Spatial Error Estimation for Collocation Solutions of Differential Equations

By

Andrew Fraser

April 26, 2020, Halifax, Nova Scotia

Approved: Dr. Paul Muir

Supervisor

Approved: Dr. Walt Finden

Reader

Date: April 26, 2020

Abstract

Computational Science is now a central component of all scientific investigation, along with the traditional modes of experimental and theoretical investigation. Computational Science involves the development and solution of mathematical models, i.e., systems of equations, that represent approximations to real world phenomenon in a wide variety of scientific areas. These mathematical models typically do not have closed form solutions and thus the models are solved using computational software to obtain approximate solutions. Since these solutions are approximate, the question that must be addressed is "How Good is the Computed Solution?". This question is answered for a given numerical solution through the computation of a good quality error estimate. This thesis will discuss current work on answering this question in the area of computational methods for differential equations that depend on time and/or one or more spatial dimensions. We will describe the use of collocation, a general numerical method that can be used to obtain approximate solutions for a wide range of problem classes, as well as our recent work in the development of efficiently computable error estimates for collocation solutions based on special types of interpolants. We provide results from numerical experiments to demonstrate the effectiveness of our approach.

Contents

List of Figures

List of Tables

Acknowledgments

The process leading to the creation of this thesis has had more outcomes than just this thesis. When this process begun I had heard of differential equations, but knew very little about them. In the span of four months I was able to become sufficiently familiar with differential equations to have developed numerical software for various types of these equations. For this, I would like to thank my supervisor, Dr. Paul Muir. From you I have also learned much more about the scientific community, and have had the opportunity to present at a conference, on the topics within this thesis. Most importantly, you have kept me on track through this lengthy process despite any excuses I would make.

Next, I would like to thank Connor Tannahill. Throughout the summer we worked together, you had also helped me understand the background material necessary for this thesis. Not only that, but your major role in the creation of the LOI2 is something that I can not understate.

I would also like to thank Daniel Doucett, Ben Jollymore, and Owen Sharpe. Throughout the final year of my undergraduate studies at SMU, we spent so much time in our McNally North room that it began to become our home away from home. This has made it much easier for me to get through this academic year.

Finally, I would like to thank my family, especially my parents, for the unending support throughout the past 22 years of my life. Without this, I don't think I would have ended up where I am today.

Chapter 1

Introduction

Differential equations (DEs) are often used for the mathematical modelling of real world phenomenon such as the spread of a brain tumor or the spread of a disease within a population. These differential equations generally do not have a closed form solution, and as such one can only calculate an approximate solution. Numerical methods are used in these situations to calculate an approximate solution. As these solutions are approximate there is an error associated with them, which gives rise to the question of what the error of a given approximation is. The true error of an approximation can only be calculated given that the true solution is known, in which case one would not require an approximate solution in the first place. However an error estimate can be calculated for an approximation of the error and this can give a good indication of the accuracy of the numerical solution. However, one can go further. Given a high quality error estimate, it is possible to implement adaptive methods. Adaptive methods allow for the calculation of an approximation that uses increased resolution in regions where the error estimate is large while not incurring unnecessary computational cost in regions where the error estimate is small. With error estimation and adaptive methods, one can then implement error control, where a tolerance can be specified and a solution approximation will be calculated such that the error estimate satisfies the specified tolerance.

One class of numerical methods for the numerical solution of differential equations is B-spline Gaussian Collocation. A numerical solution calculated using this method will be a smooth, continuous approximation across the problem domain. The general concept of B-spline Gaussian Collocation is as follows. The approximate solution is expressed in terms of a set of known B-spline basis functions with unknown coefficients. Then various points across the problem domain are chosen as collocation points. Finally the coefficients of the B-spline basis functions are determined by requiring that the approximate solution satisfy the DEs at the collocation points.

Once the approximate solution is obtained, the next step is to compute an error estimate for the approximate solution. One class of error estimates is based on the use of interpolants to the approximate solution that are of a different order of accuracy than the collocation solution. In this thesis, we investigate the use of interpolation-based error estimates for B-spline collocation solutions of boundary-value ordinary differential equations (BVODEs), 1D time-dependent partial differential equations (PDEs), 2D elliptic PDEs (EPDEs), and 2D time-dependent PDEs.

