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.

Michael 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,

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

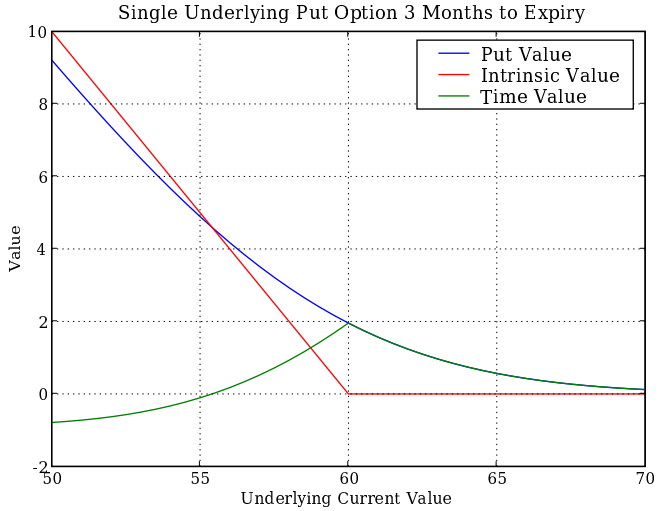

Figure 1.1: Put value for a strike at \$60, 0.25 years to maturity, r =0.06 and a volatility (σ) of 0.2.

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:

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

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

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.,

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

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

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

$$P(S_1,0,t)=g(S_1,K,t) \quad (1.9a)$$

$$P(0,S_2,t)=g(S_2,K,t) \quad (1.9b)$$

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.

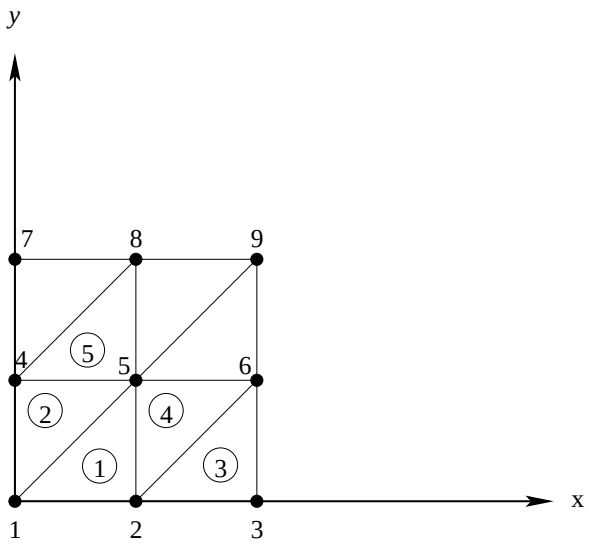

Figure 2.1: The mesh used for triangular elements in this analysis. Circled numbers are the element numbers (not all shown). Others are the node numbers. Modified from Reddy (1993).

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.

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

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

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.

$$[K]\{\Phi\} + [C]\left\{\dot{\Phi}\right\} = \{Q\} \quad (2.4)$$

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.

$$[K] = [S_1] + [S_2] + [S_3] \quad (2.5)$$

where,

$$[S_1] = \int_{V_{uv}} \left\{N'\right\}[A]\left\lfloor N'\right\rfloor \|J\| dV \quad (2.6)$$

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

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

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

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

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

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.

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

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.

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

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

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

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

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

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.

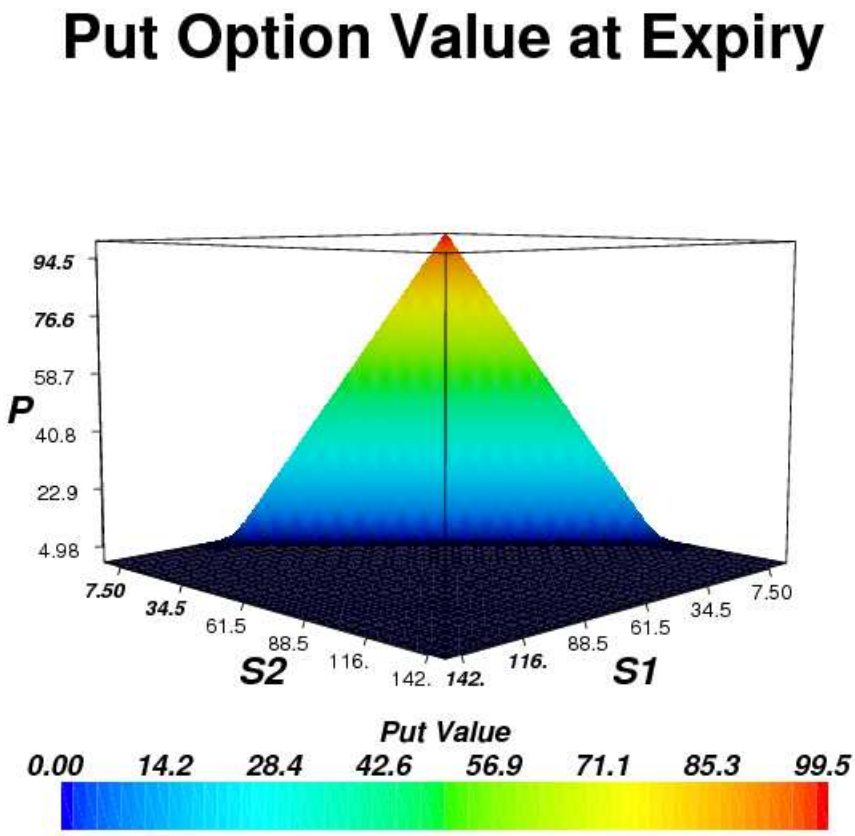

Figure 3.1: Final condition (1.7a).

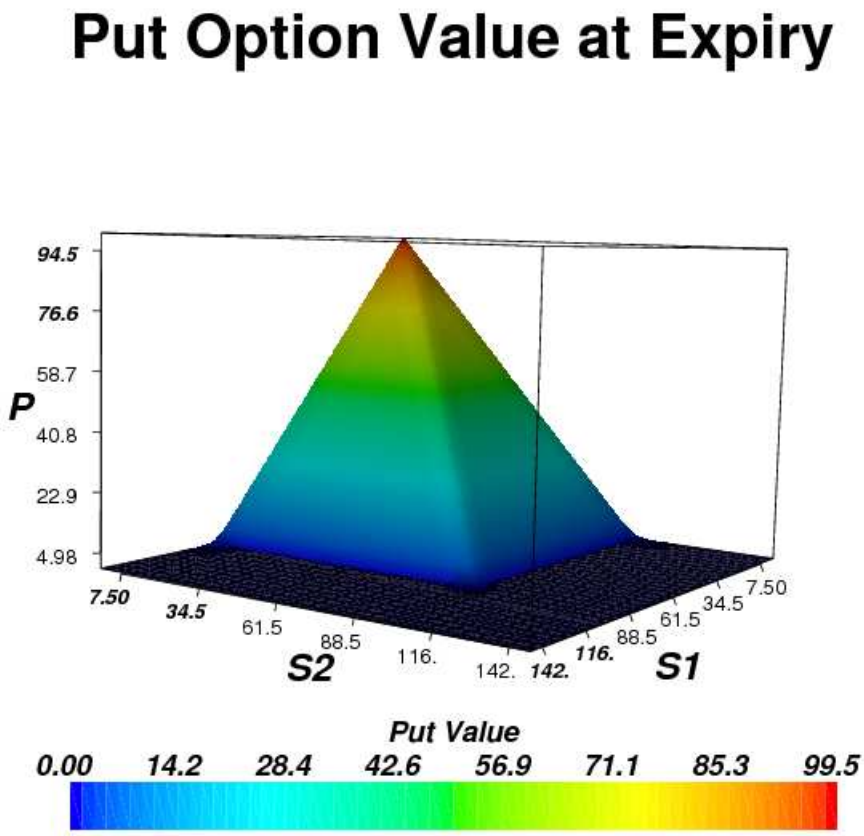

Figure 3.2: Final condition (1.7b).

