'_movie.avi'],'fps',4,'compression','None');

for nf = 1:max_files
    % Open the data input files and obtain plot temperatures.
    Z1 = dlmread([filestart,int2str(nf)], ' ');

    % Extend the temperatures symmetrically across to the other semi-circle.
    Z = zeros(grid_rows,grid_columns);
    for j = 1:grid_rows
        for i = 1:grid_columns
            if j < grid_rows_by_2
                if rem(grid_rows,2) == 0
                    disp('******************** WARNING: Variable GRID_ROWS is even! **************************')
                    pause
                    %if (grid_rows-j) > 0
                    %Z(j,i) = Z1( (grid_rows_by_2-j), i );
                    %else
                    % SATISFY PERIODICITY: Set the z data at the top of the y-axis range (if 2*PI) the same as that at the bottom (0)
                    %    Z(j,i) = Z1(1,i);
                    %end
                else
                    Z(j,i) = Z1( (grid_rows_by_2-j+1), i );
                end
            else
                Z(j,i) = Z1(j-grid_rows_by_2+1,i);
            end
        end
    end
    if log_flag == 'y'                          % Use log scale when z data varies by orders of magnitude.
        for j = 1:grid_rows
            for i = 1:grid_columns
                if Z(j,i) > 0.0
                    Z(j,i) = log10(Z(j,i));
                else
                    Z(j,i) = logzero;           % Approximating Log(0), for plotting purposes.
                end
            end
        end
    end

    subplot(2,1,1)              % ROW 1
    h1 = meshc_zcontour(X',Y',Z);
    load thesis_colormap -mat
    colormap(temperature_colormap)
    camproj perspective
    view(24.0,12.0)
    daspect([1 1 zaspect])        % For thesis problem, when using absolute temperatures.
    if log_flag == 'y'
        %caxis([z_min_colormap z_max])
        caxis([0 4])
    else                                        % Set colormap scale for the first time level, and HOLD IT ON for next plot.
    %    caxis([min_z max_z])
    %    caxis([min_z 2050])                    % max_z based on T_melt of Quartz, ~2050 K.
        caxis([min_z 1500])                    % max_z based on T_melt of Feldspar, ~1500 K.
    end
    % Set background color and axes properties for current figure.
    set(gcf, 'Color'                , 'white'   ,...
             'DefaultAxesColor'      , 'white'   ,...
             'DefaultAxesFontName'   , 'times'   ,...
             'DefaultAxesFontSize'   , 8          )
    t_text = {['Fig',int2str(nf),'. POLAR Color Mesh-Contour Plots for: ',inpfileend,' at time = ',num2str(t(nf)),' s.'],...
              ['(k=',num2str(ht),'*hx=',num2str(hx),'*hy=',num2str(hy),')']};
    h_title = text('String',t_text,'Color', 'black', 'FontAngle', 'normal', 'FontName', 'times', 'FontWeight', 'bold', 'FontSize', 9);
    x_label = text(x_label_xloc,x_label_yloc,x_label_zloc,'x','Color','blue','FontAngle','italic','FontWeight','bold','FontSize', 8);
    y_label = text(y_label_xloc,y_label_yloc,y_label_zloc,'y','Color','blue','FontAngle','italic','FontWeight','bold','FontSize',8);
    z_label = text(z_label_xloc,z_label_yloc,z_label_zloc,'T','Color','blue','FontAngle','italic','FontWeight','bold','FontSize', 8);
    set(gca,'Title'     , h_title                           ,...
            'FontName'  , 'times'                           ,...
            'FontSize'  , 8                                 ,...
            'XLim'      , [x_left x_right]                  ,...
            'XTick'     , x_tick                            ,...
            'XTickLabel', x_tick_label                      ,...
            'YLim'      , [y_bottom y_top]                  ,...
            'YTick'     , y_tick                            ,...
            'YTickLabel', y_tick_label                      ,...
            'ZLim'      , [z_min z_max]                     ,...
            'ZTick'     , z_tick                            ,...
            'ZTickLabel', z_tick_label                          )
```

```
    subplot(2,1,2)              % ROW 2
    h2 = pcolor(X',Y',Z);
    colormap(temperature_colormap)
    daspect([1 1 1])        % For thesis problem, when using absolute temperatures.
    % "pcolor" plots data in plan view (Elevation = 90 Deg). Rotate the plot by 90 Deg. along the Azimuth, for proper orientation:
    % 0 Deg. at bottom & 180 Deg. at top.
    view(90.0, 90.0)             % Azimuth, Elevation.
    shading faceted
    set(h2,'LineStyle','none')
    if log_flag == 'y'
        %caxis([z_min_colormap z_max])
        caxis([0 4])
    else                                        % Set colormap scale for the first time level, and HOLD IT ON for next plot.