This thesis is organized as follows. Chapter 2 presents the different classes of DEs which will be considered within this thesis, as well as relevant background information for calculating a collocation solution for each of these classes, and for calculating an error estimate for the one spatial dimension problem classes. Chapter 3 presents a generalization of pre-existing 1D error estimation methods to two spatial dimensions, as well as a new error estimation method which can be used in one or two spatial dimensions. Chapter 4 provides an overview of the software created for this thesis. Chapter 5 contains the numerical results from various tests of the collocation and error estimation methods. Chapter 6 finishes with our conclusions, and suggests topics for potential future work.

Chapter 2

Background

In this chapter we will first be introducing the various problem classes considered in this thesis, as well as specific test problems from each of these problem classes. This chapter also contains an overview of previous work in the area of B-spline Gaussian collocation for ODEs and PDEs in one and two spatial dimensions, along with associated software. Finally this chapter will discuss error estimation and control, along with associated algorithms.

2.1 Problem Classes

Within this thesis we will be considering four different classes of problems. For each of these problem classes, there are two test problems that will be used to evaluate the performance of the numerical methods that we consider. The problem classes represent all combinations of time-dependent or time-independent problems, in one or two spatial dimensions.

2.1.1 Boundary Value Ordinary Differential Equations

BVODEs are the one spatial dimension, time-independent problem class. BVODEs can be used to model the value of a function between two known points, such as the temperature throughout an object with known temperatures at each end. While any order of solution derivative can appear in a BVODE, in this thesis we will only consider second-order BVODEs, meaning that the second derivative is the highest order solution derivative present in the DE. The general form for a BVODE that we consider in this thesis will include the ODE,

$$u_{xx}(x) = f(x, u(x), u_x(x)), \tag{2.1}$$

where $u(x)$ represents the true solution to the ODE, and f is some function of x, $u(x)$, and $u_x(x)$. The spatial domain is $A \leq x \leq B$, and the boundary conditions are $u(A) = \alpha$, $u(B) = \beta$. The test problems for this class are as follows.

BVODE Problem 1

The ODE given by,

$$u_{xx}(x) = 2u_x(x) - u(x) + xe^x - x, \quad 0 \leq x \leq 2, \tag{2.2}$$

with boundary conditions taken from the true solution:

$$u(x) = \frac{1}{6}x^3e^x - \frac{5}{3}xe^x + 2e^x - x - 2. \tag{2.3}$$

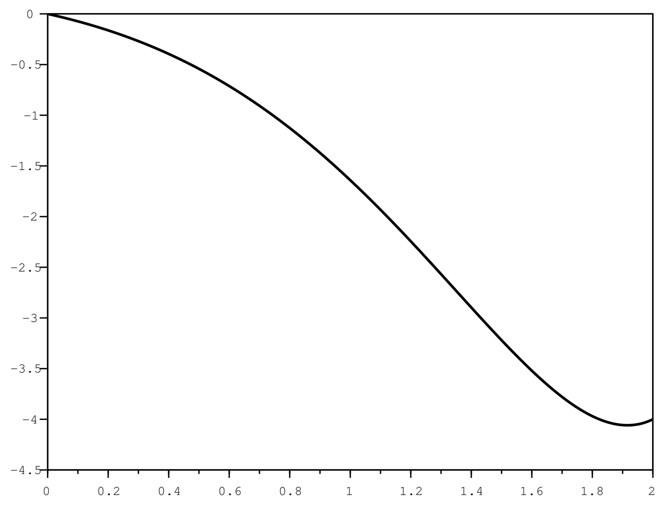


Figure 2.1: The true solution to BVODE problem 1.

