2D Basket Options and Finite Elements

Term project for AM562b

BY TYLER HAYES

EPIGRAPH

Men give me credit for genius, but all the genius I have lies in this: When I have a subject in mind I study it profoundly. Day and night it is before me. The result is what some people call the fruits of genius, whereas it is in reality the fruits of study and labour.

Alexander Hamilton

CONTENTS

LIST OF FIGURES

Chapter 1

A BRIEF INTRODUCTION TO OPTIONS

I'll make him an offer he can't refuse.

Don Vito Corleone

1.1 A financial instrument: The option

There exist many different types of financial instruments traded every day on exchanges around the world. Indeed, not just equity stocks are bought and sold. Products such as bonds, mutual funds, currency, and commodites, such as grain, oil, and other raw goods are among the more traditional investment products most of us are familiar with. For those of us who remember Eddie Murphy's, Trading Places, we were introduced to a different type of financial instrument known as a future.

The future is a standized instrument in which a buyer and seller agree to sell a commodity at a specified price, to be delivered to the buyer on a specific date. Going back to our example of Trading Places, the futures were for frozen concentrated orange juice (FCOJ), in the hopes that a poor yield of fresh oranges would result in higher FCOJ prices. In such a case, the purchasers were speculating that the market would swing a particular way in their favour. The downfall of such speculation (and for Duke & Duke!) is that the seller/buyer is obligated to sell/buy the commodity at the specified price, even if it results in a net loss on the transaction. It is at this stage that we are introduced to the option.

Similar to the future, an option is a financial instrument where the holder has the right to purchase or sell the underlying product (stock, bond, commodity, etc.) at the agreed upon stike price on a specified date from the writer of that option. However, unlike the future, the holder does not have the obligation to purchase or sell the underlying product. As such, the buyer pays a premium for this right or, you guessed it, option. This simplified description describes what is known as a *Vanilla* or *European* option.

1.2 Calls and puts

In general, we can classify options into one of two categories: a call, or a put. By definition, a CALL OPTION is the right to buy a particular asset for an agreed amount at a specified time in the future. Conversely, a PUT OPTION is the right to sell a particular asset (Wilmott, 2006) for an agreed amount at a specified time in the future. It is important to note that the holder of the option does not have to purchase/sell the underlying asset if it would not benefit him. He can simply let the option expire and is only out the premium paid at the outset. We describe payoff function at expiry for the call option as,

$$C = \max(S - K, 0) \tag{1.1}$$

and similarly for the put option as,

$$P = \max(K - S, 0) \tag{1.2}$$

where K is the strike price of the option, i.e., the price initially paid for the option, and S is the underlying asset's price. Now it is important to note that (1.1) and (1.2) describe the payoff at the time of expiry, and are the option's intrinsic value for the call and put respectively if the option was exercised today(Wilmott, 2006).

1.3 The Black-Scholes formulation

The preceeding description of an option raises an interesting question: What is the value of the option (i.e., C or P) now, at some time prior to the expiry date? In other words, how does one know what the option is worth? The problem arises from several factors, most notably the random walk nature of the underlying asset, as well as the fact the money today is not worth the same amount at some time in the future. In the early seventies, Black and Scholes (1973) (and expanded upon by Merton (1973)) derived an expression to accurately take into these factors *et cetera*. The resulting expression is often refered to as the BLACK-SCHOLE EQUATION. Without deriving it, we simply state their result below.

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0 \tag{1.3}$$

Where, $V = V(S,t)$, is the value of the option (put or call), r is the paying risk-free interest rate, S is the current value of the underlying asset, and σ is the volatility of S. Wilmott (2006) points out that the first two terms can be interpreted as diffusion in a non-homogeneous medium, the third term can be thought of as a convection term, and the fourth term as a reaction term, such as in radioactive decay. We can solve (1.3) by using the knowledge that at expiry the value of the option is given by (1.1) and (1.2) for the call and put. The expression (1.3) has the following analytic solutions for the call and put respectively, and 1.1 shows a plot of a put with a single underlying asset.

$$C(S,t) = SN(d_1) - Ke^{-r(T-t)}N(d_2) \tag{1.4}$$

$$P(S,t) = -SN(-d_1) + Ke^{-r(T-t)}N(-d_2) \tag{1.5}$$

where $N(\cdot)$ is given by,

$$N(x) = \frac{1}{2\sqrt{2\pi}} \int_{-\infty}^{x} e^{\frac{-1}{2}y^2} dy$$

and d_1 and d_2 are given by,

$$d_1 = \frac{\log(S/K) + (r + \frac{1}{2}\sigma^2)(T-t)}{\sigma\sqrt{T-t}}$$
$$d_2 = \frac{\log(S/K) + (r - \frac{1}{2}\sigma^2)(T-t)}{\sigma\sqrt{T-t}}$$

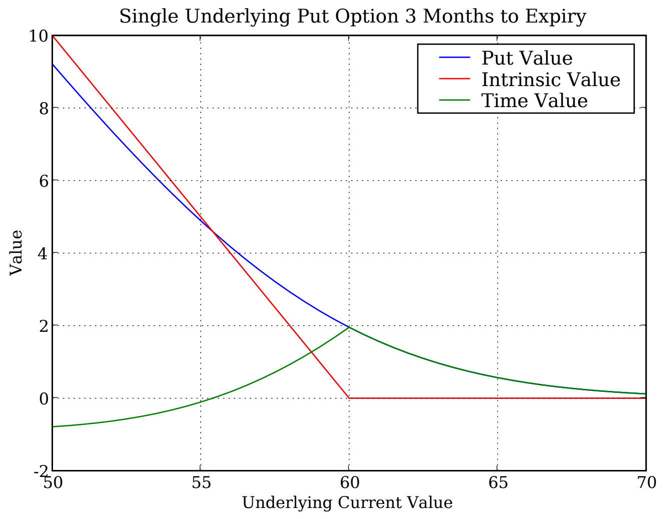

Figure 1.1: Put value for a strike at \$60, 0.25 years to maturity, r =0.06 and a volatility (σ) of 0.2.

1.4 Rainbow Options

We are now ready to extend the discussion to more complex scenarios. An option which consists of more than one underlying assets are called RAINBOW OPTIONS, BASKET OPTIONS, or OPTIONS ON BASKETS (Wilmott, 2006). For our analysis we will consider a option consisting of two underlying assets. This amounts to solving the multi-dimensional version of the Black-Scholes equation and it is given by the following PDE.

$$\frac{\partial V}{\partial t}+\frac{1}{2}\sigma_1^2 S_1^2\frac{\partial^2 V}{\partial S_1^2}+\frac{1}{2}\sigma_2^2 S_2^2\frac{\partial^2 V}{\partial S_2^2}+\rho\sigma_1\sigma_2 S_1 S_2\frac{\partial^2 V}{\partial S_1\partial S_2} +rS_1\frac{\partial V}{\partial S_1}+rS_2\frac{\partial V}{\partial S_2}-rV=0 \quad (1.6)$$

Where the symbols retain their same meaning as before, but now $V=V(S_1,S_2,t)$ and we have a new variable introduced, ρ, which is the correlation coefficient between the two underlying assets, S_1 and S_2. Moreover, the final condition must now be modified.

There are several ways to choose an exit strategy, or final condition, and Achdou and Pironneau (2005) provide the following as possibilities for the value of the put option:

$$P(S_1,S_2,T) = \max(K-(S_1+S_2),0) \tag{1.7a}$$

$$P(S_1,S_2,T) = \max(K-\max((S_1,S_2),0)) \tag{1.7b}$$

For the boundaries of this problem, we can choose a values far "out of the money" and set those to zero as for all times, i.e.,

$$P(S_1,150,t) = 0 \quad ; \forall\, t \tag{1.8a}$$

$$P(150,S_2,t) = 0 \quad ; \forall\, t \tag{1.8b}$$

However, the boundaries "in the money" are now functions of time, i.e.,

$$P(S_1,0,t) = g(S_1,K,t) \tag{1.9a}$$

$$P(0,S_2,t) = g(S_2,K,t) \tag{1.9b}$$

where equations (1.9) are simply the analytic solutions to the Black-Scholes equation with a single underlying asset as given by (1.5).

Chapter 2

THE FINITE ELEMENT FORMULATION

A proof is a proof. What kind of a proof? It's a proof. A proof is a proof. And when you have a good proof, it's because it's proven.

Jean Chrétien

2.1 Finite elements and PDEs

This chapter will serve two basic purposes. The first to describe briefly the concepts behind the finite element method for the numerical solution of partial differential equations, more specifically second-order transient problems of one variable, i.e., parabolic problems. The second purpose is to present the necessary mathematical framework upon which the solution to (1.6) under the conditions given by (1.7), (1.8), and (1.9) is founded for numerical analysis. This chapter is not, however, intended to serve as a complete introduction to the method. As such, some of the expressions will be presented under the assumption that they have been demonstrated in the references provided.

Following Thompson (2005), let us consider the simplified 2-D parabolic problem given by the following expression.

$$\frac{\partial}{\partial x}\left[a_{11}\frac{\partial\Phi}{\partial x}\right]+\frac{\partial}{\partial y}\left[a_{22}\frac{\partial\Phi}{\partial y}\right]=-Q+c_0\frac{\partial\Phi}{\partial t} \tag{2.1}$$

Where we note that a_{11}, a_{22}, Q, and c_0 can be functions of x, y, and t. To reformulate the problem in a tractable way in which we can solve the system numerically we will follow the basic procedure given below (modified from Reddy (1993)).

(i) Discretize the domain.

(ii) Assume that the unknown variable is of a the form:

$$\Phi = \sum_{i=1}^{n} \Phi_i N_i$$

(iii) Find the variational formualtion of the element equations.

(iv) Assemble the elements into the global system of equations.

(v) Apply the boundary conditions.

(vi) Solve the system of equations.

(vii) Repeat from assembly until the final time step is reached.

2.2 Discretization and the element equations

Finite elements requires that the domain of the field being examined be discretized into elements. The field is then approximated within each element using interpolation theory within each of the elements, which can then be summed into the global solution. This discretization is known as MESHING.

For our analysis we are considering a two dimension system and will use triangular elements. Each of the elements is described by three nodes at which the solution solved for. As such, the role meshing plays is crucial, as proper mapping is required to correctly generate the global solution. The mapping is achieved through a connectivity matrix which keeps track of each element and the nodes which describe it. In Figure 2.1, we show the numbering system used. Note that not all of the element values are shown.