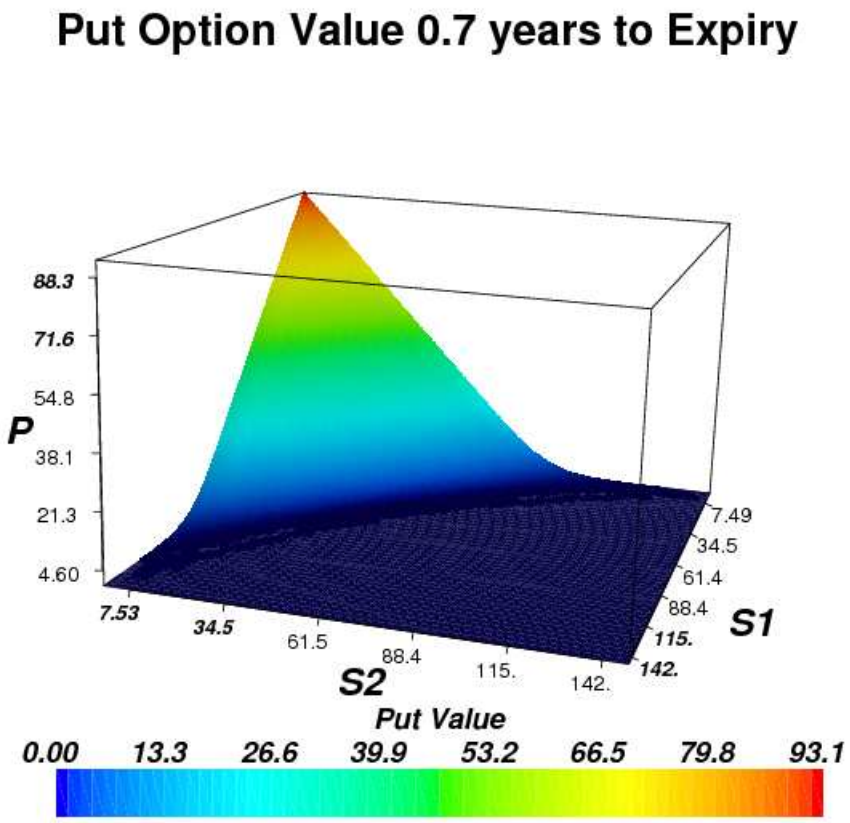

Figure 3.3: Final condition (1.7a) at 0.7 years from maturity.

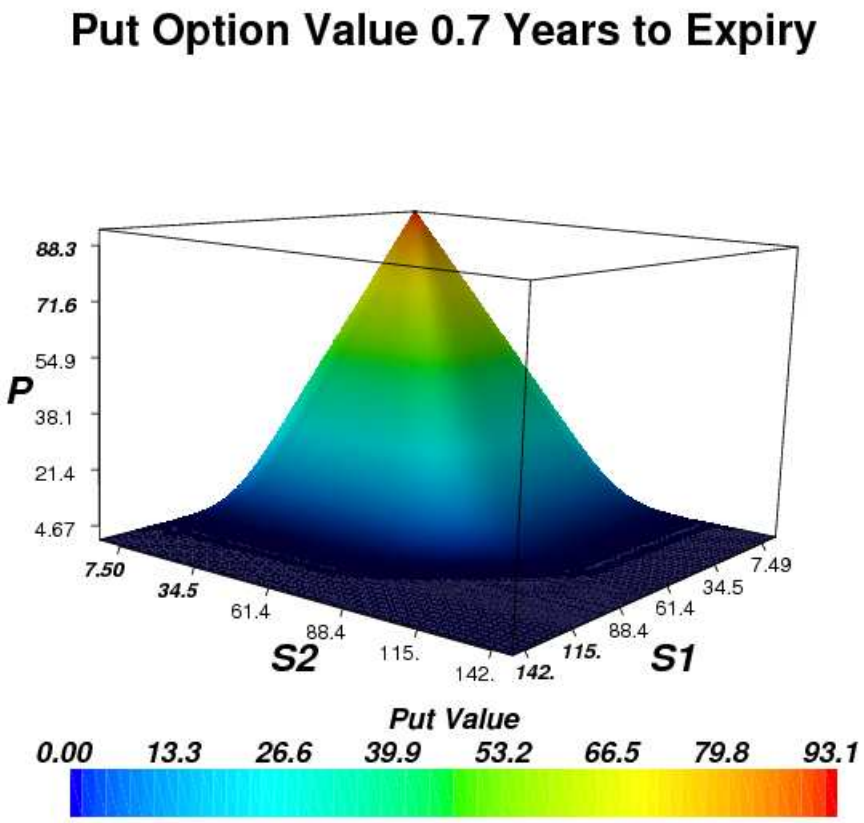

Figure 3.4: Final condition (1.7b) at 0.7 years from maturity.

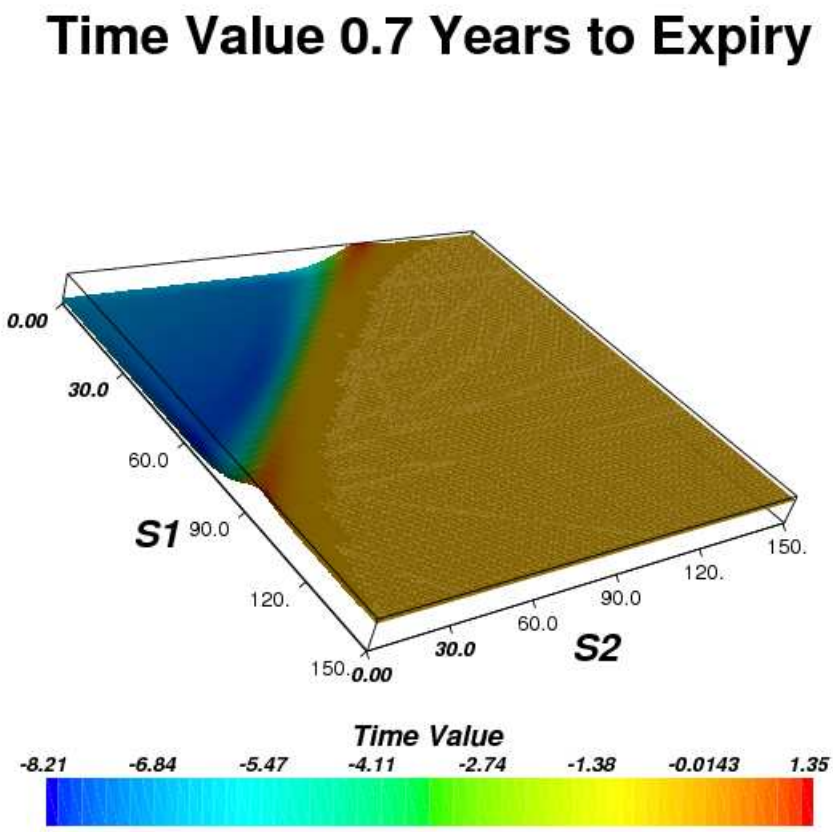

Figure 3.5: Time value of (1.7a) at 0.7 years from maturity.