    %    caxis([min_z max_z])
    %    caxis([min_z 2050])                    % max_z based on T_melt of Quartz, ~2050 K.
        caxis([min_z 1500])                    % max_z based on T_melt of Feldspar, ~1500 K.
    end
  %  set(gca,'FontName'  , 'times'                          ,...
  %          'FontSize'  , 8                                ,...
  %          'XLim'      , [x_left x_right]                 ,...
  %          'XTick'     , x_tick                           ,...
  %          'XTickLabel', x_tick_label                     ,...
  %          'YLim'      , [y_bottom y_top]                 ,...
  %          'YTick'     , y_tick                           ,...
  %          'YTickLabel', y_tick_label                         )
    set(gca,'FontName'  , 'times'                          ,...
            'FontSize'  , 8                                ,...
            'XLim'      , [x_left x_right]                 ,...
            'YLim'      , [y_bottom y_top])
    colorbar('horiz')

    % Save current figure and export it to uncompressed tiff format (at specified dpi).
    %saveas(gcf, [outfile,'_image_',int2str(nf),'.fig'])
    dpi = 200;
    print(['-r',int2str(dpi)], '-dtiff', [outfile,'_image_',int2str(nf),'.tif'])
    print(['-r',int2str(dpi)], '-dpsc', [outfile,'_image_',int2str(nf),'.ps'])            % THESE TWO POSTSCRIPT FILES ARE FOR USING
THESE PLOTS IN POSTERS.
    %print(['-r',int2str(dpi)], '-dpsc2', [outfile,'_image_',int2str(nf),'_L2.ps'])

    % Create and save as a movie frame for the current time step.
    F(nf) = getframe(gcf);
    aviobj = addframe(aviobj,F(nf));
    disp (['Time Step = ',int2str(nf),': Plot and Movie Output SAVED.'])

    %pause
end
aviobj = close(aviobj);
% Save the movie frame to a "MAT" file, using the save command. The command load <filename> X,Y,Z can be used to load the above "MAT"
file later. This allows for the
% movie to be stored in a MATLAB readable format.
movfile = [outfile,'_movie.mat'];
eval( ['save ',movfile,' F'], ['Error saving MATLAB movie file!'] )
% PLAY MOVIE "num" TIMES. The loading procedure shown is redundant here. But it is being used to test the frame saving and retrieval
process. Just using the movie
% command will do the job, as in the next two lines.
num = input('Number of times you want to play the movie: ');
fps = input('Input speed in frames per second, fps: ');
load(movfile,'-mat')
movie(gcf,F,num,fps, [0 0 0 0])

% ------------------------------------------------ END ---------------------------------------------------------------------
```

REFERENCES

Archard, J. F., 1953. Contact and rubbing of flat surfaces. Journal of Applied Physics, 24, 981-988.

Archard, J. F., 1957. Elastic deformation and the laws of friction. Proceedings of the Royal Society of London, A243, 190-205.

Archard, J. F., 1958-59. The temperature of rubbing surfaces. Wear, 2, 438-455.

Asmar, N. H., 2000. Partial Differential Equations and Boundary Value Problems. Prentice Hall, Upper Saddle River, NJ.

Barber, J. R., 1967. The distribution of heat between sliding surfaces. Journal of Mechanical Engineering Science, 9, 351-354.

Barber, J. R., 1970. The conduction of heat from sliding solids. International Journal of Heat and Mass Transfer, 13, 857-869.

Bellman, R., 1948. On the existence and boundedness of solutions of nonlinear partial differential equations of parabolic type. Transactions of the American Mathematical Society, 64, 21-44.

Berry, M. V., Lewis, Z. V., 1980. On the Weierstrass-Mandelbrot fractal function. Proceedings of the Royal Society of London, A370, 459-484.

Berry, G. A., Barber, J. R., 1984. The division of frictional heat – a guide to the nature of sliding contact. Journal of Tribology, 106, 405- 415.

Bowden, F. P., Tabor, D., 1950. The Friction and Lubrication of Solids – Part I. Oxford University Press, Oxford.

Bowden, F. P., and P. H. Thomas, 1954. The surface temperature of sliding solids, Proceedings of the Royal Society of London, Series A, 29-40.

Bowden, F. P., Tabor, D., 1964. The Friction and Lubrication of Solids – Part II. Oxford University Press, Oxford.

Broadbridge, P., Lavrentiev, M. M., and Williams, G. H., 1999. Nonlinear heat conduction through an externally heated radiant plasma: Background analysis for a numerical study. Journal of Mathematical Analysis and Applications, 238, 353-368.