We now must consider the variational formulation of the element equations for (2.1). To do so, we will first describe the field variable, Φ, and its time derivative, $\dot{\Phi}$, in terms of an approximation as follows.

$$\Phi(x,y,t) = [N]\{\Phi\} \tag{2.2a}$$

$$\dot{\Phi}(x,y,t) = [N]\{\Phi\} \tag{2.2b}$$

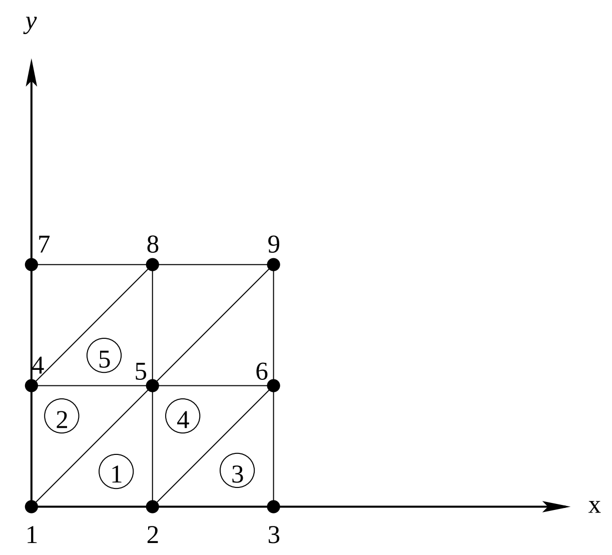

Figure 2.1: The mesh used for triangular elements in this analysis. Circled numbers are the element numbers (not all shown). Others are the node numbers. Modified from Reddy (1993).

Note that we have treated the two variables as separate from each other and that we have now switched to a matrix representation of the expressions. These approximations to the variables Φ and $\dot{\Phi}$ will be substituted into the WEAK FORM of (2.1).

The weak form of the PDE described by (2.1) is found by first multiplying both sides of (2.1) by a small variation, $\delta\Phi$, and integrating over the volume of the domain, then finding the minimum by requiring that the expression be equal to zero, i.e.,

$$\int_V \delta\Phi \left\{ \frac{\partial}{\partial x}\left[a_{11}\frac{\partial\Phi}{\partial x}\right] + \frac{\partial}{\partial y}\left[a_{22}\frac{\partial\Phi}{\partial y}\right] \right\} dV - \int_V \delta\Phi \left\{ -Q + c_0\frac{\partial\Phi}{\partial t} \right\} dV = 0 \quad (2.3)$$

We then use integration by parts for the first term in braces in (2.3) to remove the strong condition of second derivative continuity, and substuting in our approximating functions, (2.2). After which we then use the Galerkin method of applying weight functions equal to the shape functions used to approximate Φ and minimize the integral for an arbitrary variation $\delta\Phi$ and are left with the result below.

$$[K]\{\Phi\}+[C]\{\dot{\Phi}\}=\{Q\} \tag{2.4}$$

While this expression was derived for (2.1), it is in general true for a more complex second-order parabolic PDE such as (1.6). We accomplish by recognizing that the matrix $[K]$ in (2.4) contains only those components of the spatially varying components over the volume of the domain considered. As such, we can write the relevant components for our problem of the matrices in the forms given by Thompson (2005) as follows.

$$[K]=[S_1]+[S_2]+[S_3] \tag{2.5}$$

where,

$$[S_1]=\int\limits_{V_{uv}}\{N'\}[A]\lfloor N'\rfloor\|J\|dV \tag{2.6}$$

$$[S_2]=\int\limits_{V_{uv}}\{N\}\lfloor B\rfloor\lfloor N'\rfloor\|J\|dV \tag{2.7}$$

$$[S_3]=\int\limits_{V_{uv}}\{N\}G\lfloor N\rfloor\|J\|dV \tag{2.8}$$

$$[A]=\begin{bmatrix} a11 & a12 \\ a21 & a22 \end{bmatrix} \tag{2.9}$$

$$\lfloor B\rfloor=\lfloor b1b2\rfloor \tag{2.10}$$

$$[C]=\int\limits_{V_{uv}}\{N\}c_0\lfloor N\rfloor\|J\|dV \tag{2.11}$$

The coefficients G and c_0 can be functions of x, y, and t, and that for the Black-Scholes equation (1.6) the forcing term, $\{Q\}=0$ and thus left out. You also note that we have included the determinant of the Jacobian matrix, $\|J\|$, so as to properly map the intgeration regions which are now in terms of u,v and not, x,y. This is because, we will actually numerically solve the integrals using Gaussian quadrature.

2.3 The shape functions and quadrature

For our problem, we have chosen triangular elements and now we must construct a set of functions for each node. The method of choice is to use non-dimensionalized functions which are zero at two nodes, and unity at its own. In other words, the interpolation varies linearly from zero at the two opposite nodes to unity at its own. Other interpolation functions could be used, such as those which vary in a quadratic fashion, as well as interior nodes in the element, but for our purposes, linear triangular functions and three node elements will suffice. As such, we can set the shape functions $N_i = L_i$, where L_i is our linear shape function for the triangle.

At some point, we still have to numerically solve the integrals presented in the previous section. This is most accurately carried out by employing Gaussian quadrature. However, Gaussian quadrature requires that the integral be evaluated at specified points which in most cases will not coincide with the global coordinates of the nodes. Therefore we transform the global nodes to their Gaussian counterparts (hence the Jacobian in expressions (2.6) to (2.8)) and carry out the integration over the shape functions and their derivatives (i.e., $\{N\}$ and $\lfloor N' \rfloor$ respectively). Thus we follow Reddy (1993) and approximate the integral as below.

$$\int\limits_{G_e} G(L_1, L_2, L_3)\, dL_1\, dL_2 \approx \sum_{i=1}^{3} \frac{1}{2} W_i G(S_i) \tag{2.12}$$

Where W_i and $G(S_i)$ are the associated Gaussian weights and integration points for the triangular element G_e which has been transformed to the Gaussian coordinates (u, v) are given by,

$$W_i = 1/3 \quad \text{for } i = 1,2,3 \tag{2.13a}$$
$$G(S_1) = (1/2, 1/2, 0) \tag{2.13b}$$
$$G(S_2) = (0, 1/2, 1/2) \tag{2.13c}$$
$$G(S_3) = (1/2, 0, 1/2) \tag{2.13d}$$

Once an element has been numerically integrated, it is ready to be assembled into the global matrices given by (2.4). At which point we are in a position to march the system along in time to solve the transient problem.

2.4 Marching in time

In order to solve (2.4), we will employ the Crank-Nicolson finite difference method to march our system in time, as this method is unconditionally stable (Smith, 2003). The method requires us to rewrite (2.4) as follows.

$$\left[\hat{K}\right]_{t+1}\{\Phi\}_{t+1} = [\bar{K}]_t\{\Phi\}_t + \left\{\hat{F}\right\}_{t,t+1} \tag{2.14}$$

where t is the current time step and $t+1$ is the next time step and,

$$\left[\hat{K}\right]_{t+1} = [C] + a_1[K]_t \tag{2.15}$$

$$[\bar{K}]_{t+1} = [C] - a_2[K]_t \tag{2.16}$$

$$\left\{\hat{F}\right\}_{t,t+1} = \Delta t\left[\alpha\left\{\hat{F}\right\}_{t+1} + (1-\alpha)\left\{\hat{F}\right\}_t\right] \tag{2.17}$$

and $a_1 = \alpha\Delta t$, $a_2 = (1-\alpha)\Delta t$, and Δt is the time step for the marching scheme. For a Crank-Nicolson solution, $\alpha = 1/2$.

2.5 Boundary conditions and solving

The last step involved in the process, prior to solving the system of equations, is to apply the boundary conditions to the field variable, Φ. We are fortunate that European Options have relatively simple boundary conditions and surface integrals are not required. To apply the boundary conditions described by (1.8) and (1.9) we will employ the method of "blasting" the diagonal as described by Thompson (2005). Once the boundary conditions have been applied to the system, any preferred method of solving the system can be used. LU decomposition is an obvious choice, other other Gaussian Elimination routines. For our analysis, we actually assemble the equations in a non-symmetric sparse matrix, and we will employ an appropriate Gaussian elimination routine (Thompson, 2005).

Chapter 3

IMPLEMENTATION

You knew the job was dangerous when you took it, Fred.

Super Chicken

3.1 The main algorithm

This section will briefly outline the methdology used to implement the FEM for the basket option described in Chapter 1. The codes listed in the Appendices serve as a complement to this chapter and the comments within the code offer more explanations where required.

The first part of the code concerns mostly initializing the required matrices, the initial conditions (actually interpreted as the final conditions here), and setting up the input parameters. Referring to (1.6), we are able to specify the coefficients for the matrices (2.9), (2.10), (2.11) and the coefficient G. We set them as follows.

$$[A] = \begin{bmatrix} \frac{1}{2}\sigma_1^2 S_1^2 & \frac{1}{2}\rho\sigma_1\sigma_2 S_1 S_2 \\ \frac{1}{2}\rho\sigma_1\sigma_2 S_1 S_2 & \frac{1}{2}\sigma_2^2 S_2^2 \end{bmatrix} \tag{3.1}$$

$$[B] = [rS_1 \; rS_2] \tag{3.2}$$

and $G = -r$, $[C] = 1.0$, and the source term is zero. Next, the mesh routine is called to create the connectivity matrix and generate the global coordinate matrix. From here, several things are done. We find identify the boundary nodes using the function findBC, we generate the initial solution set to start the problem using initialVal. It should be noted that the values used are set by the user at the initial stage of running the program via a GUI interface.

At this stage, the FEM program has all it needs to begin. The time loop is set to stop at the user specified interval, and we enter the loop. Since this is a transient problem, we need to find the boundary conditions for the next time step and this is done at the beginning of the time loop using newBound.

We then enter into the loop for each element and begin the quadrature over the element. The shape functions for each global coordinate are found using the sfntri routine and the quadrature data is loaded by a call to the quad function. The element matrices are calculated via the method described in Chapter 2. However, in the implementation of the time-differencing scheme, we use the method of Reddy (1993) and calculate (2.14) at the elemental level, *prior* to assembly into the global matrix. It should be noted that the global stiffness matrix is stored as a non-symmetric sparse matrix to save memory storage, should the need arise.