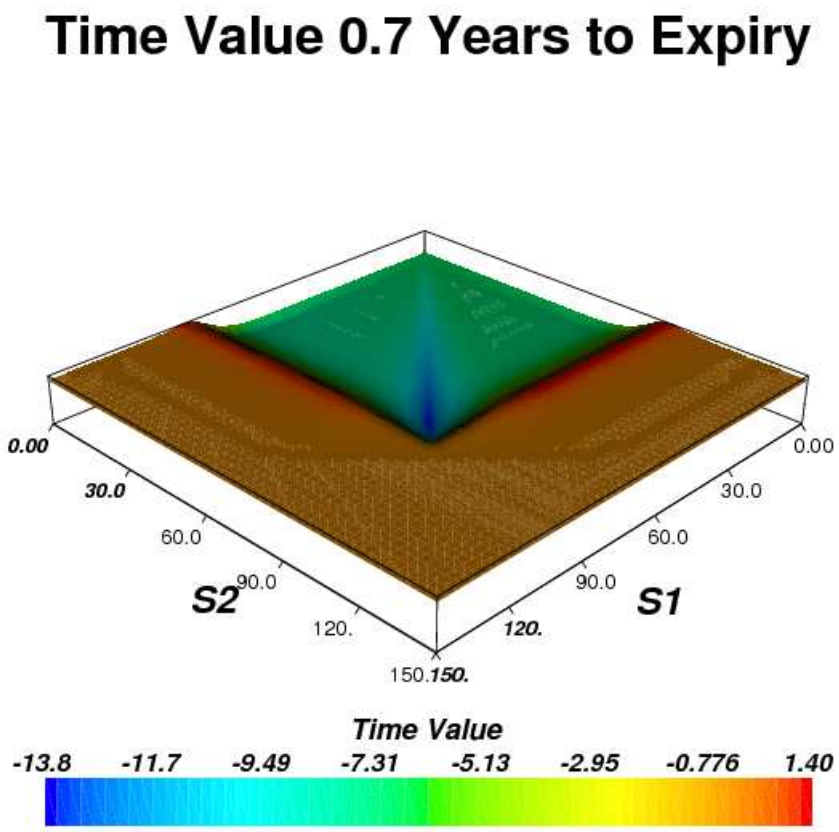

Figure 3.6: Time value of (1.7b) at 0.7 years from maturity.