Byerlee, J., 1978. Friction of Rocks. Pure and Applied Geophysics,116, 617-626.

Cameron, A., Gordon, A. N., Symm, G. T., 1965. Contact temperatures in rolling/sliding surfaces. Proceedings of the Royal Society of London, A286, 45-61.

Cardwell, R. K., Chinn, D. S., Moore, G. F., Turcotte, D. L., 1978. Frictional heating on a fault zone with finite thickness. Geophysical Journal of the Royal Astronomical Society, 52, 525-530.

Carslaw, H. S., Jaeger, J. C., 1959. Conduction of heat in solids. Oxford University Press. New York, NY.

Curewitz, D., Karson, J. A., 1999. Ultracataclasis, sintering, and frictional melting in pseudotachylytes from East Greenland. Journal of Structural Geology, 21, 1693-1713.

Carslaw, H. S., Jaeger, J. C., 1959. Conduction of Heat in Solids. Oxford University Press. New York, NY.

Dendy, J. E., 1977. Alternating direction methods for nonlinear time-dependent problems. SIAM Journal on Numerical Analysis, 14, 313-326.

Douglas, J. and Gunn, J. E., 1964. A general formulation of alternating direction methods, Part I: Parabolic and hyperbolic problems, Numerische Mathematik, 6, 428-453.

Douglas, J. and Rachford, H. H., 1956. On the numerical solution of heat conduction problems in two and three space dimensions, Transactions of the American Mathematical Society, 82, 421-439.

Erismann, T. H., 1979. Mechanisms of large landslides. Rock Mechanics, 12, 15-46.

Ettles, C. M. McC., 1986. The thermal control of friction at high slidingspeeds, J. Tribology, Trans. ASME, 108, 98-104.

Grocott, J., 1981. Fracture geometry of pseudotachylyte generation zones: a study of shear fractures formed during seismic events. Journal of Structural Geology, 3, 169-178.

Hall, H. S., Knight, S. R., 1967. Higher Algebra. Macmillan and Company. London.

Harris, T. A. 1966. Rolling Bearing Analysis. John Wiley & Sons. New York, NY

HP Fortran 90 Users Guide, 1998. HP 9000 computers (1^{st} Ed.). B3909-90002, Hewlett Packard, San Jose, CA.

Jacques, L. M., and Rice, J. R., 2002. Partial melting and liquefaction of granular fault gouge during earhthquake slip. Eos Transactions, American Geophysical Union, 83(47), Fall Meeting Supplement, Abstract T11F-07.

Jaeger, J. C., and Cook, N. G. W., 1979. Fundamentals of Rock Mechanics. 3 ed. , Chapman and Hall, New York, NY.

Kanamori, H., 1994. Mechanics of earthquakes. Annual Review of Earth and Planetary Science Letters, 22, 207-237.

Kanamori, H., Anderson, D. L., Heaton, T. H., 1998. Frictional melting during the rupture of the 1994 Bolivian earthquake. Science, 279, 839-842.

Killick, A. M., Roering, C., 1998. An estimate of the physical conditions of pseudotachylyte formation in the West Rand Goldfield, Witwatersrand Basin, South Africa. Tectonophysics, 284, 247-259.

Kuo, C. H., Keer, L. M., 1992. Contact stress analysis of a layered transversely isotropic half-space. Journal of Tribology, 114, 253-262.

Lim, S. C., and M. F. Ashby, 1987. Wear mechanism maps, ActaMetallurgica, 35, 1-24.

Lim, S. C., M. F. Ashby, and J. F. Brunton, 1989. The effect of slidingconditions on the dry friction of metals, Acta Metallurgica, 37, 767-772.

Lachenbruch, A. H., and Sass, J. H., 1980. Heat flow and energetics of the San Andreas fault zone. Journal of Geophysical Research, B85, 6185-6222.

Logan, J. M., and Teufel, L. W., 1986. The effect of normal stresses on the real area of contact during frictional sliding in rocks. Pure and Applied Geophysics, 124, 471-485.

Lowell, M. R., and Khonsari, M. M, 1999. On the frictional characteristics of ball bearings coated with solid lubricants. Journal of Tribology, 121, 761-767.

Lowell, M. R., Khonsari, M. M, and Marangoni, R. D., 1997. Frictional characteristics of MoS_2 coated ball bearings: A three dimensional finite element analysis. Transactions of the ASME: Journal of Tribology, 119, 754-763.