Once the time-differencing is complete, and the element values stored in the global stiffness matrix, the loop returns to find the next element's values, and so on. Once each element has been calculated and assembled into the global matrix we then apply the boundary conditons using the blasting technique of Thompson (2005). The solution for the next time step is then found using the equation solver nsymgauss. This entire process is repeated until a specied time has been reached.

3.2 Running BlackScholes2dPut.py

Actually running the program requires very little effort, assuming all of the required software has been installed, in particular, Python, SciPy, NumPy, wxPython, PyVTK, and MayaVi (or alternatively, Matplotlib). To run the program, make sure all of the files are in the same directory, simple open up a terminal window, and enter the command:

```
user@somemachine:$ python BlackScholes2dPut.py
```

At this stage, Python will load various modules and a GUI will prompt the user for the following inputs.

(i) Initial condition choice. Enter 1 for (1.7a) or 2 for (1.7b).

(ii) $S1$ high and $S2$ high. These are values of the two underlying assets which are considered well "out of the money" and will be the upper bounds of the domain.

(iii) $S1$ and $S2$ volatilities. Self explanatory.

(iv) The interest rate to be used and the correlation between $S1$ and $S2$.

(v) The strike price (K) of the option, and the expiry date. The expiry date is the full amount of time to be considered until expiry.

(vi) Delta T, NX and NY. Delta T is the time step to be used and should divide nicely into the expiry date. NX and NY are the number of divisions along the $S1$ and $S2$ axes. As such, they control how many nodes are used and how fine grained the mesh will be. Once again, one should use numbers which divide nicely into $S1$ high and $S2$ high, respectively.

Once all of the data has been entered, press the "Run" button, and the FEM program will work away. Depending on the size of mesh, time to expiry, and Δt, used, the program may take several minutes to run. Once completed, the final solution will be displayed by MayaVi for interactive graphical analysis of the data.

3.3 Results

To test the method, I used the inputs given by Achdou and Pironneau (2005) and looked at both exit strategies given by (1.7). The values used were: $S1 = S2 = 150$, $\sigma_1 = \sigma_2 = 0.1414$, $r = 0.1$, $\rho = -0.6$, $K = 100$, time to expiry $= 0.7$, $\Delta t = 0.01$, and $N_x = N_y = 50$ (i.e., 2601 nodes). The results are shown in Figure 3.1, Figure 3.2, Figure 3.3, Figure 3.4, Figure 3.5, and Figure 3.6.

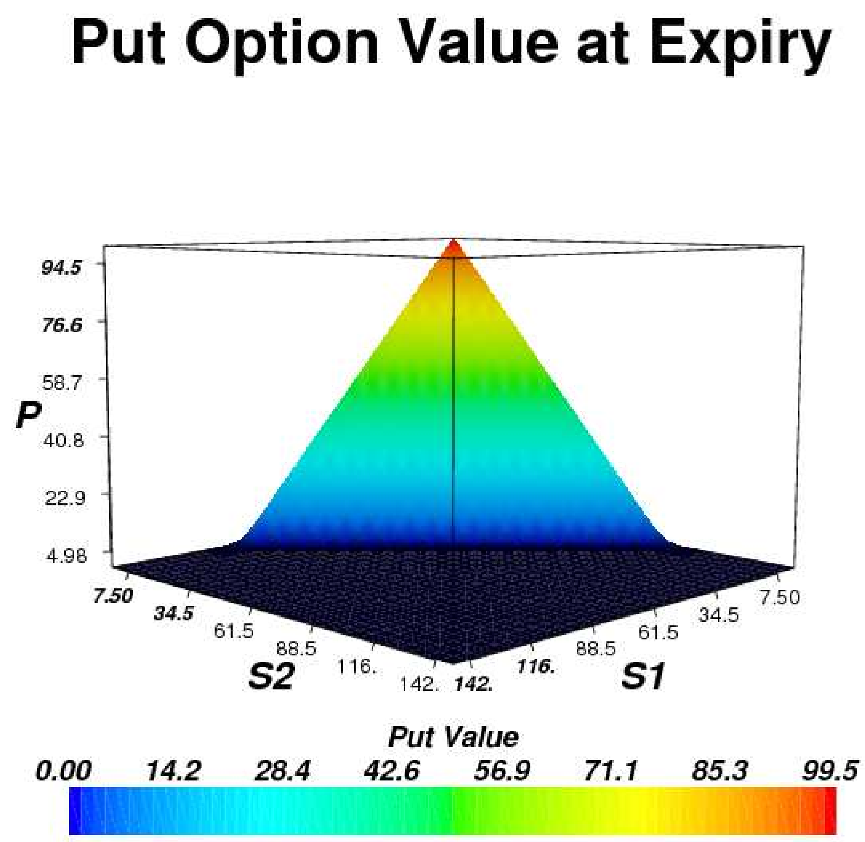

Figure 3.1: Final condition (1.7a).

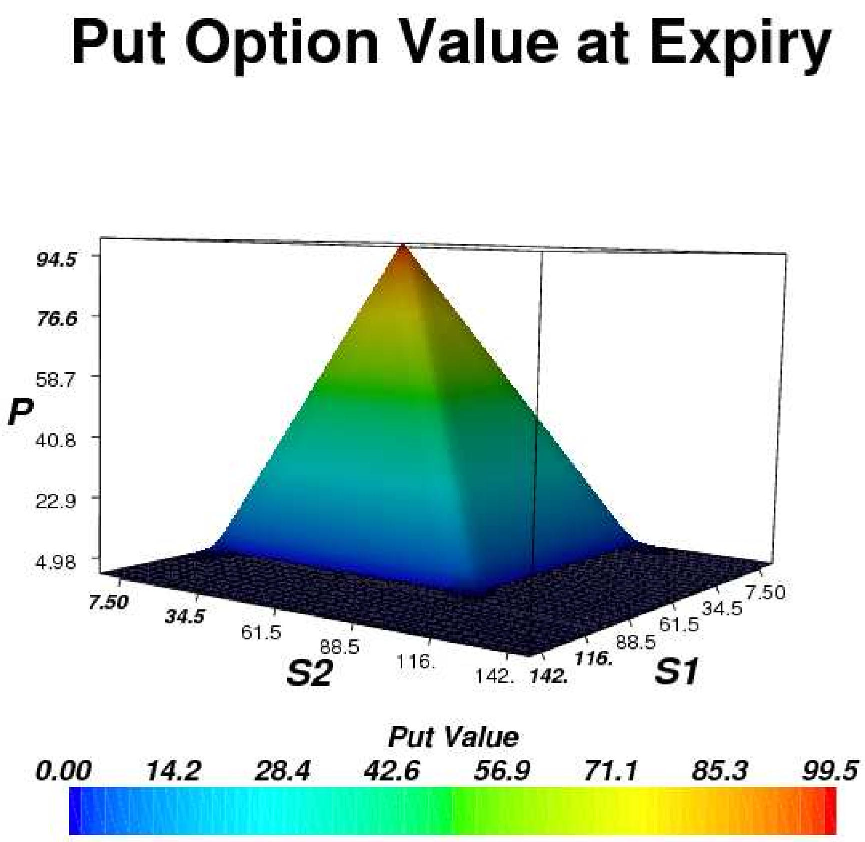

Figure 3.2: Final condition (1.7b).

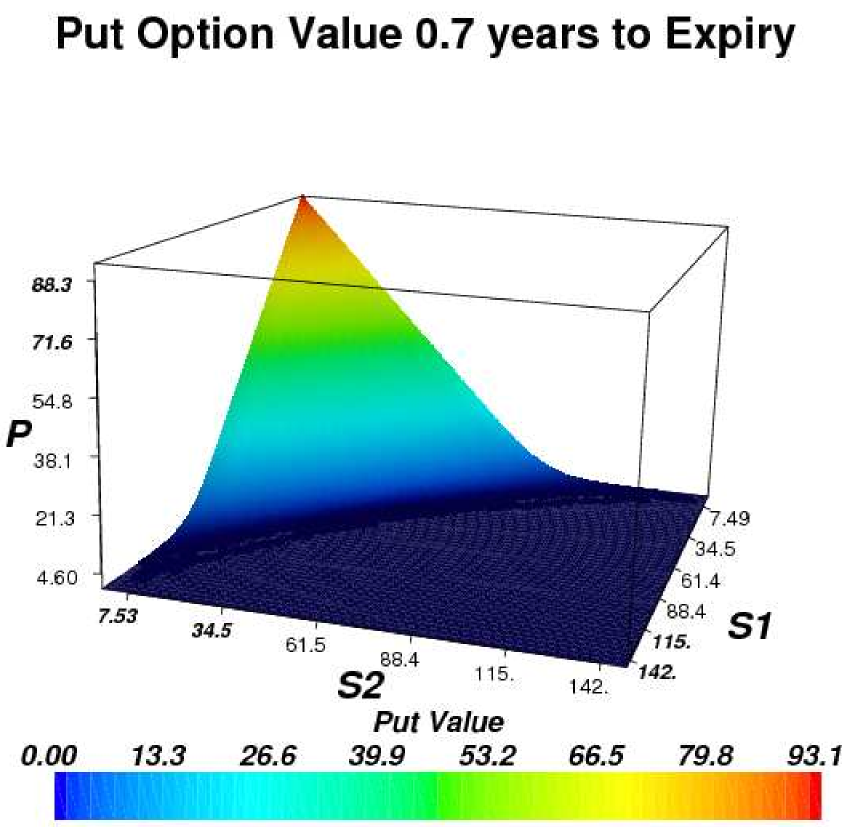

Figure 3.3: Final condition (1.7a) at 0.7 years from maturity.