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.

```
1  #!/usr/bin/env python
2
3  # Import Modules
4  #-----------------------------------------------------------
5
6  # Scipy and Numpy
7  from scipy import *
8
9  # Mesh generator
10 import tri
11
12 # Non-symmetric Gaussian Solver
13 import nsymgauss as NSG
14
15 # Quadrature data
16 import quad_data as GQD
17
18 # Shape function module
19 import shpfn as SFN
```

```
20
21 # GUI
22 import femGUI
23
24
25 # Define python functions
26 #---------------------------------------------------------
27 def vanilla(r, K, dt, sigma, S):
28     """
29     This is a simple Vanilla Put Option calcualtion
30     based on the analytic solution for a single
31     underlying asset. The solution used is from
32     The Mathematics of Financial Derivatives, Wilmott, et al.
33     Uses ndtr and exp from scipy and ndtr scipy.special modules.
34
35     r     : risk free rate (float)
36     K     : strike price (float)
37     dt    : time to expiry (float)
38     sigma : volatility of S (float)
39     S     : range of underlying values (array[float])
40
41     Usage:
42
43     put_value = vanilla(r, K, dt, sigma, S)
44
45     """
46
47     d1 = zeros(len(S))
48     d2 = zeros(len(S))
49     n1 = zeros(len(S))
50     n2 = zeros(len(S))
51     pt = zeros(len(S))
52     b  = sigma*sqrt( dt)
53     dsct = exp(-1.0*r*dt)
54     for i in range(len(S)):
55         d1[i] = (log(S[i]/K) + (r + (0.5* sigma**2))*dt)/b
56         d2[i] = (log(S[i]/K) + (r - (0.5* sigma**2))*dt)/b
57         n1[i] = special.ndtr(-1.0*d1[i])
58         n2[i] = special.ndtr(-1.0*d2[i])
```

```
59          pt[i] = K*dsct*n2[i] - S[i]*n1[i]
60
61      return pt
62
63
64  # Create the boundary information arrays only once to
65  # save calculations during time-dependent BCs
66  def findbc(gnodes,s1max,s2max,nnm):
67      """
68  ____This function will return an array of values for
69  ____which global nodes lie on the boundaries.
70
71  ____bnode:
72  ____________0 = interior node
73  ____________1 = boundary node
74
75  ____mind*: the indices of the axes' s1min, s2min nodes
76  ____maxd*: the indices of the axes' s1max, s2max nodes
77
78  ____gnodes is an array of size (num of nodes) x 2
79  ____gnodes[:,0] = global x values
80  ____gnodes[:,1] = global y values
81
82  ____bnode,mindx,maxdx,mindy,maxdy,s1y0,s2x0
83  ____________= findbc(gnodes,s1max,s2max,nnm)
84  ____"""
85
86      # The "max" matrices are not actually called in the present
87      # program, but may be needed later.
88      bnode  = zeros(nnm,dtype=int)
89      mxndx  = zeros(nnm,dtype=int)
90      mnndx  = zeros(nnm,dtype=int)
91      mxndy  = zeros(nnm,dtype=int)
92      mnndy  = zeros(nnm,dtype=int)
93      for i in range(nnm):
94          if allclose(gnodes[i,0],0.0): # axis -> (x=0, y[:])
95              bnode[i] = 1              # BC here = vanilla(s2,t)
96              mnndx[i] = i
97          elif allclose(gnodes[i,0],s1max):
```

```
98              # axis -> (x=s1max, y[:]) BC here = 0.0
99              bnode[i] = 1
100             mxndx[i] = i
101
102     for j in range(nnm):
103         if allclose(gnodes[j,1],0.0): # axis -> (x=[:],y=0)
104             bnode[j] = 1              # BC here = vanilla(s1,t)
105             mnndy[j] = j
106         elif allclose(gnodes[j,1],s2max):
107             # axis -> (x=[:],y=s2max) BC here = 0.0
108             bnode[j] = 1
109             mxndy[j] = j
110
111     # Create array of only the non-zero entries
112     # These are the outer nodes.
113     tmp1x = mnndx[mnndx.nonzero()]
114     tmp2x = mxndx[mxndx.nonzero()]
115     tmp1y = mnndy[mnndy.nonzero()]
116     tmp2y = mxndy[mxndy.nonzero()]
117
118     # must include the origin
119     origin = 0
120     mindx  = sort(append(tmp1x,origin))
121     maxdx  = sort(append(tmp2x,origin))
122     mindy  = sort(append(tmp1y,origin))
123     maxdy  = sort(append(tmp2y,origin))
124
125     # Need these global coords for time dependent BCs on
126     # the boundaries. The convention used here is:
127     # -> s1 = 0.0 and all S2 is the y-axis
128     # -> s2 = 0.0 and all S1 is the x-axis
129     s1yo  = zeros(len(mindy),dtype=float)
130     s2xo  = zeros(len(mindx),dtype=float)
131
132     # These are the actual global coordinates of the
133     # outer nodes. These are required for the BC
134     # calculation
135     for i in range(len(mindy)):
136         s1yo[i] = gnodes[mindy[i],0]
```

```
137
138     for i in range(len(mindx)):
139         s2x0[i] = gnodes[mindx[i],1]
140
141     return bnode, mindx, maxdx, mindy, maxdy, s1y0, s2x0
142
143
144
145 # Create inital value (actually the "final" sol'n here)
146 def initialVal(K,gnodes,nnm,etype):
147     u0   = zeros(nnm,dtype=float)
148     for i in range(nnm):
149
150         # Toggle the two definitions below for
151         # a different exit strategies
152         # See ACHDOU & PIRONNEAU eqn's [2.64] & [2.65]
153         # You get very different graphs
154         if allclose(etype,1.0):
155             # [2.64]
156             s1s2  = gnodes[i,0] + gnodes[i,1]
157         else:
158             # [2.65]
159             s1s2  = max(gnodes[i,0], gnodes[i,1])
160
161         test  = K - s1s2
162         u0[i] = max(test,0.0)
163
164     return u0
165
166
167 # Create the function to update the time-dependent BCs
168 def newBound(nnm,mindx,mindy,r,K,vol1,vol2,dt,s1y0,s2x0):
169     newBC = zeros(nnm,dtype=float)
170
171     # Call the vannila PUT function and use outer
172     # nodal values
173     s1bc = vanilla(r,K,dt,vol1,s1y0)
174     s2bc = vanilla(r,K,dt,vol2,s2x0)
175
```

```
176     # Set the values equal to the output from vanilla
177     # NOTE: I assume that the min of S1 and S2 are at the
178     # origin and that they are equal because they share the
179     # same strike. The rest are zeros
180     for i in range(len(s1bc)):
181         bnod = mindy[i]
182         newBC[bnod] = s1bc[i]
183 
184     for i in range(len(s2bc)):
185         bnod = mindx[i]
186         newBC[bnod] = s2bc[i]
187 
188     return newBC
189 
190 
191 # Get user input
192 #------------------------------------------------------------
193 app = femGUI.MyApp(False)
194 app.MainLoop()
195 inputs = femGUI.values
196 
197 # Multiply everything by 1.0 or 1 to ensure we have SciPy dtype
198 # floats or integers as the GUI passes UNICODE STRINGS!!!
199 etype  = float(inputs[0])*1.0
200 s1high = float(inputs[1])*1.0
201 s2high = float(inputs[2])*1.0
202 vol1   = float(inputs[3])*1.0
203 vol2   = float(inputs[4])*1.0
204 rate   = float(inputs[5])*1.0
205 pcorr  = float(inputs[6])*1.0
206 K      = float(inputs[7])*1.0
207 lastT  = float(inputs[8])*1.0
208 dt     = float(inputs[9])*1.0
209 nx     = int(inputs[10])*1
210 ny     = int(inputs[11])*1
211 
212 # Specify zero as the minimum value for the grid.
213 s1low  = 0.0
214 s2low  = 0.0
```

```
215
216 # Below for comparison purposes
217 # Comment out GUI inputs and run with the below values
218 #
219 # These values are the same used by ACHDOU & PIRONNEAU
220 # for the creation of Figures [4.11] and [4.12]
221 # These are the equivalent values for their THETA matrix
222 # using this formulation
223 ## etype  = 1.0
224 ## s1high = 150.0
225 ## s2high = 150.0
226 ## vol1   = 0.1414
227 ## vol2   = 0.1414
228 ## rate   = 0.1
229 ## pcorr  = -0.6
230 ## K      = 100.0
231 ## lastT  = 0.70
232 ## dt     = 0.01
233 ## nx     = 50
234 ## ny     = 50
235
236 # Initialize vectors/matrices
237 # Integer values for loops/sizes
238 nex1 = nx + 1
239 ney1 = ny + 1
240 nem  = 2*nx*ny
241 nnm  = nex1*ney1
242 npe  = 3
243 ndf  = 1
244 neq  = nnm*ndf
245 nn   = npe*ndf
246
247 # Number of quadrature points
248 nipf = 3
249
250 # Floats and arrays
251 xo = s1low
252 yo = s2low
253 dx = ones(nex1,float)*float((s1high/nx))
```

```
254 dy = ones(ney1,float)*float((s2high/ny))
255 dx[-1] = 0.0
256 dy[-1] = 0.0
257
258 # Create the differential eqn's coefficients
259 fo   = 0.0
260 co   = 1.0
261 a110 = 0.5*(vol1**2.0)
262 a220 = 0.5*(vol2**2.0)