BVODE Problem 2

The ODE given by,

$$u_{xx}(x) = e^{x}(x+2), \quad 0 \le x \le 2, \tag{2.4}$$

with boundary conditions taken from the true solution,

$$u(x) = x(e^{x} - e). \tag{2.5}$$

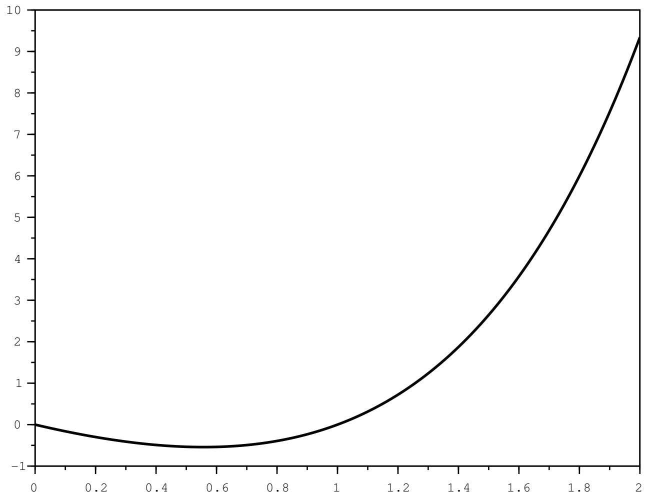


Figure 2.2: The true solution to BVODE problem 2.

2.1.2 Parabolic Partial Differential Equations

Parabolic PDEs (PPDEs) are the one spatial dimension, time-dependent problem class. PPDEs are commonly used to model the change of a function in time and across a spatial domain, such as the diffusion of heat in a cross-section of some material. While a PPDE can contain any order of spatial derivative, we will only be considering second-order spatial derivative PPDEs in this thesis. We will also only consider PDEs for which only the first temporal derivative of the solution is present. The general form we will consider is,

$$u_t(x,t) = f(x,t,u(x,t),u_x(x,t),u_{xx}(x,t))), \tag{2.6}$$

where $u(x,t)$ is the true solution of the PPDE. The spatial domain is $A \le x \le B$ and the temporal domain is $t_0 \le t \le t_f$. The initial condition is,

$$u(x,t_0) = \mu(x), \tag{2.7}$$

where $\mu(x)$ is some function which represents the beginning state of the solution across the spatial domain at the initial time. The boundary conditions are $u(A,t) = \alpha(t), u(B,t) = \beta(t)$.

The test problems for this class are as follows.

PPDE Problem 1

The PDE given by,

$$u_t(x,t) = \epsilon u_{xx}(x,t) - u(x,t)u_x(x,t), \tag{2.8}$$

where the problem dependent parameter, $\epsilon = \frac{1}{16}$. The spatial domain is $0 \le x \le 1$, and the temporal domain is $0 \le t \le 1$. The initial and boundary conditions are taken from the true solution,

$$u(x,t) = \frac{1}{2} - \frac{1}{2}\tanh\left(\frac{x - \frac{t}{2} - 0.25}{4\epsilon}\right). \tag{2.9}$$

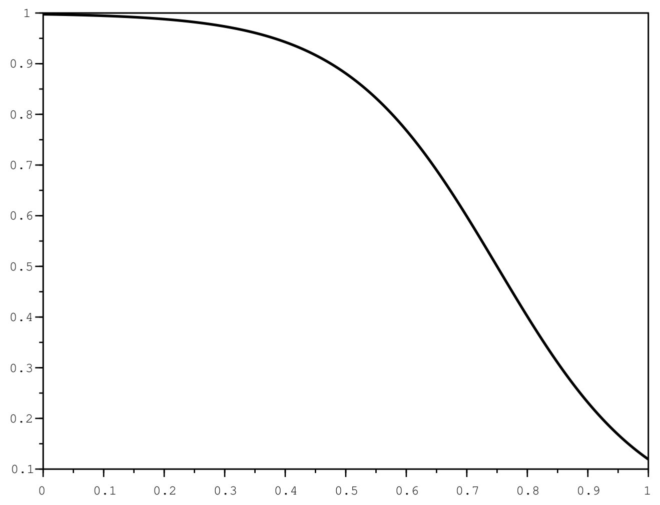


Figure 2.3: The true solution to PPDE problem 1 at $t = 1$.