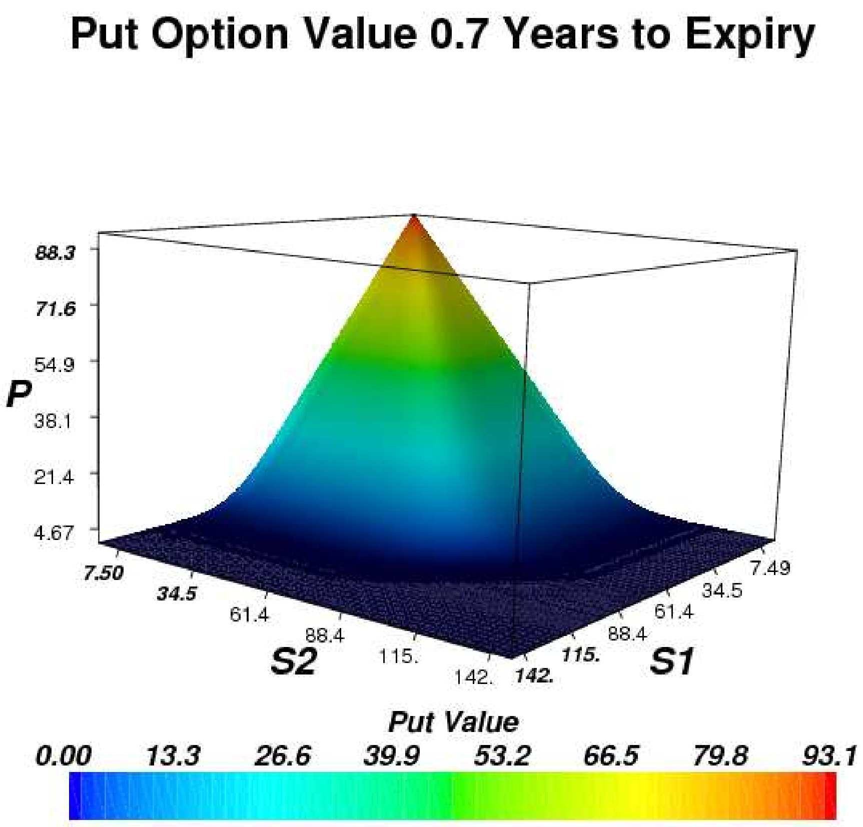

Figure 3.4: Final condition (1.7b) at 0.7 years from maturity.

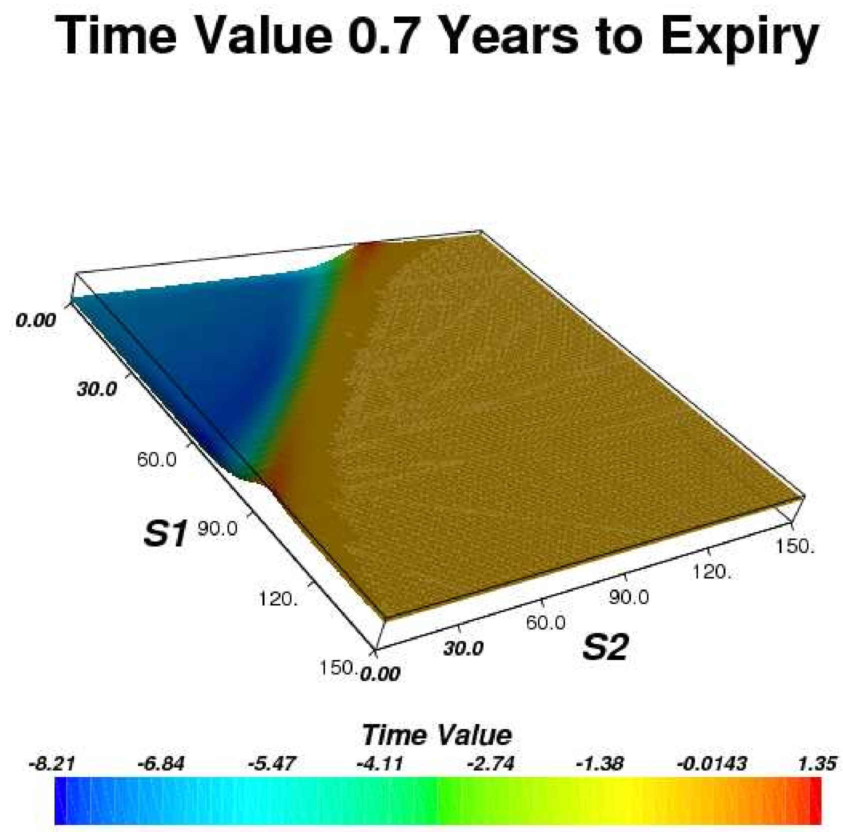

Figure 3.5: Time value of (1.7a) at 0.7 years from maturity.

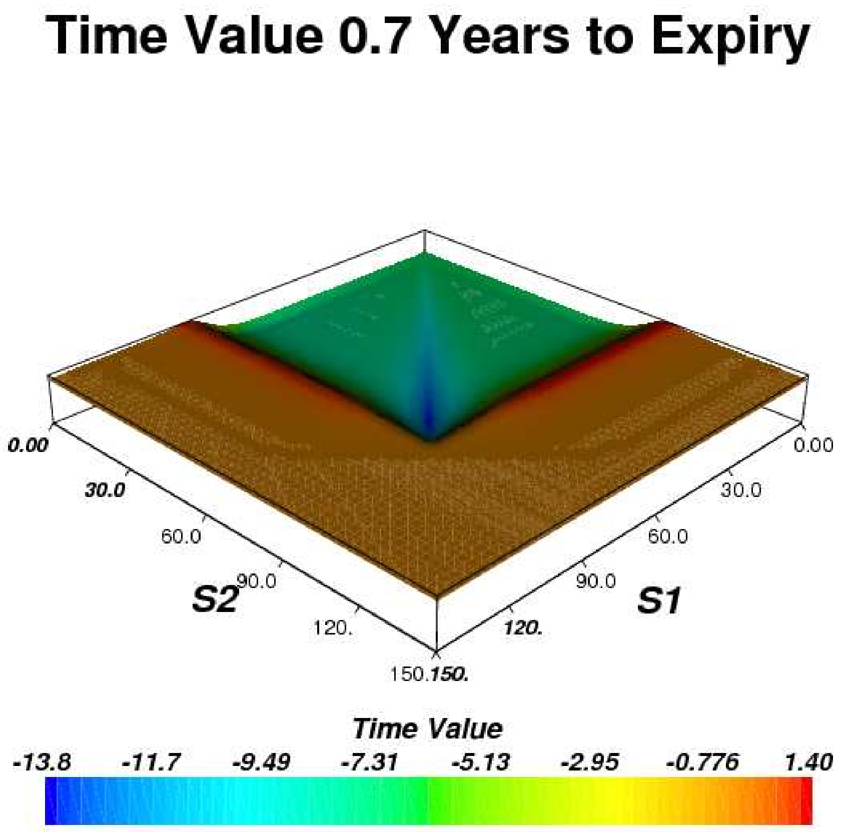

Figure 3.6: Time value of (1.7b) at 0.7 years from maturity.

Chapter 4

EPILOGUE

There is much Obi-wan did not tell you.

Darth Vader

This project serves to demonstrate that the FEM is able to solve higher dimension financial derivative problems, and extensions to more complex options are possible. Other examples of using the FEM to solve financial options can be found in the texts of Achdou and Pironneau (2005) and Topper (2005), both of which consider more exotic options than the problem presented here. As mentioned before, the previous chapters served only as a brief introduction to the subject and should not be taken as complete proofs. I relied heavily on the texts of Wilmott (2006), Reddy (1993), and Thompson (2005) in order to complete this project, and one is encouraged to consult those texts for a more complete explanation of options and the finite element method. As such, this serves merely as a guide to help one understand the logic and method used to implement the code as found in the Appendices. Moreover, the approach taken was to present the most transparent FE routine, and more efficient methods as given by Topper (2005) and Achdou and Pironneau (2005), would normally be employed.

BIBLIOGRAPHY

Achdou, Y. and O. Pironneau (2005). *Computational Methods for Option Pricing*. Frontiers in Applied Mathematics. SIAM.

Black, F. and M. Scholes (1973, May–June). The pricing of options and corporate liabilities. *The Pricing of Options and Corporate Liabilities 81*(3), 637–654.

Merton, R. C. (1973, Spring). Theory of rational option pricing. *The Bell Journal of Economics and Management Science 4*(1), 141–183.

Reddy, J. N. (1993). *An Introduction to the Finite Element Method* (2nd ed.). McGraw-Hill Series in Mechanical Engineering. McGraw-Hill.

Smith, G. D. (2003). *Numerical Solution of Partial Differential Equations: Finite Difference Methods* (Third ed.). Oxford applied mathematics and computing science series. Oxford University Press.

Thompson, E. G. (2005). *Introduction to the Finite Element Method: Theory, Programming, and Applications*. John Wiley & Sons, Inc.

Topper, J. (2005). *Financial Engineering with Finite Elements*. Wiley Finance Series. John Wiley & Sons, Inc.

Wilmott, P. (2006). *Paul Wilmott on Quantitative Finance* (2nd ed.). John Wiley & Sons, Inc. (3-Volume set).

Appendix A

PYTHON CODE LISTINGS

A.1 Main FEM Routine

Please note that some of the formatting has changed to allow for the listings to fit within the textblock. Always adhere to proper Python indenting conventions.

```
#!/usr/bin/env python

# Import Modules
#
    ------------------------------------------------------------


# Scipy and Numpy
from scipy import *

# Mesh generator
import tri

# Non-symmetric Gaussian Solver
import nsymgauss as NSG

# Quadrature data
import quad_data as GQD

# Shape function module
import shpfn as SFN