263 a120 = pcorr*vol1*vol2
264 b10  = rate
265 b20  = rate
266 G    = -1.0*rate
267
268 # Call Fortran Mesh routine
269 # NOTA BENE: The connectivity matirx NODF has indices
270 #            according to the FORTRAN CONVENTION!
271 nodf,glxy = tri.mesh(nx,ny,nex1,ney1,nem,nnm,dx,dy,xo,yo)
272
273 # Switch NODF indices for the Python convention
274 fort2py   = ones(shape(nodf),dtype=int)
275 nodp      = nodf - fort2py
276
277 # Find IdaigF and Idiag where they are the Fortran and Python
278 # index of the diagonal for the non-symmetric stiffness matrix
279 # respectively -> RECALL: Python starts indexing at 0!
280 IdiagF = 0
281 for i in range(nem):
282     for j in range(npe):
283         for k in range(npe):
284             nw = (int(abs(nodf[i,j] - nodf[i,k])+1))*ndf
285             if IdiagF < nw:
286                 IdiagF = nw
287
288 # Band width of sparse matrix
289 band  = (IdiagF*2) - 1
290 Idiag = IdiagF - 1
291
292
```

```
293 #-------------------------------------------------------------#
294 #                                                             #
295 #                      Begin FEM Routine                      #
296 #                                                             #
297 #-------------------------------------------------------------#
298
299
300 # [1] Set time values
301 # Time dependent variables & Crank-Nicolson parameters
302 alfa    = 0.5
303 ntime   = int(lastT/dt) + 1
304 a1      = alfa*dt
305 a2      = (1.0 - alfa)*dt
306
307 # Create storage matrices for values at each time step
308 optionValue = zeros((nnm,ntime),dtype=float)
309 optionParam = zeros((nnm,ntime),dtype=float)
310
311 # [2] Initialize BCs
312
313 # Create "final" condition and store for option price calculation
314 # once all the values in time have been calculated
315 uo    = initialVal(K,glxy,nnm,etype)
316 glu   = uo
317
318 # Generate boundary information matrices from global matrix
319 bnode,mindx,maxdx,mindy,maxdy,s1yo,s2xo = \
320                                     findbc(glxy,s1high,s2high,nnm)
321
322 # An array of Python indices
323 nwld = arange(nnm,dtype=int)
324
325 # [3] Enter time loop
326 time    = 0.0
327 ncount = 0
328 while ncount < ntime :
329
330     # Find new BCs for future time step
331     time += dt
```

```
332     newBC = \
333       newBound(nnm,mindx,mindy,rate,K,vol1,vol2,time,s1yo,s2xo)
334 
335     # Global matrices
336     glk  = zeros((neq,band),dtype=float)
337     glf  = zeros(neq,dtype=float)
338 
339     # Begin loop over each element
340     for n in range(nem):
341         # Element matrices
342         elxy = zeros((npe,2),dtype=float)
343         elu  = zeros(npe,dtype=float)
344         elf  = zeros(npe,dtype=float)
345         elm  = zeros((npe,npe),dtype=float)
346         elk  = zeros((npe,npe),dtype=float)
347 
348         for i in range(npe):
349             # Assign global values for each node in the element
350             ni = nodp[n,i]
351             elxy[i,0] = glxy[ni,0]
352             elxy[i,1] = glxy[ni,1]
353             elu[i]    = glu[ni]
354 
355         # [4] Now compute elemental matrices
356         # Load quadrature data from Fortran Module
357         l1,l2,l3,lwt = GQD.quad()
358 
359         # [5] Begin quadtrature loop
360         for nl in range(npe):
361             ac1 = l1[nl]
362             ac2 = l2[nl]
363             ac3 = l3[nl]
364 
365             # Call Fortran Shape Function Module
366             det,sf,gdsf = SFN.sfntri(ac1,ac2,ac3,elxy)
367             cnst = 0.5*det*lwt[nl]
368 
369             # Global x an y in terms of the unit triangle
370             x = 0.0
```

```
371             y = 0.0
372             for it in range(npe):
373                 x += elxy[it,0]*sf[it]
374                 y += elxy[it,1]*sf[it]
375
376             # Set coefficients with mapped x and y coordinates
377             a11 = a110*x*x
378             a22 = a220*y*y
379             a12 = a120*x*y
380             b1  = b10*x
381             b2 = b20*y
382             source = fo
383             ct     = co
384
385             # Create Elemental K, M, and F matrices/vector
386             # by integrating over the element
387             for ip in range(npe):
388                 for jp in range(npe):
389                     soo = sf[ip]*sf[jp]*cnst
390                     s11 = gdsf[0,ip]*gdsf[0,jp]*cnst
391                     s22 = gdsf[1,ip]*gdsf[1,jp]*cnst
392                     s12 = gdsf[0,ip]*gdsf[1,jp]*cnst
393                     so1 = sf[ip]*gdsf[0,jp]*cnst
394                     so2 = sf[ip]*gdsf[1,jp]*cnst
395                     # Now assemble ELEMENT MATRIX [K]
396                     # using the form from THOMPSON
397                     # [K]  = [S1] - [S2] - [S3] - [Sh] where
398                     # [Sh] = 0.0 for this problem
399                     elk[ip,jp] += (a11*s11 + a12*s12 + a22*s22)\
400                                   - (b1*so1 + b2*so2) \
401                                   - G*soo
402                     elm[ip,jp] += ct*soo
403                 elf[ip] += cnst*sf[ip]*source
404
405         # [6] Apply CRANK-NICOLSON to find K^ and F^
406         # See J.N. REDDY, eqn (6.42b)
407         for ik in range(nn):
408             summ = 0.0
409             for jk in range(nn):
```

```
410                 summ += (elm[ik,jk] - a2*elk[ik,jk])*elu[jk]
411                 elk[ik,jk] = elm[ik,jk] + a1*elk[ik,jk]
412             elf[ik] = (a1+a2)*elf[ik] + summ
413
414         # [7] Assemble into global matrices using the
415         # routine from THOMPSON for banded & non-symmetric
416         for j in range(npe):
417             jnp = nodp[n,j]
418             jeq = nwld[jnp]
419             glf[jeq] += elf[j]
420             for k in range(npe):
421                 knp = nodp[n,k]
422                 keq = nwld[knp]
423                 kb  = (keq-jeq) + Idiag
424                 glk[jeq,kb] += elk[j,k]
425
426     # [8] Apply BCs by BLASTING technique (also a THOMPSON thing)
427     BLAST = 1.0e6
428     for i in range(nnm):
429         if allclose(bnode[i],1):
430             nb = nwld[i]
431             glu[nb] = newBC[i]
432             glk[nb,Idiag] *= BLAST
433             glf[nb] = glu[i]*glk[nb,Idiag]
434
435     # [9] Solve GLOBAL MATRICES using Fortran nsymgauss module
436     glu = NSG.nsymgauss(glk,glf,neq,band)
437
438     # [10] Store data for visualization at the end
439     oValu = glu
440     oPara = glu - uo
441     for i in range(nnm):
442         optionValue[i,ncount] = oValu[i]
443         optionParam[i,ncount] = oPara[i]
444
445     # [11] Update the time loop and BCs for next time step
446     ncount += 1
447
448 # END OF TIME LOOP HERE ----------------------------------
```

```
449
450
451 # Visualize with MayaVi
452 #-------------------------------------------------------------
453
454 # Set the z1 variable (second column is time)
455 ##z1 = u0
456 z1 = optionValue[:,-1]
457 ## z1 = optionParam[:,-1]
458
459 import pyvtk
460 # Scale the data in the Z-direction
461 dzz = dx[0]*2
462 dxx = dx[0]
463 dyy = dy[0]
464
465 # Convert z1 to vtk structured point data
466 # Note: No need to rearrange z1 as it is already in the
467 #       proper sequence from the meshing routine
468 point_data = pyvtk.PointData(pyvtk.Scalars(z1))
469
470 # Generate the grid sizing
471 grid=pyvtk.StructuredPoints((nex1,nex1,1),(0,0,0),(dxx,dyy,dzz))
472
473 # Save to temporary file
474 data = pyvtk.VtkData(grid, point_data)
475 data.tofile('/tmp/test.vtk')
476
477 # Now use MayaVi to visualize
478 import mayavi
479 v = mayavi.mayavi() # create a MayaVi window.
480 d = v.open_vtk('/tmp/test.vtk', config=0) # open the data file.
481
482 # Load the filters.
483 f = v.load_filter('WarpScalar', config=0)
484 n = v.load_filter('PolyDataNormals', 0)
485 n.fil.SetFeatureAngle (45)
486
487 # Load the necessary modules.
```

```
488 m = v.load_module('SurfaceMap', 0)
489 a = v.load_module('Axes', 0)
490 t = v.load_module('Text',0)
491 o = v.load_module('Outline', 0)
492
493 # Re-render the scene.