Lowell, M. R., Khonsari, M. M, and Marangoni, R. D., 1996. Finite element analysis of the frictional forces between a cylindrical bearing element and MoS_2 coated and uncoated surfaces. Wear, 194, 60-70.

Magloughlin, J. F., 1992. Microstructural and chemical changes associated with cataclasis and frictional melting at shallow crustal levels: the cataclasite-pseudotachylyte connection. In Magloughlin, J. F., Spray, J. G., (Ed.). Frictional Melting Processes and Products in Geologic Materials. Tectonophysics, 204, 243-260.

Magloughlin, J. F., Spray, J. G., 1992. Frictional melting processes and products in geological materials: introduction and discussion. In Magloughlin, J. F., Spray, J. G., (Ed.). Frictional Melting Processes and Products in Geologic Materials. Tectonophysics, 204, 197-206.

Mandelbrot, B. B., 1983. The Fractal Geometry of Nature. W.H. Freeman, San Francisco, CA.

Masch, L., Wenk, H. R., Preuss, E., 1985. Electron microscopy study of hyalomylonites – evidence of frictional melting in landslides. Tectonophysics, 115, 131-160.

McDonough, J., M., 2001. Lectures in Basic Computational Numerical Analysis. Class Notes, http://www.engr.uky.edu/~egr537), University of Kentucky. Email: jmmcd@uky.edu

McDonough, J., M., 2002. Lectures on Computational Numerical Analysis of Partial Differential Equations. Class Notes, http://www.engr.uky.edu/~me690), University of Kentucky. Email: jmmcd@uky.edu

McDonough, J. M., and Dong, S., 2001. 2-D to 3-D conversion for Navier-Stokes codes: parallelization issues. In Jenssen, C.B., Andersson, H.I., Ecer, A., Satofuka, N., Kvamsdal, T., Pettersen, B., Periaux, J., and Fox, P. (Eds.). Parallel Computational Fluid Dynamics – Trends and Applications: Proceedings of the Parallel CFD 2000 Conference. Elsevier, New York, 173-180.

McGarr, A., 1980. Some constraints on levels of shear stress in the crust from observations and theory. Journal of Geophysical Research, B85, 6231-6238.

McKinzie, D., Brune, J. N., 1972. Melting on fault planes during large earthquakes. Geophysical Journal of the Royal Astronomical Society, 29, 65-78.

Molinari, A., Estrin, Y. and Mercier, S., 1999. Dependence of the coefficient offriction on sliding conditions in the high velocity range, Journal of Tribology, Transactions of the American Society of Mechanical Engineers, 121, 35-41.

Moon, P. H., and Spencer, D. E., 1988. Field Theory Handbook, Including Coordinate Systems, Differential Equations, and Their Solutions. Springer-Verlag, New York, 1-48.

Nadeau, R. M. and Johnson, L. R., 1998. Seismological studies at Parkfield VI: Moment release rates and estimates of source parameters for small repeating earthquakes, Bulletin of the Seismological Society of America, 88, 790-814.

O'Hara, K., 1992. Major- and trace-element constraints on the petrogenesis of a fault-related pseudotachylyte, western Blue Ridge province, North Carolina. In Magloughlin, J. F., Spray, J. G., (Ed.). Frictional Melting Processes and Products in Geologic Materials. Tectonophysics, 204, 261-278.

O'Hara, K. D., 2001. A pseudotachylyte geothermometer. Journal of Structural Geology, 28, 1345-1357.

Oxburgh, E., R., and Turcotte, D., L., 1974. Thermal Gradients and Regional Metamorphism in Overthrust Terrains with Special Reference to the Eastern Alps, Schweizerische Mineralogische und Petrogphische Mitteilungen, v 54, 641-662.

Power, W., Tullis, T. E., Weeks, J. D., 1988. Roughness and wear during brittle faulting. Journal of Geophysical Research, B12, 15268-15278.

Power, W., Tullis, T. E., 1995. Review of the fractal character of natural fault surfaces with implications for friction and the evolution of fault zones. In Barton, C. C., and LaPointe, P. R. (Eds). Fractals in the Earth Sciences. Plenum Press. New York, NY, 89-105.

Queener, C. A., Smith, T. C., Mitchell, W. L., 1965. Transient wear of machine parts. Wear, 8, 391-400.

Rabinowicz, E., 1995. Friction and Wear of Materials. John Wiley & Sons. New York, NY.

Ranalli, G., 1995. Rheology of the Earth. Chapman & Hall, New York, NY.

Ray, S. K., 1998. Transformation of cataclastically deformed rocks to pseudotachylyte by pervasion of frictional melt: inferences from clast-size analysis. Tectonophysics, 301, 283-304.