```

```python
# GUI
import femGUI


# Define python functions
# -------------------------------------------------------
def vanilla(r, K, dt, sigma, S):
    """
    This is a simple Vanilla Put Option calcualtion
    based on the analytic solution for a single
    underlying asset. The solution used is from
    The Mathematics of Financial Derivatives, Wilmott, et al.
    Uses ndtr and exp from scipy and ndtr scipy.special modules.

    r    : risk free rate (float)
    K    : strike price (float)
    dt   : time to expiry (float)
    sigma: volatility of S (float)
    S    : range of underlying values (array[float])

    Usage:

    put_value = vanilla(r, K, dt, sigma, S)

    """

    d1 = zeros(len(S))
    d2 = zeros(len(S))
    n1 = zeros(len(S))
    n2 = zeros(len(S))
    pt = zeros(len(S))
    b  = sigma*sqrt( dt)
    dsct = exp(-1.0*r*dt)
    for i in range(len(S)):
        d1[i] = (log(S[i]/K) + (r + (0.5* sigma**2))*
            dt)/b
        d2[i] = (log(S[i]/K) + (r - (0.5* sigma**2))*
            dt)/b
        n1[i] = special.ndtr(-1.0*d1[i])
        n2[i] = special.ndtr(-1.0*d2[i])
        pt[i] = K*dsct*n2[i] - S[i]*n1[i]

    return pt


# Create the boundary information arrays only once to
# save calculations during time-dependent BCs
def findbc(gnodes,s1max,s2max,nnm):
    """
    This function will return an array of values for
    which global nodes lie on the boundaries.

    bnode:
          0 = interior node
          1 = boundary node

    mind*: the indices of the axes' s1min,s2min nodes
    maxd*: the indices of the axes' s1max,s2max nodes

    gnodes is an array of size (num of nodes) x 2
    gnodes[:,0] = global x values
    gnodes[:,1] = global y values

    bnode,mindx,maxdx,mindy,maxdy,s1y0,s2x0
          = findbc(gnodes,s1max,s2max,nnm)
    """

    # The "max" matrices are not actually called in
        the present
    # program, but may be needed later.
    bnode  = zeros(nnm,dtype=int)
    mxndx  = zeros(nnm,dtype=int)
    mnndx  = zeros(nnm,dtype=int)
    mxndy  = zeros(nnm,dtype=int)
    mnndy  = zeros(nnm,dtype=int)
    for i in range(nnm):
        if allclose(gnodes[i,0],0.0): # axis -> (x=0,
            y[:])
            bnode[i] = 1              # BC here =
                vanilla(s2,t)
            mnndx[i] = i
        elif allclose(gnodes[i,0],s1max):
            # axis -> (x=s1max, y[:]) BC here = 0.0
            bnode[i] = 1
            mxndx[i] = i

    for j in range(nnm):
        if allclose(gnodes[j,1],0.0): # axis -> (x
            =[:],y=0)
            bnode[j] = 1              # BC here =
                vanilla(s1,t)
            mnndy[j] = j
        elif allclose(gnodes[j,1],s2max):
            # axis -> (x=[:],y=s2max) BC here = 0.0
            bnode[j] = 1
```

```
                mxndy[j] = j

    # Create array of only the non-zero entries
    # These are the outer nodes.
    tmp1x = mnndx[mnndx.nonzero()]
    tmp2x = mxndx[mxndx.nonzero()]
    tmp1y = mnndy[mnndy.nonzero()]
    tmp2y = mxndy[mxndy.nonzero()]

    # must include the origin
    origin = 0
    mindx  = sort(append(tmp1x,origin))
    maxdx  = sort(append(tmp2x,origin))
    mindy  = sort(append(tmp1y,origin))
    maxdy  = sort(append(tmp2y,origin))

    # Need these global coords for time dependent BCs
        on
    # the boundaries. The convention used here is:
    # -> s1 = 0.0 and all S2 is the y-axis
    # -> s2 = 0.0 and all S1 is the x-axis
    s1y0  = zeros(len(mindy),dtype=float)
    s2x0  = zeros(len(mindx),dtype=float)

    # These are the actual global coordinates of the
    # outer nodes. These are required for the BC
    # calculation
    for i in range(len(mindy)):
        s1y0[i] = gnodes[mindy[i],0]

    for i in range(len(mindx)):
        s2x0[i] = gnodes[mindx[i],1]

    return bnode, mindx, maxdx, mindy, maxdy, s1y0,
        s2x0



# Create inital value (actually the "final" sol'n here
    )
def initialVal(K,gnodes,nnm,etype):
    u0    = zeros(nnm,dtype=float)
    for i in range(nnm):

        # Toggle the two definitions below for
        # a different exit strategies
        # See ACHDOU & PIRONNEAU eqn's [2.64] & [2.65]
        # You get very different graphs
        if allclose(etype,1.0):
            # [2.64]
            s1s2  = gnodes[i,0] + gnodes[i,1]
        else:
            # [2.65]
            s1s2  = max(gnodes[i,0], gnodes[i,1])

        test  = K - s1s2
        u0[i] = max(test,0.0)

    return u0


# Create the function to update the time-dependent BCs
def newBound(nnm,mindx,mindy,r,K,vol1,vol2,dt,s1y0,
    s2x0):
    newBC = zeros(nnm,dtype=float)

    # Call the vannila PUT function and use outer
    # nodal values
    s1bc = vanilla(r,K,dt,vol1,s1y0)
    s2bc = vanilla(r,K,dt,vol2,s2x0)

    # Set the values equal to the output from vanilla
    # NOTE: I assume that the min of S1 and S2 are at the
    # origin and that they are equal because they share the
    # same strike. The rest are zeros
    for i in range(len(s1bc)):
        bnod = mindy[i]
        newBC[bnod] = s1bc[i]

    for i in range(len(s2bc)):
        bnod = mindx[i]
        newBC[bnod] = s2bc[i]

    return newBC


# Get user input
# -------------------------------------------------------
app = femGUI.MyApp(False)
app.MainLoop()
inputs = femGUI.values

# Multiply everything by 1.0 or 1 to ensure we have SciPy dtype
# floats or integers as the GUI passes UNICODE STRINGS
```

```
    !!!
etype  = float(inputs[0])*1.0
s1high = float(inputs[1])*1.0
s2high = float(inputs[2])*1.0
vol1   = float(inputs[3])*1.0
vol2   = float(inputs[4])*1.0
rate   = float(inputs[5])*1.0
pcorr  = float(inputs[6])*1.0
K      = float(inputs[7])*1.0
lastT  = float(inputs[8])*1.0
dt     = float(inputs[9])*1.0
nx     = int(inputs[10])*1
ny     = int(inputs[11])*1

# Specify zero as the minimum value for the grid.
s1low  = 0.0
s2low  = 0.0

# Below for comparison purposes
# Comment out GUI inputs and run with the below values
#
# These values are the same used by ACHDOU & PIRONNEAU
# for the creation of Figures [4.11] and [4.12]
# These are the equivalent values for their THETA
    matrix
# using this formulation
## etype  = 1.0
## s1high = 150.0
## s2high = 150.0
## vol1   = 0.1414
## vol2   = 0.1414
## rate   = 0.1
## pcorr  = -0.6
## K       = 100.0
## lastT   = 0.70
## dt      = 0.01
## nx      = 50
## ny      = 50

# Initialize vectors/matrices
# Integer values for loops/sizes
nex1 = nx + 1
ney1 = ny + 1
nem  = 2*nx*ny
nnm  = nex1*ney1
npe  = 3
ndf  = 1
neq  = nnm*ndf
nn   = npe*ndf

# Number of quadrature points
nipf = 3

# Floats and arrays
x0 = s1low
y0 = s2low
dx = ones(nex1,float)*float((s1high/nx))
dy = ones(ney1,float)*float((s2high/ny))
dx[-1] = 0.0
dy[-1] = 0.0

# Create the differential eqn's coefficients
f0   = 0.0
c0   = 1.0
a110 = 0.5*(vol1**2.0)
a220 = 0.5*(vol2**2.0)
```

```
a120 = pcorr*vol1*vol2
b10  = rate
b20  = rate
G    = -1.0*rate

# Call Fortran Mesh routine
# NOTA BENE: The connectivity matirx NODF has indices
#             according to the FORTRAN CONVENTION!
nodf,glxy = tri.mesh(nx,ny,nex1,ney1,nem,nnm,dx,dy,x0,
    y0)

# Switch NODF indices for the Python convention
fort2py    = ones(shape(nodf),dtype=int)
nodp       = nodf - fort2py

# Find IdaigF and Idiag where they are the Fortran and
     Python
# index of the diagonal for the non-symmetric
    stiffness matrix
# respectively -> RECALL: Python starts indexing at 0!
IdiagF = 0
for i in range(nem):
    for j in range(npe):
        for k in range(npe):
            nw = (int(abs(nodf[i,j] - nodf[i,k])+1))*
                ndf
            if IdiagF < nw:
                IdiagF = nw

# Band width of sparse matrix
band   = (IdiagF*2) - 1
Idiag = IdiagF - 1