PPDE Problem 2

The PDE given by,

$$u_t(x,t) = \frac{1}{16} u_{xx}(x,t), \tag{2.10}$$

The spatial domain is $0 \leq x \leq 1$, and the temporal domain is $0 \leq t \leq 1$. The initial condition is,

$$u(x,0) = 2\sin(2\pi x), \tag{2.11}$$

boundary conditions are $u(0,t) = u(1,t) = 0$, and the true solution is

$$u(x,t) = 2\sin(2\pi x)e^{-\frac{\pi^2}{4}t}. \tag{2.12}$$

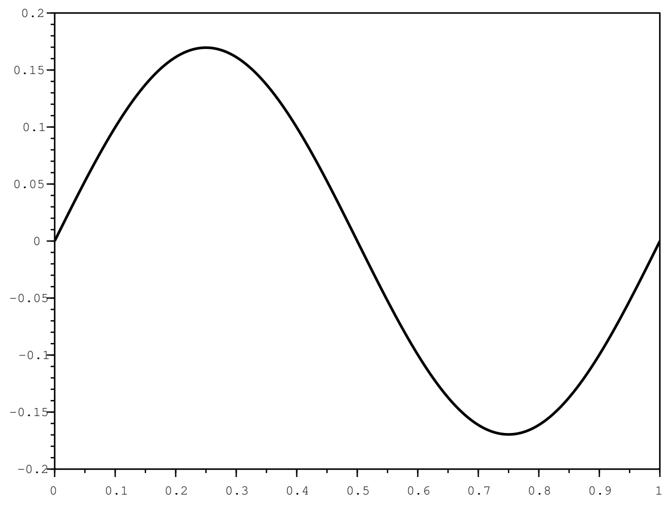


Figure 2.4: The true solution to PPDE problem 2 at $t = 1$.

2.1.3 Elliptic Partial Differential Equations

EPDEs are the two spatial dimension, time-independent problem class, and can be viewed as a generalization of the BVODE class to two spatial dimensions. As with the previous problem classes, we will only be considering second-order EPDEs. The general form we will consider in this thesis is,

$$u_{xx}(x, y) + u_{yy}(x, y) = f(x, y, u(x, y), u_x(x, y), u_y(x, y)), \tag{2.13}$$

where $u(x, y)$ is the true solution to the EPDE. The spatial domain is $A \leq x \leq B$, $C \leq y \leq D$. The boundary conditions are $u(A, y) = \alpha(y)$, $u(B, y) = \beta(y)$, $u(x, C) = \zeta(x)$, $u(x, D) = \eta(x)$. Note that, for consistency, we must have $u(A, C) = \alpha(C) = \zeta(A)$, $u(B, C) = \beta(C) = \zeta(B)$, $u(A, D) = \alpha(D) = \eta(A)$, and $u(B, D) = \beta(D) = \eta(B)$.

EPDE Problem 1

The PDE given by,

$$u_{xx}(x, y) + u_{yy}(x, y) = xe^y. \tag{2.14}$$

The spatial domain is $0 \le x \le 2$, $0 \le y \le 1$, and boundary conditions taken from the true solution,

$$u(x, y) = xe^{y}. \tag{2.15}$$

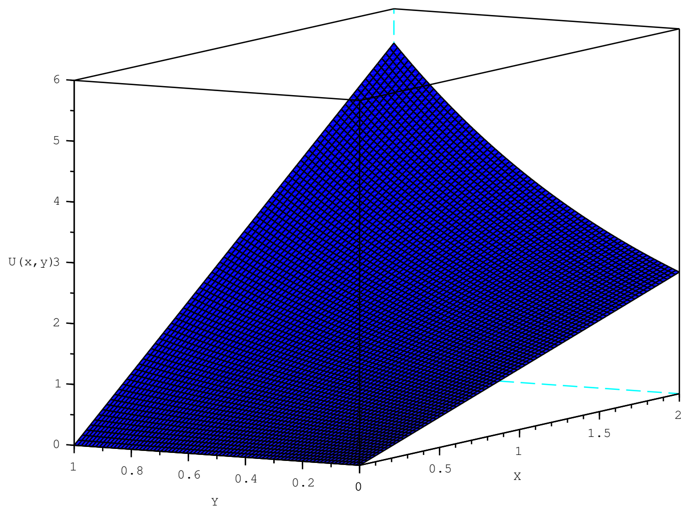


Figure 2.5: The true solution to EPDE problem 1.