Rice, J. R., 1999. Flash heating at asperity contacts and rate-dependent friction. Eos Transactions, American Geophysical Union, 80(47), Fall Meeting Supplement, Abstract S21D-05.

Sammis, C. G., Nadeau, R. M., and Johnson, L. R., 1999. How Strong is an Asperity? Journal of Geophysical Research, 104-B, 10609-10619.

Scholz, C. H., 1990. Mechanics of Earthquakes and Faulting. Cambridge University Press. New York, NY.

Scott, J. S., Drever, H. I., 1953. Frictional fusion along a Himalayan thrust. Proceedings of the Royal Society of Edinburgh, 65, 121-142.

Shimamoto, T., Nagahama, H., 1992. An argument against the crush origin of pseudotachylytes based on the analysis of clast-size distribution. Journal of Structural Geology, 14, 999-1006.

Sibson, R. H., 1975. Generation of pseudotachylyte by ancient seismic faulting. Geophysical Journal of the Royal Astronomical Society, 43, 775-794.

Singh, K. P., Paul, B., 1974. Numerical solution of non-Hertzian contact problems. Applied Mechanics, 41, 484-490.

Spence, E. W., Kaminski, E. A., 1996. Thermal evaluation of a dry non-rotating thin section contact bearing. Transactions of the ASME: Journal of Manufacturing Science and Engineering, 118, 610-614.

Spray, J. G., 1992. A physical basis for the frictional melting of some rock-forming minerals. In Magloughlin, J. F., Spray, J. G., (Ed.). Frictional Melting Processes and Products in Geologic Materials. Tectonophysics, 204, 205-221.

Spray, J. G., 1995. Pseudotachylyte controversy: fact or friction? Geology, 23, 1119-1122.

Spray, J. G., 1997. Superfaults. Geology, 25, 579-582.

Strauss, W. A., 1992. Partial Differential Equations: An Introduction. John Wiley and Sons, New York.

Swanson, M. T., 1992. Fault structure, wear mechanisms, and rupture processes in pseudotachylyte generation. In Magloughlin, J. F., Spray, J. G., (Ed.). Frictional Melting Processes and Products in Geologic Materials. Tectonophysics, 204, 223-242.

Timoshenko, S. P., Goodier, J. N., 1970. Theory of Elasticity. McGraw Hill. New York, NY.

Turcotte, D., L., and Schubert, G., 2001. Geodynamics. Cambridge, New York, NY.

Turcotte, D. L., Tag, P. H., Cooper, R. F., 1980. A steady state model for the distribution of stress and temperature on the San Andreas Fault. Journal of Geophysical Research, B85, 6224-6230.

Touloukian, Y.S., Judd, W.R., Roy, R.F., 1981. Physical Properties of Rocks and Minerals. In Touloukian, Y.S., and Ho, C.Y. (Eds). McGraw-Hill/CINDAS Data Series on Material Properties, Volume II-2. McGraw Hill, New York.

Wang, W., Scholz, C. C., 1994. Wear processes during frictional sliding of rock: a theoretical and experimental study. Journal of Geophysical Research, B99, 6789-6799.

Webster, M. N., Sayles, R. S., 1986. A numerical model for the elastic frictionless contact of real rough surfaces. Transactions of the ASME: Journal of Tribology, 108, 314-320.

Wenk, H. R., Weiss, L. E., 1982. Al-rich calcic pyroxene in pseudotachylyte: an indicator of high pressure and high temperature? Tectonophysics, 84, 329-341.

Yovanovich, M. M., 1966. Thermal contact resistance across elastically deformed spheres. Journal of Spacecraft, 4, 119-122.

Yovanovich, M. M., 1974. Modeling the effect of air and oil upon the thermal resistance of a sphere-flat contact. In Hering, R. G., (Ed.). Thermophysics and Spacecraft Thermal Control. Progress in Astronautics and Aeronautics, 53, 293-319. MIT Press, Cambridge, MA.

Yu, H., 2001. A local space-time adaptive scheme in solving two-dimensional parabolic problems based on domain decomposition methods. SIAM Journal on Scientific Computing, 23, 304-322.

VITA

Date of Birth: 18th of April, 1971.

Place of Birth: Kakinada, Andhra Pradesh, India.

Previous Educational Institutions Attended & Degrees Earned:

- Bachelor of Technology in Mechanical Engineering, Indian Institute of Technology – Bombay, Mumbai, India, 1994.
- Master of Science in Environmental Engineering, University of Cincinnati, Cincinnati, Ohio, 1997.

Previous Professional Positions Held:

Environmental Engineer, Science Applications International Corporation (1997-2001).