```

```python

# ---------------------------------------------------------#
#                                                          #
#                    Begin FEM Routine                     #
#                                                          #
# ---------------------------------------------------------#


# [1] Set time values
# Time dependent variables & Crank-Nicolson parameters
alfa   = 0.5
ntime  = int(lastT/dt) + 1
a1     = alfa*dt
a2     = (1.0 - alfa)*dt

# Create storage matrices for values at each time step
optionValue = zeros((nnm,ntime),dtype=float)
optionParam = zeros((nnm,ntime),dtype=float)

# [2] Initialize BCs

# Create "final" condition and store for option price calculation
# once all the values in time have been calculated
```

```
u0    = initialVal(K,glxy,nnm,etype)
glu   = u0

# Generate boundary information matrices from global
    matrix
bnode,mindx,maxdx,mindy,maxdy,s1y0,s2x0 = \
                                findbc(glxy,s1high,
                                    s2high,nnm)

# An array of Python indices
nwld = arange(nnm,dtype=int)

# [3] Enter time loop
time    = 0.0
ncount = 0
while ncount < ntime :

    # Find new BCs for future time step
    time += dt
    newBC = \
      newBound(nnm,mindx,mindy,rate,K,vol1,vol2,time,
          s1y0,s2x0)

    # Global matrices
    glk   = zeros((neq,band),dtype=float)
    glf   = zeros(neq,dtype=float)

    # Begin loop over each element
    for n in range(nem):
        # Element matrices
        elxy = zeros((npe,2),dtype=float)
        elu   = zeros(npe,dtype=float)
        elf   = zeros(npe,dtype=float)
```

```python
        elm  = zeros((npe,npe),dtype=float)
        elk  = zeros((npe,npe),dtype=float)

        for i in range(npe):
            # Assign global values for each node in the element
            ni = nodp[n,i]
            elxy[i,0] = glxy[ni,0]
            elxy[i,1] = glxy[ni,1]
            elu[i]    = glu[ni]

        # [4] Now compute elemental matrices
        # Load quadrature data from Fortran Module
        l1,l2,l3,lwt = GQD.quad()

        # [5] Begin quadtrature loop
        for nl in range(npe):
            ac1 = l1[nl]
            ac2 = l2[nl]
            ac3 = l3[nl]

            # Call Fortran Shape Function Module
            det,sf,gdsf = SFN.sfntri(ac1,ac2,ac3,elxy)
            cnst = 0.5*det*lwt[nl]

            # Global x an y in terms of the unit triangle
            x = 0.0
            y = 0.0
            for it in range(npe):
                x += elxy[it,0]*sf[it]
                y += elxy[it,1]*sf[it]

            # Set coefficients with mapped x and y coordinates
            a11 = a110*x*x
            a22 = a220*y*y
            a12 = a120*x*y
            b1  = b10*x
            b2 = b20*y
            source = f0
            ct     = c0

            # Create Elemental K, M, and F matrices/vector
            # by integrating over the element
            for ip in range(npe):
                for jp in range(npe):
                    s00 = sf[ip]*sf[jp]*cnst
                    s11 = gdsf[0,ip]*gdsf[0,jp]*cnst
                    s22 = gdsf[1,ip]*gdsf[1,jp]*cnst
                    s12 = gdsf[0,ip]*gdsf[1,jp]*cnst
                    s01 = sf[ip]*gdsf[0,jp]*cnst
                    s02 = sf[ip]*gdsf[1,jp]*cnst
                    # Now assemble ELEMENT MATRIX [K]
                    # using the form from THOMPSON
                    # [K]  = [S1] - [S2] - [S3] - [Sh] where
                    # [Sh] = 0.0 for this problem
                    elk[ip,jp] += (a11*s11 + a12*s12 + a22*s22)\
                                  - (b1*s01 + b2*s02) \
                                  - G*s00
                    elm[ip,jp] += ct*s00
                elf[ip] += cnst*sf[ip]*source
```

```python

        # [6] Apply CRANK-NICOLSON to find K^ and F^
        # See J.N. REDDY, eqn (6.42b)
        for ik in range(nn):
            summ = 0.0
            for jk in range(nn):
                summ += (elm[ik,jk] - a2*elk[ik,jk])*elu[jk]
                elk[ik,jk] = elm[ik,jk] + a1*elk[ik,jk]
            elf[ik] = (a1+a2)*elf[ik] + summ

        # [7] Assemble into global matrices using the
        # routine from THOMPSON for banded & non-symmetric
        for j in range(npe):
            jnp = nodp[n,j]
            jeq = nwld[jnp]
            glf[jeq] += elf[j]
            for k in range(npe):
                knp = nodp[n,k]
                keq = nwld[knp]
                kb  = (keq-jeq) + Idiag
                glk[jeq,kb] += elk[j,k]

    # [8] Apply BCs by BLASTING technique (also a THOMPSON thing)
    BLAST = 1.0e6
    for i in range(nnm):
        if allclose(bnode[i],1):
            nb = nwld[i]
            glu[nb] = newBC[i]
            glk[nb,Idiag] *= BLAST
            glf[nb] = glu[i]*glk[nb,Idiag]

    # [9] Solve GLOBAL MATRICES using Fortran
        nsymgauss module
    glu = NSG.nsymgauss(glk,glf,neq,band)

    # [10] Store data for visualization at the end
    oValu = glu
    oPara = glu - u0
    for i in range(nnm):
        optionValue[i,ncount] = oValu[i]
        optionParam[i,ncount] = oPara[i]

    # [11] Update the time loop and BCs for next time
        step
    ncount += 1

# END OF TIME LOOP HERE
    --------------------------------


# Visualize with MayaVi
#
    -------------------------------------------------------


# Set the z1 variable (second column is time)
##z1 = u0
z1 = optionValue[:,-1]
## z1 = optionParam[:,-1]

import pyvtk
# Scale the data in the Z-direction
```

```
dzz = dx[0]*2
dxx = dx[0]
dyy = dy[0]

# Convert z1 to vtk structured point data
# Note: No need to rearrange z1 as it is already in
    the
#       proper sequence from the meshing routine
point_data = pyvtk.PointData(pyvtk.Scalars(z1))

# Generate the grid sizing
grid=pyvtk.StructuredPoints((nex1,nex1,1),(0,0,0),(dxx
    ,dyy,dzz))

# Save to temporary file
data = pyvtk.VtkData(grid, point_data)
data.tofile('/tmp/test.vtk')

# Now use MayaVi to visualize
import mayavi
v = mayavi.mayavi() # create a MayaVi window.
d = v.open_vtk('/tmp/test.vtk', config=0) # open the
    data file.

# Load the filters.
f = v.load_filter('WarpScalar', config=0)
n = v.load_filter('PolyDataNormals', 0)
n.fil.SetFeatureAngle (45)

# Load the necessary modules.
m = v.load_module('SurfaceMap', 0)
a = v.load_module('Axes', 0)
t = v.load_module('Text',0)
```

```
o = v.load_module('Outline', 0)

# Re-render the scene.
v.Render()
v.master.wait_window()


# Or visualize with Matplotlib
#
    ----------------------------------------------------------


# An alternative option (for speed) is matplotlib
# Output data to figures using matplotlib below
## import pylab as p
## import matplotlib.axes3d as p3
## x  = reshape(glxy[:,0],(nex1,ney1))
## y  = reshape(glxy[:,1],(nex1,ney1))
## init_val = u0
## finalval = optionValue[:,-1]
## time_val = optionParam[:,-1]
## z1 = reshape(init_val,(nex1,ney1))
## z2 = reshape(finalval,(nex1,ney1))
## z3 = reshape(time_val,(nex1,ney1))

## # Make three figures
## fig1= p.figure(1)
## ax1 = p3.Axes3D(fig1)
## ax1.plot_wireframe(x,y,z1)
## ax1.set_xlabel('S1')
## ax1.set_ylabel('S2')
## ax1.set_zlabel('Final Condition at Expiry')

## fig2= p.figure(2)
## ax2 = p3.Axes3D(fig2)
## ax2.plot_wireframe(x,y,z2)
## ax2.set_xlabel('S1')
## ax2.set_ylabel('S2')
## ax2.set_zlabel('Option Value')

## fig3= p.figure(3)
## ax3 = p3.Axes3D(fig3)
## ax3.plot_wireframe(x,y,z3)
## ax3.set_xlabel('S1')
## ax3.set_ylabel('S2')
## ax3.set_zlabel('Time Value')

## # Show the plots: NOTE that you can rotate them
## # with a mouse
## p.show()
```