494 v.Render()
495 v.master.wait_window()
496
497
498 # Or visualize with Matplotlib
499 #---------------------------------------------------------------
500
501 # An alternative option (for speed) is matplotlib
502 # Output data to figures using matplotlib below
503 ## import pylab as p
504 ## import matplotlib.axes3d as p3
505 ## x  = reshape(glxy[:,0],(nex1,ney1))
506 ## y  = reshape(glxy[:,1],(nex1,ney1))
507 ## init_val = u0
508 ## finalval = optionValue[:,-1]
509 ## time_val = optionParam[:,-1]
510 ## z1 = reshape(init_val,(nex1,ney1))
511 ## z2 = reshape(finalval,(nex1,ney1))
512 ## z3 = reshape(time_val,(nex1,ney1))
513
514 ## # Make three figures
515 ## fig1= p.figure(1)
516 ## ax1 = p3.Axes3D(fig1)
517 ## ax1.plot_wireframe(x,y,z1)
518 ## ax1.set_xlabel('S1')
519 ## ax1.set_ylabel('S2')
520 ## ax1.set_zlabel('Final Condition at Expiry')
521
522 ## fig2= p.figure(2)
523 ## ax2 = p3.Axes3D(fig2)
524 ## ax2.plot_wireframe(x,y,z2)
525 ## ax2.set_xlabel('S1')
526 ## 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
```

```
21 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣2␣for␣(K-max(S1,S2)+␣",
22                                     size=(220,-1))
23         self.etype = wx.TextCtrl(panel, -1, "", size=(100,-1));
24
25
26         # S1 and S2 upper bounds for grid
27         s2label = wx.StaticText(panel, -1, "S1␣High,␣S2␣High:␣",\
28                                 size=(220,-1))
29         self.s1upper = wx.TextCtrl(panel, -1, "", size=(100,-1));
30         self.s2upper = wx.TextCtrl(panel, -1, "", size=(100,-1));
31
32         # S1 and S2 volatility
33         vlabel = wx.StaticText(panel, -1, "S1␣Volatility,
34 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣S2␣Volatility:␣", size=(220,-1))
35         self.v1vol = wx.TextCtrl(panel, -1, "", size=(100,-1));
36         self.v2vol = wx.TextCtrl(panel, -1, "", size=(100,-1));
37
38         # Risk free rate and correlation
39         prlabel = wx.StaticText(panel, -1, "Interest␣Rate,
40 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Correlation:␣", size=(220,-1))
41         self.risk = wx.TextCtrl(panel, -1, "", size=(100,-1));
42         self.corr = wx.TextCtrl(panel, -1, "", size=(100,-1));
43
44
45         # Strike and Exercise Date
46         kTlabel = wx.StaticText(panel, -1, "Srike␣Price,
47 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Exercise␣Date:␣", size=(220,-1))
48         self.strike = wx.TextCtrl(panel, -1, "", size=(100,-1));
49         self.finalT = wx.TextCtrl(panel, -1, "", size=(100,-1));
50
51         # deltaT and deltaX
52         dTXlabel = wx.StaticText(panel, -1, "delta␣T,␣NX,␣NY:␣",\
53                                 size=(220,-1))
54         self.deltaT = wx.TextCtrl(panel, -1, "", size=(100,-1));
55         self.nxval  = wx.TextCtrl(panel, -1, "", size=(100,-1));
56         self.nyval  = wx.TextCtrl(panel, -1, "", size=(100,-1));
57
58
59         # Execute program
```

```
60          runBtn = wx.Button(panel, -1, "Run")
61          self.Bind(wx.EVT_BUTTON, self.OnSubmit, runBtn)
62
63          # Now do the layout.
64
65          # mainSizer is the top-level one that manages everything
66          mainSizer = wx.BoxSizer(wx.VERTICAL)
67          mainSizer.Add(topLbl, 0, wx.ALL, 5)
68          mainSizer.Add(wx.StaticLine(panel), 0,
69                  wx.EXPAND|wx.TOP|wx.BOTTOM, 5)
70
71          # femSizer is a grid that holds all of the address info
72          femSizer = wx.FlexGridSizer(cols=2, hgap=5, vgap=5)
73          femSizer.AddGrowableCol(1)
74
75          # Expiry Type
76          femSizer.Add(sclabel, 0,
77                  wx.ALIGN_RIGHT|wx.ALIGN_CENTER_VERTICAL)
78          # the lower and upper bounds are in a sub-sizer
79          etSizer = wx.BoxSizer(wx.HORIZONTAL)
80          etSizer.Add(self.etype, 1)
81          femSizer.Add(etSizer, 1, wx.EXPAND)
82
83
84          # S1 and S2 HIGH label
85          femSizer.Add(s2label, 0,
86                  wx.ALIGN_RIGHT|wx.ALIGN_CENTER_VERTICAL)
87          # the lower and upper bounds are in a sub-sizer
88          s2Sizer = wx.BoxSizer(wx.HORIZONTAL)
89          s2Sizer.Add(self.s1upper, 1)
90          s2Sizer.Add((10,10)) # some empty space
91          s2Sizer.Add(self.s2upper, 1, wx.LEFT|wx.RIGHT, 5)
92          femSizer.Add(s2Sizer, 1, wx.EXPAND)
93
94
95          # Volatility label
96          femSizer.Add(vlabel, 0,
97                  wx.ALIGN_RIGHT|wx.ALIGN_CENTER_VERTICAL)
98          # the lower and upper bounds are in a sub-sizer
```

```
99          volSizer = wx.BoxSizer(wx.HORIZONTAL)
100         volSizer.Add(self.v1vol, 1)
101         volSizer.Add((10,10)) # some empty space
102         volSizer.Add(self.v2vol, 1, wx.LEFT|wx.RIGHT, 5)
103         femSizer.Add(volSizer, 1, wx.EXPAND)
104
105
106         # Risk free Rate and corelation
107         femSizer.Add(prlabel, 0,
108                 wx.ALIGN_RIGHT|wx.ALIGN_CENTER_VERTICAL)
109         # the lower and upper bounds are in a sub-sizer
110         rcSizer = wx.BoxSizer(wx.HORIZONTAL)
111         rcSizer.Add(self.risk, 1)
112         rcSizer.Add((10,10)) # some empty space
113         rcSizer.Add(self.corr, 1, wx.LEFT|wx.RIGHT, 5)
114         femSizer.Add(rcSizer, 1, wx.EXPAND)
115
116
117         # Strike and Exercise Date
118         femSizer.Add(kTlabel, 0,
119                 wx.ALIGN_RIGHT|wx.ALIGN_CENTER_VERTICAL)
120         # the lower and upper bounds are in a sub-sizer
121         ktSizer = wx.BoxSizer(wx.HORIZONTAL)
122         ktSizer.Add(self.strike, 1)
123         ktSizer.Add((10,10)) # some empty space
124         ktSizer.Add(self.finalT, 1, wx.LEFT|wx.RIGHT, 5)
125         femSizer.Add(ktSizer, 1, wx.EXPAND)
126
127
128         # deltaT and deltaX
129         femSizer.Add(dTXlabel, 0,
130                 wx.ALIGN_RIGHT|wx.ALIGN_CENTER_VERTICAL)
131         # the lower and upper bounds are in a sub-sizer
132         dtxSizer = wx.BoxSizer(wx.HORIZONTAL)
133         dtxSizer.Add(self.deltaT, 1)
134         dtxSizer.Add((10,10)) # some empty space
135         dtxSizer.Add(self.nxval, 1, wx.LEFT|wx.RIGHT, 5)
136         dtxSizer.Add((10,10)) # some empty space
137         dtxSizer.Add(self.nyval, 1, wx.LEFT|wx.RIGHT, 5)
```

```
138         femSizer.Add(dtxSizer, 1, wx.EXPAND)
139
140
141         # now add the femSizer to the mainSizer
142         mainSizer.Add(femSizer, 0, wx.EXPAND|wx.ALL, 10)
143
144         # gaps between and on either side of the buttons
145         btnSizer = wx.BoxSizer(wx.HORIZONTAL)
146         btnSizer.Add((10,10)) # some empty space
147         btnSizer.Add(runBtn)
148         btnSizer.Add((10,10)) # some empty space
149         mainSizer.Add(btnSizer, 0, wx.EXPAND|wx.BOTTOM, 10)
150
151         panel.SetSizer(mainSizer)
152
153         # Fit the frame to the needs of the sizer.  The frame
154         # will automatically resize the panel as needed.
155         # Also prevent the frame from getting smaller than
156         # this size.
157         mainSizer.Fit(self)
158         mainSizer.SetSizeHints(self)
159
160     def OnSubmit(self, evt):
161         # Allow the inputs to be viewed by the calling program
162         global values
163         values = (self.etype.GetValue(),
164                   self.s1upper.GetValue(),
165                   self.s2upper.GetValue(),
166                   self.v1vol.GetValue(),
167                   self.v2vol.GetValue(),
168                   self.risk.GetValue(),
169                   self.corr.GetValue(),
170                   self.strike.GetValue(),
171                   self.finalT.GetValue(),
172                   self.deltaT.GetValue(),
173                   self.nxval.GetValue(),
174                   self.nyval.GetValue())
175         self.Close(True)
176
```

```
177 class MyApp(wx.App):
178
179     def OnInit(self):
180         frame = FemInput()
181         self.SetTopWindow(frame)
182         frame.Show()
183         return True
184
185
186 # Needed if called as a module
187 if __name__ == '__main__':
188     app = MyApp(False)
189     app.MainLoop()
```