EPDE Problem 2

The PDE given by,

$$u_{xx}(x, y) + u_{yy}(x, y) = (x^2 + y^2)e^{xy}. \tag{2.16}$$

The spatial domain is $0 \le x \le 2$, $0 \le y \le 1$, and boundary conditions taken from the true solution,

$$u(x, y) = e^{xy}. \tag{2.17}$$

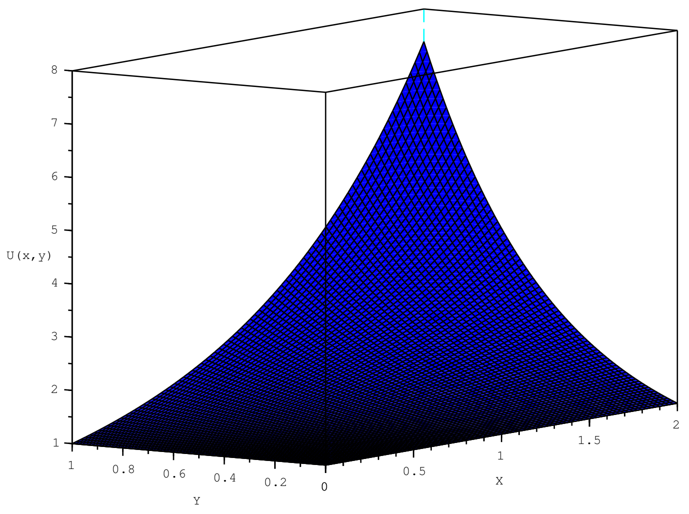


Figure 2.6: The true solution to EPDE problem 2.

2.1.4 Two-Dimensional Parabolic Partial Differential Equations

Two-dimensional Parabolic PDEs (2DPPDEs) are the time-dependent, two spatial dimension problem class, and can be considered to be a generalization of the PPDE problem class to two spatial dimensions. We will only be considering second-order spatial derivative 2DPPDEs, and with only the first temporal derivative of the solution present. The general form we will consider in this thesis is,

$$u_t(x,y,t) = f(x,y,t,u(x,y,t),u_x(x,y,t),u_{xx}(x,y,t),u_y(x,y,t),u_{yy}(x,y,t),u_{xy}(x,y,t)), \quad (2.18)$$

where $u(x,y,t)$ is the true solution to the 2DPPDE. The spatial domain is $A \leq x \leq B$, $C \leq y \leq D$, and temporal domain is $t_0 \leq t \leq t_f$.

The initial condition is,

$$u(x,y,t_0) = \mu(x,y), \quad (2.19)$$

where $\mu(x,y)$ is some function which represents the beginning state of the solution across the spatial domain at the initial time. The boundary conditions are $u(A,y,t) = \alpha(y,t)$, $u(B,y,t) = \beta(y,t)$,

$u(x,C,t) = \zeta(x,t)$, $u(x,D,t) = \eta(x,t)$. Note that, for consistency, we must have $u(A,C,t) = \alpha(C,t) = \zeta(A,t)$, $u(B,C,t) = \beta(C,t) = \zeta(B,t)$, $u(A,D,t) = \alpha(D,t) = \eta(A,t)$, and $u(B,D,t) = \beta(D,t) = \eta(B,t)$.

2DPPDE Problem 1

The PDE given by,

$$u_t(x,y,t) = \epsilon(u_{xx}(x,y,t) + u_{yy}(x,y,t)), \tag{2.20}$$

where the problem dependent parameter, $\epsilon = \frac{1}{10}$. The spatial domain is $0 \le x \le 2$, $0 \le y \le 2$ and the temporal domain is $0 \le t \le 1$. The initial and boundary conditions are taken from the true solution,

$$u(x,y,t) = \sin\left(\frac{\pi}{2}x\right)\sin\left(\frac{\pi}{2}y\right)e^{\frac{-\epsilon t\pi^2}{2}}. \tag{2.21}$$

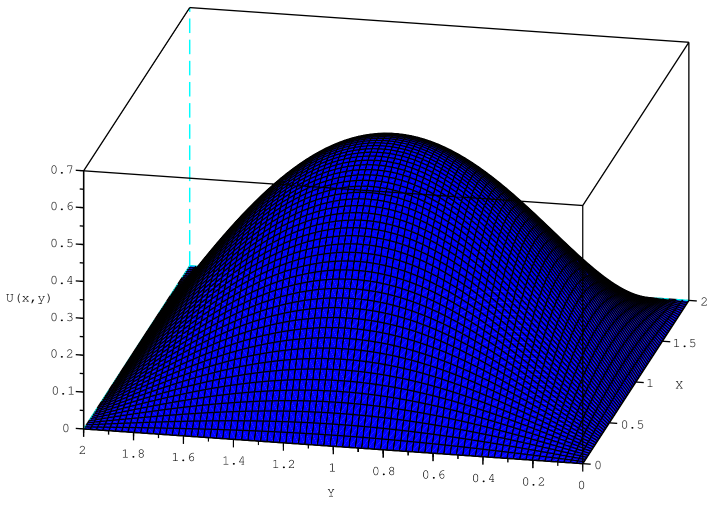


Figure 2.7: The true solution to 2DPPDE problem 1 at $t = 1$.