A.2 GUI Interface Class

Once again, please note that some of the formatting has changed to allow for the listings to within the textblock.

```
import wx

class FemInput(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, \
                          "Options␣Input␣Interface")
        panel = wx.Panel(self)

        # First create the controls

        # Title
        topLbl = wx.StaticText(panel, -1, \
                                  "FEM␣2D␣Basket␣Put␣Option
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\nBy␣Tyler␣Hayes",size=(420,-1))
        topLbl.SetFont(wx.Font(18, wx.SWISS, wx.NORMAL, wx.BOLD))

        # Choose Expiry Type
        sclabel  = wx.StaticText(panel, -1, \
                                    "Choose␣Expiry␣Type:\n
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Enter␣1␣for␣(K-(S1+S2))+␣or␣\n
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣2␣for␣(K-max(S1,S2)+␣",
                                    size=(220,-1))
        self.etype = wx.TextCtrl(panel, -1, "", size=(100,-1));


        # S1 and S2 upper bounds for grid
        s2label = wx.StaticText(panel, -1, "S1␣High,␣S2␣High:␣",\
                                   size=(220,-1))
        self.s1upper = wx.TextCtrl(panel, -1, "", size=(100,-1));
        self.s2upper = wx.TextCtrl(panel, -1, "", size=(100,-1));

        # S1 and S2 volatility
        vlabel = wx.StaticText(panel, -1, "S1␣
            Volatility,
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣S2␣Volatility:␣", size=(220,-1))
        self.v1vol = wx.TextCtrl(panel, -1, "", size=(100,-1));
        self.v2vol = wx.TextCtrl(panel, -1, "", size=(100,-1));

        # Risk free rate and correlation
        prlabel = wx.StaticText(panel, -1, "Interest␣Rate,
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Correlation:␣", size=(220,-1))
        self.risk = wx.TextCtrl(panel, -1, "", size=(100,-1));
        self.corr = wx.TextCtrl(panel, -1, "", size=(100,-1));


        # Strike and Exercise Date
        kTlabel = wx.StaticText(panel, -1, "Srike␣Price,
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Exercise␣Date:␣", size=(220,-1))
        self.strike = wx.TextCtrl(panel, -1, "", size=(100,-1));
        self.finalT = wx.TextCtrl(panel, -1, "", size=(100,-1));

        # deltaT and deltaX
        dTXlabel = wx.StaticText(panel, -1, "delta␣T,␣NX,␣NY:␣",\
                                  size=(220,-1))
        self.deltaT = wx.TextCtrl(panel, -1, "", size=(100,-1));
        self.nxval  = wx.TextCtrl(panel, -1, "", size=(100,-1));
        self.nyval  = wx.TextCtrl(panel, -1, "", size=(100,-1));


        # Execute program
        runBtn = wx.Button(panel, -1, "Run")
        self.Bind(wx.EVT_BUTTON, self.OnSubmit, runBtn)

        # Now do the layout.

        # mainSizer is the top-level one that manages everything
        mainSizer = wx.BoxSizer(wx.VERTICAL)
        mainSizer.Add(topLbl, 0, wx.ALL, 5)
        mainSizer.Add(wx.StaticLine(panel), 0,
                wx.EXPAND|wx.TOP|wx.BOTTOM, 5)

        # femSizer is a grid that holds all of the address info
        femSizer = wx.FlexGridSizer(cols=2, hgap=5, vgap=5)
        femSizer.AddGrowableCol(1)

        # Expiry Type
        femSizer.Add(sclabel, 0,
                wx.ALIGN_RIGHT|wx.ALIGN_CENTER_VERTICAL)
        # the lower and upper bounds are in a sub-
        sizer
        etSizer = wx.BoxSizer(wx.HORIZONTAL)
        etSizer.Add(self.etype, 1)
        femSizer.Add(etSizer, 1, wx.EXPAND)


        # S1 and S2 HIGH label
        femSizer.Add(s2label, 0,
                wx.ALIGN_RIGHT|wx.ALIGN_CENTER_VERTICAL)
        # the lower and upper bounds are in a sub-sizer
        s2Sizer = wx.BoxSizer(wx.HORIZONTAL)
        s2Sizer.Add(self.s1upper, 1)
        s2Sizer.Add((10,10)) # some empty space
        s2Sizer.Add(self.s2upper, 1, wx.LEFT|wx.RIGHT, 5)
        femSizer.Add(s2Sizer, 1, wx.EXPAND)


        # Volatility label
        femSizer.Add(vlabel, 0,
                wx.ALIGN_RIGHT|wx.ALIGN_CENTER_VERTICAL)
        # the lower and upper bounds are in a sub-sizer
        volSizer = wx.BoxSizer(wx.HORIZONTAL)
        volSizer.Add(self.v1vol, 1)
        volSizer.Add((10,10)) # some empty space
        volSizer.Add(self.v2vol, 1, wx.LEFT|wx.RIGHT, 5)
        femSizer.Add(volSizer, 1, wx.EXPAND)


        # Risk free Rate and corelation
        femSizer.Add(prlabel, 0,
                wx.ALIGN_RIGHT|wx.
                    ALIGN_CENTER_VERTICAL)
        # the lower and upper bounds are in a sub-
            sizer
        rcSizer = wx.BoxSizer(wx.HORIZONTAL)
        rcSizer.Add(self.risk, 1)
        rcSizer.Add((10,10)) # some empty space
        rcSizer.Add(self.corr, 1, wx.LEFT|wx.RIGHT, 5)
        femSizer.Add(rcSizer, 1, wx.EXPAND)


        # Strike and Exercise Date
        femSizer.Add(kTlabel, 0,
                wx.ALIGN_RIGHT|wx.
                    ALIGN_CENTER_VERTICAL)
        # the lower and upper bounds are in a sub-
            sizer
        ktSizer = wx.BoxSizer(wx.HORIZONTAL)
        ktSizer.Add(self.strike, 1)
        ktSizer.Add((10,10)) # some empty space
        ktSizer.Add(self.finalT, 1, wx.LEFT|wx.RIGHT,
            5)
        femSizer.Add(ktSizer, 1, wx.EXPAND)


        # deltaT and deltaX
        femSizer.Add(dTXlabel, 0,
                wx.ALIGN_RIGHT|wx.
                    ALIGN_CENTER_VERTICAL)
        # the lower and upper bounds are in a sub-
```

```
            sizer
        dtxSizer = wx.BoxSizer(wx.HORIZONTAL)
        dtxSizer.Add(self.deltaT, 1)
        dtxSizer.Add((10,10)) # some empty space
        dtxSizer.Add(self.nxval, 1, wx.LEFT|wx.RIGHT,
            5)
        dtxSizer.Add((10,10)) # some empty space
        dtxSizer.Add(self.nyval, 1, wx.LEFT|wx.RIGHT,
            5)
        femSizer.Add(dtxSizer, 1, wx.EXPAND)


        # now add the femSizer to the mainSizer
        mainSizer.Add(femSizer, 0, wx.EXPAND|wx.ALL,
            10)

        # gaps between and on either side of the
            buttons
        btnSizer = wx.BoxSizer(wx.HORIZONTAL)
        btnSizer.Add((10,10)) # some empty space
        btnSizer.Add(runBtn)
        btnSizer.Add((10,10)) # some empty space
        mainSizer.Add(btnSizer, 0, wx.EXPAND|wx.BOTTOM
            , 10)

        panel.SetSizer(mainSizer)

        # Fit the frame to the needs of the sizer.
            The frame
        # will automatically resize the panel as
            needed.
        # Also prevent the frame from getting smaller
            than
        # this size.
        mainSizer.Fit(self)
        mainSizer.SetSizeHints(self)

    def OnSubmit(self, evt):
        # Allow the inputs to be viewed by the calling
            program
        global values
        values = (self.etype.GetValue(),
                  self.s1upper.GetValue(),
                  self.s2upper.GetValue(),
                  self.v1vol.GetValue(),
                  self.v2vol.GetValue(),
                  self.risk.GetValue(),
                  self.corr.GetValue(),
                  self.strike.GetValue(),
                  self.finalT.GetValue(),
                  self.deltaT.GetValue(),
                  self.nxval.GetValue(),
                  self.nyval.GetValue())
        self.Close(True)

class MyApp(wx.App):

    def OnInit(self):
        frame = FemInput()
        self.SetTopWindow(frame)
        frame.Show()
        return True


# Needed if called as a module
if __name__ == '__main__':
    app = MyApp(False)
    app.MainLoop()
```

Appendix B

FORTRAN CODE LISTINGS

B.1 Fortran Mesh Routine

```
! This is the 2D mesh generator for linear, triangular elements
!
! This code is modified from J.N. REDDY, AN INTRODUCTION TO THE
! FINITE ELEMENT METHOD, 2nd EDITION
!
! Coded by Tyler Hayes, April 3, 2007
!
!
! INPUTS:
!
!       NX = NUMBER OF DIVISIONS IN THE X-DIR
!       NY = NUMBER OF DIVISIONS IN THE Y-DIR
!     NEX1 = INTEGER = NX+1
!     NEY1 = INTEGER = NY+1
!      NEM = INTEGER = 2*NX*NY
!      NNM = INTEGER = ((IEL*NX)+1) * ((IEL*NY)+1) IEL = 1 for tri
!       DX = VECTOR OF SPACINGS IN THE X-DIR <- CAN BE VARIABLE
!            THE VECTOR SHOULD BE NX+1 IN SIZE BUT THE DX(NX+1) IS
!             SET TO ZERO AND IS A DUMMY VARIABLE
!       DY = VECTOR OF SPACINGS IN THE Y-DIR <- CAN BE
   VARIABLE
!             THE VECTOR SHOULD BE NY+1 IN SIZE BUT THE
   DY(NY+1) IS
!             SET TO ZERO AND IS A DUMMY VARIABLE
!       X0 = ORIGIN OF THE X AXIS
!       Y0 = ORIGIN OF THE Y AXIS
!
! OUTPUT:
!       NOD = INTEGER MATRIX OF ELEMENT NOD INDICES
!      GLXY = GLOBAL COORDINATES OF NOD
!
! MISC.:
!       NPE = NODES PER ELEMENT
!
   ----------------------------------------------------------------


SUBROUTINE MESH(NOD,GLXY,NX,NY,NEX1,NEY1,NEM,NNM,DX,DY
   ,X0,Y0)
  IMPLICIT NONE
  INTEGER :: NXX,NYY,NX2,NY2,NXX1,NYY1
  INTEGER :: K,IY,L,M,N,I,NI,NJ,IEL
  INTEGER,PARAMETER :: NPE=3
  REAL :: XC,YC
! INPUTS
  INTEGER,INTENT(IN) :: NX,NY,NEM,NNM,NEX1,NEY1
  REAL,INTENT(IN) :: X0,Y0
  REAL,DIMENSION(NEX1),INTENT(IN) :: DX
  REAL,DIMENSION(NEY1),INTENT(IN) :: DY
! PARAMETERS
! OUTPUTS
```

```
  INTEGER,DIMENSION(NEM,NPE),INTENT(OUT) :: NOD
  REAL,DIMENSION(NNM,2),INTENT(OUT) :: GLXY

! CREATE VARIABLES
  IEL  = 1 ! FOR <= 4 NODES PER ELEMENT
  NXX  = IEL*NX
  NYY  = IEL*NY
  NXX1 = NXX + 1
  NYY1 = NYY + 1
  NX2  = 2*NX
  NY2  = 2*NY


! CREATE TRIANGUALR ELEMENTS
!
    ------------------------------------------------------------------


! INITIALIZE FIRST TWO ELEMENTS
  NOD(1,1) = 1
  NOD(1,2) = IEL+1
  NOD(1,3) = IEL*NXX1 + IEL + 1
  NOD(2,1) = 1
  NOD(2,2) = NOD(1,3)
  NOD(2,3) = IEL*NXX1 + 1

! LOOP THROUGH MESH
  K=3
  DO IY=1,NY
     L=IY*NX2
     M=(IY-1)*NX2
     IF (NX > 1) THEN
        DO N=K,L,2
```

```
            DO I=1,NPE
               NOD(N,I)   = NOD(N-2,I) + IEL
               NOD(N+1,I) = NOD(N-1,I) + IEL
            END DO
         END DO
      END IF
      IF (IY < NY) THEN
         DO I=1,NPE
            NOD(L+1,I) = NOD(M+1,I) + IEL*NXX1
            NOD(L+2,I) = NOD(M+2,I) + IEL*NXX1
         END DO
      END IF
      K=L+3
   END DO


! NOW GENERATE GLOBAL COORDINATES OF THE NODES
   XC = X0
   YC = Y0

   DO NI=1,NEY1
      XC = X0
      I  = NXX1*IEL*(NI-1)
      DO NJ = 1,NEX1
         I=I+1
         GLXY(I,1) = XC
         GLXY(I,2) = YC
         IF (NJ < NEX1) THEN
            IF (IEL == 2) THEN
               I=I+1
               XC = XC + 0.5*DX(NJ)
               GLXY(I,1) = XC
               GLXY(I,2) = YC
            END IF
         END IF
         XC = XC + DX(NJ)/IEL
      END DO
      XC = X0
      IF (IEL == 2) THEN
         YC = YC + 0.5*DY(NI)
         DO NJ = 1,NEX1
            I=I+1
            GLXY(I,1) = XC
            GLXY(I,2) = YC
            IF (NJ < NEX1) THEN
               I=I+1
               XC = XC + 0.5*DX(NJ)
               GLXY(I,1) = XC
               GLXY(I,2) = YC
            END IF
            XC = XC + 0.5*DX(NJ)
         END DO
      END IF
      YC = YC + DY(NI)/IEL
   END DO
END SUBROUTINE MESH
```