Appendix B

Fortran Code Listings

B.1 Fortran Mesh Routine

```
1  ! This is the 2D mesh generator for linear, triangular elements
2  !
3  ! This code is modified from J.N. REDDY, AN INTRODUCTION TO THE
4  ! FINITE ELEMENT METHOD, 2nd EDITION
5  !
6  ! Coded by Tyler Hayes, April 3, 2007
7  !
8  !
9  ! INPUTS:
10 !
11 !       NX = NUMBER OF DIVISIONS IN THE X-DIR
12 !       NY = NUMBER OF DIVISIONS IN THE Y-DIR
13 !     NEX1 = INTEGER = NX+1
14 !     NEY1 = INTEGER = NY+1
15 !      NEM = INTEGER = 2*NX*NY
16 !      NNM = INTEGER = ((IEL*NX)+1) * ((IEL*NY)+1) IEL = 1 for tri
17 !       DX = VECTOR OF SPACINGS IN THE X-DIR <- CAN BE VARIABLE
18 !            THE VECTOR SHOULD BE NX+1 IN SIZE BUT THE DX(NX+1) IS
19 !            SET TO ZERO AND IS A DUMMY VARIABLE
20 !       DY = VECTOR OF SPACINGS IN THE Y-DIR <- CAN BE VARIABLE
21 !            THE VECTOR SHOULD BE NY+1 IN SIZE BUT THE DY(NY+1) IS
22 !            SET TO ZERO AND IS A DUMMY VARIABLE
```

```
23 !       X0 = ORIGIN OF THE X AXIS
24 !       Y0 = ORIGIN OF THE Y AXIS
25 !
26 ! OUTPUT:
27 !       NOD = INTEGER MATRIX OF ELEMENT NOD INDICES
28 !      GLXY = GLOBAL COORDINATES OF NOD
29 !
30 ! MISC.:
31 !       NPE = NODES PER ELEMENT
32 !-----------------------------------------------------------------------
33
34 SUBROUTINE MESH(NOD,GLXY,NX,NY,NEX1,NEY1,NEM,NNM,DX,DY,Xo,Yo)
35    IMPLICIT NONE
36    INTEGER :: NXX,NYY,NX2,NY2,NXX1,NYY1
37    INTEGER :: K,IY,L,M,N,I,NI,NJ,IEL
38    INTEGER,PARAMETER :: NPE=3
39    REAL :: XC,YC
40 ! INPUTS
41    INTEGER,INTENT(IN) :: NX,NY,NEM,NNM,NEX1,NEY1
42    REAL,INTENT(IN) :: Xo,Yo
43    REAL,DIMENSION(NEX1),INTENT(IN) :: DX
44    REAL,DIMENSION(NEY1),INTENT(IN) :: DY
45 ! PARAMETERS
46 ! OUTPUTS
47    INTEGER,DIMENSION(NEM,NPE),INTENT(OUT) :: NOD
48    REAL,DIMENSION(NNM,2),INTENT(OUT) :: GLXY
49
50 ! CREATE VARIABLES
51    IEL  = 1 ! FOR <= 4 NODES PER ELEMENT
52    NXX  = IEL*NX
53    NYY  = IEL*NY
54    NXX1 = NXX + 1
55    NYY1 = NYY + 1
56    NX2  = 2*NX
57    NY2  = 2*NY
58
59
60 ! CREATE TRIANGUALR ELEMENTS
61 !-----------------------------------------------------------------------
```

```
62
63  ! INITIALIZE FIRST TWO ELEMENTS
64    NOD(1,1) = 1
65    NOD(1,2) = IEL+1
66    NOD(1,3) = IEL*NXX1 + IEL + 1
67    NOD(2,1) = 1
68    NOD(2,2) = NOD(1,3)
69    NOD(2,3) = IEL*NXX1 + 1
70
71  ! LOOP THROUGH MESH
72    K=3
73    DO IY=1,NY
74       L=IY*NX2
75       M=(IY-1)*NX2
76       IF (NX > 1) THEN
77          DO N=K,L,2
78             DO I=1,NPE
79                NOD(N,I)   = NOD(N-2,I) + IEL
80                NOD(N+1,I) = NOD(N-1,I) + IEL
81             END DO
82          END DO
83       END IF
84       IF (IY < NY) THEN
85          DO I=1,NPE
86             NOD(L+1,I) = NOD(M+1,I) + IEL*NXX1
87             NOD(L+2,I) = NOD(M+2,I) + IEL*NXX1
88          END DO
89       END IF
90       K=L+3
91    END DO
92
93
94  ! NOW GENERATE GLOBAL COORDINATES OF THE NODES
95    XC = Xo
96    YC = Yo
97
98    DO NI=1,NEY1
99       XC = Xo
100      I  = NXX1*IEL*(NI-1)
```

```
101       DO NJ = 1,NEX1
102          I=I+1
103          GLXY(I,1) = XC
104          GLXY(I,2) = YC
105          IF (NJ < NEX1) THEN
106             IF (IEL == 2) THEN
107                I=I+1
108                XC = XC + 0.5*DX(NJ)
109                GLXY(I,1) = XC
110                GLXY(I,2) = YC
111             END IF
112          END IF
113          XC = XC + DX(NJ)/IEL
114       END DO
115       XC = Xo
116       IF (IEL == 2) THEN
117          YC = YC + 0.5*DY(NI)
118          DO NJ = 1,NEX1
119             I=I+1
120             GLXY(I,1) = XC
121             GLXY(I,2) = YC
122             IF (NJ < NEX1) THEN
123                I=I+1
124                XC = XC + 0.5*DX(NJ)
125                GLXY(I,1) = XC
126                GLXY(I,2) = YC
127             END IF
128             XC = XC + 0.5*DX(NJ)
129          END DO
130       END IF
131       YC = YC + DY(NI)/IEL
132    END DO
133 END SUBROUTINE MESH
```