B.2 Fortran Shape Function Routine

```
SUBROUTINE SFNTRI(DET,SF,GDSF,L1,L2,L3,ELXY)
  IMPLICIT NONE
! THIS IS THE SUBROUTINE THAT CALCULATES THE SHAPE
   FUNCTIONS
! AND THEIR DERIVATIVES AT SPECIFIED GLOBAL POSITIONS
  INTEGER,PARAMETER :: NPE=3
  INTEGER :: I,J,K
  REAL,DIMENSION(3,2),INTENT(IN) :: ELXY
  REAL,INTENT(IN) :: L1,L2,L3
  REAL,INTENT(OUT) :: DET
  REAL,DIMENSION(3),INTENT(OUT) :: SF
  REAL,DIMENSION(2,3),INTENT(OUT) :: GDSF
  REAL,DIMENSION(3,3) :: DSF
  REAL,DIMENSION(2,2) :: GJ,GJINV
  REAL :: SUM

! INITIALIZE ARRAYS
  DO I=1,NPE
     DSF(1,I) = 0.0
     DSF(2,I) = 0.0
     DSF(3,I) = 0.0
     SF(I)    = 0.0
  END DO

! SET THE SHAPE FUNCTIONS FOR A TRIANGLE
  SF(1) = L1
  SF(2) = L2
  SF(3) = L3
  DSF(1,1) = 1.0
  DSF(2,2) = 1.0
  DSF(3,3) = 1.0

! FIND THE JACOBIAN
  DO I=1,2
     DO J=1,2
        SUM = 0.0
        DO K=1,NPE
           SUM = SUM + (DSF(I,K) - DSF(3,K))*ELXY(K,J)
        END DO
```

```
         GJ(I,J) = SUM
      END DO
   END DO

! FIND GJ's INVERSE
   DET        =  GJ(1,1)*GJ(2,2) - GJ(1,2)*GJ(2,1)
   GJINV(1,1) =  GJ(2,2)/DET
   GJINV(2,2) =  GJ(1,1)/DET
   GJINV(1,2) = -1.0*GJ(1,2)/DET
   GJINV(2,1) = -1.0*GJ(2,1)/DET

! OBTAIN THE DERIVATIVE OF THE SHAPE FUNCTION WITH
   PROPER
! TRANSFORMATION
   DO I=1,2
      DO J=1,NPE
         SUM = 0.0
         DO K=1,2
            SUM = SUM + GJINV(I,K)*(DSF(K,J) - DSF(3,J)
               )
         END DO
         GDSF(I,J) = SUM
      END DO
   END DO

END SUBROUTINE SFNTRI
```

B.3 Fortran Non-symmetric Gaussian Solver

```
! This is a subroutine that finds y from A.y = F for
! banded NON-symmetric matrices. From Thompson text
!
```

```fortran
! INPUTS:
!          A   = non-symmetric banded matrix
!          F   = vector from A.Y = F
!          IB = bandwidth of matrix (columns)
!          neq= number of rows in the matrix
!
! OUTPUTS:
!          Y   = solution vector from A.Y = F
!
    ----------------------------------------------------------------------------

SUBROUTINE nsymgauss(Y,A,F,neq,IB)
  IMPLICIT NONE
  INTEGER :: Idiag,i,j,k,Jend,Kbgn,Kend,Ikc,Jkc,Iback,
      Jc,Kc
  REAL :: FAC
  INTEGER, INTENT(IN) :: neq,IB
  REAL,DIMENSION(neq,IB) :: AA
  REAL,DIMENSION(neq) :: FF
  REAL,DIMENSION(neq,IB),INTENT(IN) :: A
  REAL,DIMENSION(neq),INTENT(IN) :: F
  REAL,DIMENSION(neq),INTENT(OUT) :: Y

! Initialize matrices to overwrite
  DO i = 1,neq
     DO j = 1,IB
        AA(i,j) = A(i,j)
     END DO
     FF(i) = F(i)
  END DO

! Forward elimination
  Idiag = ((IB - 1)/2) + 1
  DO i = 1,neq-1
     Jend = neq
     IF (Jend > (i+Idiag-1)) Jend = i+Idiag-1
     DO j = i+1,Jend
        Kc   = Idiag - (j-i)
        FAC  = -AA(j,Kc)/AA(i,Idiag)
        Kbgn = i
        Kend = Jend
        DO k = Kbgn,Kend
           Ikc = Idiag + (k-i)
           Jkc = Idiag + (k-j)
           AA(j,Jkc) = AA(j,Jkc) + FAC*AA(i,Ikc)
        END DO
        FF(j) = FF(j) + FAC*FF(i)
     END DO
  END DO

! Backward substitution
  Y(neq) = FF(neq)/AA(neq,Idiag)
  DO Iback = 2,neq
     i    = neq - Iback + 1
     Jend = neq
     IF (Jend > (i+(Idiag-1))) Jend = i+(Idiag-1)
     DO j = i+1,Jend
        Jc    = Idiag + (j-i)
        FF(i) = FF(i) - AA(i,Jc)*Y(j)
     END DO
     Y(i) = FF(i)/AA(i,Idiag)
  END DO
END SUBROUTINE nsymgauss
```

B.4 Fortran Quadrature Data Function

```
SUBROUTINE QUAD(L1,L2,L3,LWT)
  IMPLICIT NONE
! THIS CREATES THE TABLE OF VALUES FOR TRIANGULAR
   ELEMENTS
! USED FOR THREE-POINT QUADRATURE
!
!  INTEGRATION REGION:
!
!       0 <= X,   AND   0 <= Y,   AND   X + Y <= 1.
!
!  GRAPH:
!
!       ^
!     1 | *
!       | |\
!     Y | | \
!       | |  \
!     0 | *---*
!       +------->
!         0 X 1
!
!
!
INTEGER :: I
REAL,DIMENSION(3),INTENT(OUT) :: L1,L2,L3,LWT

! INITIALIZE
DO I=1,3
   L1(I)  = 0.0
   L2(I)  = 0.0
   L3(I)  = 0.0
```

```fortran
   LWT(I) = 0.0
END DO


! THREE-POINT QUADRATURE
L1(1)  = 0.0
L1(2)  = 0.5
L1(3)  = 0.5
L2(1)  = 0.5
L2(2)  = 0.0
L2(3)  = 0.5
L3(1)  = 0.5
L3(2)  = 0.5
L3(3)  = 0.0
LWT(1) = 1.0/3.0
LWT(2) = 1.0/3.0
LWT(3) = 1.0/3.0
END SUBROUTINE QUAD
```

Appendix C

PYTHON F2PY INTERFACE MODULES

The following modules are required by f2py. They are used to properly compile the wrapper functions that allow the Fortran routines in B to be called within the main FEM routine listed in A.

C.1 Mesh Interface File

```
!    -*- f90 -*-
! Note: the context of this file is case sensitive.

python module tri ! in
    interface  ! in :tri
        subroutine mesh(nod,glxy,nx,ny,nex1,
          ney1,nem,nnm,dx,dy,x0,y0) ! in :tri:mesh.f90
            integer dimension(nem,3),intent(out),
            depend(nem) :: nod
            real dimension(nnm,2),intent(out),depend(
                nnm) :: glxy
            integer intent(in) :: nx
            integer intent(in) :: ny
            integer intent(in) :: nex1
            integer intent(in) :: ney1
            integer intent(in) :: nem
            integer intent(in) :: nnm
            real dimension(nex1),intent(in),depend(
                nex1) :: dx
            real dimension(ney1),intent(in),depend(
               ney1) :: dy
            real intent(in) :: x0
            real intent(in) :: y0
        end subroutine mesh
    end interface
end python module tri

! This file was auto-generated with f2py (version:2
   _3396).
! See http://cens.ioc.ee/projects/f2py2e/
```

C.2 Shape Function Interface File

```
!    -*- f90 -*-
! Note: the context of this file is case sensitive.

python module shpfn ! in
    interface  ! in :shpfn
        subroutine sfntri(det,sf,gdsf,l1,l2,l3,elxy)
! in :shpfn:sfntri.f90
            real intent(out) :: det
            real dimension(3),intent(out) :: sf
            real dimension(2,3),intent(out) :: gdsf
            real intent(in) :: l1
            real intent(in) :: l2
            real intent(in) :: l3
            real dimension(3,2),intent(in) :: elxy
        end subroutine sfntri
    end interface
end python module shpfn

```

```
! This file was auto-generated with f2py (version:2
    _3396).
! See http://cens.ioc.ee/projects/f2py2e/
```

C.3 Gaussian Solver Interface File

```
!    -*- f90 -*-
! Note: the context of this file is case sensitive.

python module nsymgauss ! in
    interface  ! in :nsymgauss
        subroutine nsymgauss(y,a,f,neq,ib)
! in :nsymgauss:nsymgauss.f90
            integer intent(in) :: neq
            integer intent(in) :: ib
            real dimension(neq),intent(out),depend(neq
               ) :: y
            real dimension(neq,ib),intent(in),
            depend(ib,neq) :: a
            real dimension(neq),intent(in),depend(neq)
                :: f
        end subroutine nsymgauss
    end interface
end python module nsymgauss

! This file was auto-generated with f2py (version:2
   _3396).
! See http://cens.ioc.ee/projects/f2py2e/
```

C.4 Quadrature Data Interface File

```
!    -*- f90 -*-
! Note: the context of this file is case sensitive.

python module quad_data ! in
    interface  ! in :quad_data
```

```
        subroutine quad(l1,l2,l3,lwt) ! in :quad_data:
            quad.f90
            real dimension(3),intent(out) :: l1
            real dimension(3),intent(out) :: l2
            real dimension(3),intent(out) :: l3
            real dimension(3),intent(out) :: lwt
        end subroutine quad
    end interface
end python module quad_data

! This file was auto-generated with f2py (version:2
    _3396).
! See http://cens.ioc.ee/projects/f2py2e/
```

SOFTWARE USED

In the course of preparing this project, I made use of freely available software. The computer operating system with which this FEM project was carried out on is Kubuntu 6.06 (Dapper Drake), a Debian derivative. The main portion of my code was written using Python and the GUI widgets from wxPython. I also made use of several science specific extensions to Python. For numerical purposes, I used SciPy and its imported libraries as well as the `f2py` extension which automatically generates a wrapper around Fortran 77/90 programs to facilitate calling them within Python. The Fortran 90 routines were complied using GNU Fortran and written on the world's best text editor, GNU Emacs. Links to the software are listed below (in no particular order).

http://www.gnu.org/software/emacs/
http://cens.ioc.ee/projects/f2py2e/
http://www.python.org/
http://www.wxpython.org/
http://gcc.gnu.org/fortran/
http://www.kubuntu.org/
http://www.scipy.org/

COLOPHON

This report was written using the LaTeX typesetting system with the Century Old Style fonts from THE FONTSITE and the text block was sized to correspond to the Golden Ratio, $\varphi = 1.618\ldots$.