B.2 Fortran Shape Function Routine

```
1 SUBROUTINE SFNTRI(DET,SF,GDSF,L1,L2,L3,ELXY)
```

```
2    IMPLICIT NONE
3  ! THIS IS THE SUBROUTINE THAT CALCULATES THE SHAPE FUNCTIONS
4  ! AND THEIR DERIVATIVES AT SPECIFIED GLOBAL POSITIONS
5    INTEGER,PARAMETER :: NPE=3
6    INTEGER :: I,J,K
7    REAL,DIMENSION(3,2),INTENT(IN) :: ELXY
8    REAL,INTENT(IN) :: L1,L2,L3
9    REAL,INTENT(OUT) :: DET
10   REAL,DIMENSION(3),INTENT(OUT) :: SF
11   REAL,DIMENSION(2,3),INTENT(OUT) :: GDSF
12   REAL,DIMENSION(3,3) :: DSF
13   REAL,DIMENSION(2,2) :: GJ,GJINV
14   REAL :: SUM
15
16 ! INITIALIZE ARRAYS
17   DO I=1,NPE
18       DSF(1,I) = 0.0
19       DSF(2,I) = 0.0
20       DSF(3,I) = 0.0
21       SF(I)    = 0.0
22   END DO
23
24 ! SET THE SHAPE FUNCTIONS FOR A TRIANGLE
25   SF(1) = L1
26   SF(2) = L2
27   SF(3) = L3
28   DSF(1,1) = 1.0
29   DSF(2,2) = 1.0
30   DSF(3,3) = 1.0
31
32 ! FIND THE JACOBIAN
33   DO I=1,2
34       DO J=1,2
35           SUM = 0.0
36           DO K=1,NPE
37               SUM = SUM + (DSF(I,K) - DSF(3,K))*ELXY(K,J)
38           END DO
39           GJ(I,J) = SUM
40       END DO
```

```
41    END DO
42
43  ! FIND GJ's INVERSE
44    DET          =   GJ(1,1)*GJ(2,2) - GJ(1,2)*GJ(2,1)
45    GJINV(1,1) =   GJ(2,2)/DET
46    GJINV(2,2) =   GJ(1,1)/DET
47    GJINV(1,2) =  -1.0*GJ(1,2)/DET
48    GJINV(2,1) =  -1.0*GJ(2,1)/DET
49
50  ! OBTAIN THE DERIVATIVE OF THE SHAPE FUNCTION WITH PROPER
51  ! TRANSFORMATION
52    DO I=1,2
53        DO J=1,NPE
54            SUM = 0.0
55            DO K=1,2
56                SUM = SUM + GJINV(I,K)*(DSF(K,J) - DSF(3,J))
57            END DO
58            GDSF(I,J) = SUM
59        END DO
60    END DO
61
62  END SUBROUTINE SFNTRI
```

B.3 Fortran Non-symmetric Gaussian Solver

```
1   ! This is a subroutine that finds y from A.y = F for
2   ! banded NON-symmetric matrices. From Thompson text
3   !
4   ! INPUTS:
5   !         A   = non-symmetric banded matrix
6   !         F   = vector from A.Y = F
7   !         IB  = bandwidth of matrix (columns)
8   !         neq= number of rows in the matrix
9   !
10  ! OUTPUTS:
11  !         Y   = solution vector from A.Y = F
12  !-----------------------------------------------------------------
```

```
13 SUBROUTINE nsymgauss(Y,A,F,neq,IB)
14   IMPLICIT NONE
15   INTEGER :: Idiag,i,j,k,Jend,Kbgn,Kend,Ikc,Jkc,Iback,Jc,Kc
16   REAL :: FAC
17   INTEGER, INTENT(IN) :: neq,IB
18   REAL,DIMENSION(neq,IB) :: AA
19   REAL,DIMENSION(neq) :: FF
20   REAL,DIMENSION(neq,IB),INTENT(IN) :: A
21   REAL,DIMENSION(neq),INTENT(IN) :: F
22   REAL,DIMENSION(neq),INTENT(OUT) :: Y
23
24 ! Initialize matrices to overwrite
25   DO i = 1,neq
26     DO j = 1,IB
27       AA(i,j) = A(i,j)
28     END DO
29     FF(i) = F(i)
30   END DO
31
32 ! Forward elimination
33   Idiag = ((IB - 1)/2) + 1
34   DO i = 1,neq-1
35     Jend = neq
36     IF (Jend > (i+Idiag-1)) Jend = i+Idiag-1
37     DO j = i+1,Jend
38       Kc   = Idiag - (j-i)
39       FAC  = -AA(j,Kc)/AA(i,Idiag)
40       Kbgn = i
41       Kend = Jend
42       DO k = Kbgn,Kend
43         Ikc = Idiag + (k-i)
44         Jkc = Idiag + (k-j)
45         AA(j,Jkc) = AA(j,Jkc) + FAC*AA(i,Ikc)
46       END DO
47       FF(j) = FF(j) + FAC*FF(i)
48     END DO
49   END DO
50
51 ! Backward substitution
```

```
52    Y(neq) = FF(neq)/AA(neq,Idiag)
53    DO Iback = 2,neq
54       i     = neq - Iback + 1
55       Jend = neq
56       IF (Jend > (i+(Idiag-1))) Jend = i+(Idiag-1)
57       DO j = i+1,Jend
58          Jc     = Idiag + (j-i)
59          FF(i) = FF(i) - AA(i,Jc)*Y(j)
60       END DO
61       Y(i) = FF(i)/AA(i,Idiag)
62    END DO
63 END SUBROUTINE nsymgauss
```

B.4 Fortran Quadrature Data Function

```
1  SUBROUTINE QUAD(L1,L2,L3,LWT)
2    IMPLICIT NONE
3  ! THIS CREATES THE TABLE OF VALUES FOR TRIANGULAR ELEMENTS
4  ! USED FOR THREE-POINT QUADRATURE
5  !
6  !  INTEGRATION REGION:
7  !
8  !       0 <= X,   AND   0 <= Y,   AND    X + Y <= 1.
9  !
10 !  GRAPH:
11 !
12 !       ^
13 !    1  | *
14 !       | |\
15 !    Y  | | \
16 !       | |  \
17 !    0  | *---*
18 !       +------->
19 !         0 X 1
20 !
21 !
22 !
```

```
23 INTEGER :: I
24 REAL,DIMENSION(3),INTENT(OUT) :: L1,L2,L3,LWT
25
26 ! INITIALIZE
27 DO I=1,3
28    L1(I)  = 0.0
29    L2(I)  = 0.0
30    L3(I)  = 0.0
31    LWT(I) = 0.0
32 END DO
33
34
35 ! THREE-POINT QUADRATURE
36 L1(1)  = 0.0
37 L1(2)  = 0.5
38 L1(3)  = 0.5
39 L2(1)  = 0.5
40 L2(2)  = 0.0
41 L2(3)  = 0.5
42 L3(1)  = 0.5
43 L3(2)  = 0.5
44 L3(3)  = 0.0
45 LWT(1) = 1.0/3.0
46 LWT(2) = 1.0/3.0
47 LWT(3) = 1.0/3.0
48 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,xo,yo) ! 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
```

```
18              real dimension(ney1),intent(in),depend(ney1) :: dy
19              real intent(in) :: xo
20              real intent(in) :: yo
21          end subroutine mesh
22      end interface
23  end python module tri
24
25  ! This file was auto-generated with f2py (version:2_3396).
26  ! See http://cens.ioc.ee/projects/f2py2e/
```

C.2 Shape Function Interface File

```
1   !    -*- f90 -*-
2   ! Note: the context of this file is case sensitive.
3
4   python module shpfn ! in
5       interface  ! in :shpfn
6           subroutine sfntri(det,sf,gdsf,l1,l2,l3,elxy)
7   ! in :shpfn:sfntri.f90
8               real intent(out) :: det
9               real dimension(3),intent(out) :: sf
10              real dimension(2,3),intent(out) :: gdsf
11              real intent(in) :: l1
12              real intent(in) :: l2
13              real intent(in) :: l3
14              real dimension(3,2),intent(in) :: elxy
15          end subroutine sfntri
16      end interface
17  end python module shpfn
18
19  ! This file was auto-generated with f2py (version:2_3396).
20  ! 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
```

```
14
15 ! This file was auto-generated with f2py (version:2_3396).
16 ! 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, $\varphi = 1.618\ldots$.