2DPPDE Problem 2

The PDE given by,

$$u_t(x,y,t) = \epsilon(u_{xx}(x,y,t) + u_{yy}(x,y,t)) - u(x,y,t)(u_x(x,y,t) + u_y(x,y,t)), \tag{2.22}$$

where the problem dependent parameter, $\epsilon = \frac{1}{10}$. The spatial domain is $0 \leq x \leq 1$, $0 \leq y \leq 1$ and the temporal domain is $0 \leq t \leq 1$. The initial and boundary conditions are taken from the true solution,

$$u(x, y, t) = (1 + e^{\frac{x+y-t}{2\epsilon}})^{-1}. \tag{2.23}$$

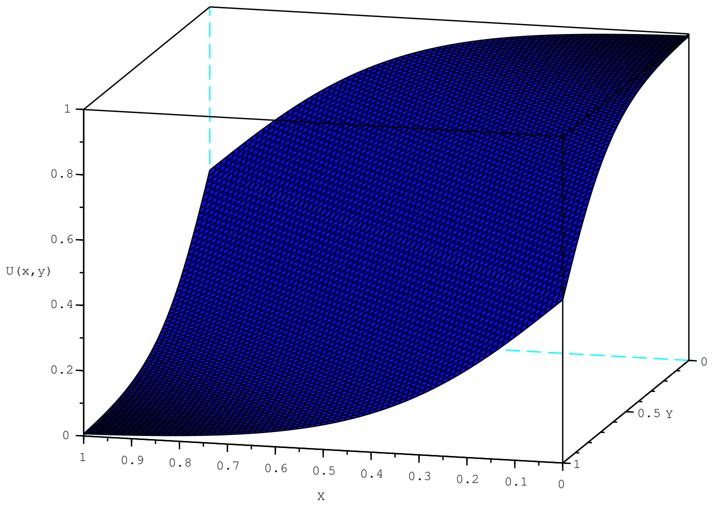


Figure 2.8: The true solution to 2DPPDE problem 2 at $t = 1$.

2.2 B-spline Gaussian Collocation

B-spline Gaussian collocation is a specific family of methods from the more general class of collocation methods. This section will begin by introducing the general concept of collocation, followed by an explanation of B-spline Gaussian collocation. We will then describe how a collocation solution can be calculated, along with examples of existing numerical software, for each of the four problem classes.

The main idea of collocation is to create a space of potential approximate solutions as the span of a set of basis functions, followed by the choosing of the collocation points, which are points in the spatial domain at which the approximate solution is required to satisfy the differential equation. Collocation can be used for time-dependent problems as well by applying this algorithm at multiple time steps through the temporal domain.

The spatial domain of a collocation problem is usually divided into smaller subintervals or rectangles determined by a set of mesh points. When this approach is used, the basis functions we consider are defined across each of these subregions, and will be a polynomial or a bivariate polynomial on each subregion. The collocation solution will therefore be a piecewise polynomial or a bivariate piecewise polynomial across the spatial domain. B-spline basis functions are a common choice for this algorithm class as they can be calculated such that the resulting piecewise polynomial or bivariate piecewise polynomial will have any desired degree and continuity. Furthermore there are easily available, established software packages for calculating B-splines, such as the de Boor B-spline package [11].

By choosing the collocation points on each subregion to be the mapping of Gauss-Legendre quadrature points from $[-1, 1]$ onto each subregion of the spatial domain, we get Gaussian collocation. Combining this with B-splines as the choice of basis functions, we get B-spline Gaussian collocation. We express the collocation solution as a linear combination of the B-spline basis functions with unknown coefficients. The unknown coefficients are determined by requiring that the collocation solution satisfy the DE at the collocation points on each subregion and that it also satisfies the boundary conditions at certain points.

This gives a non-linear system of equations in the time-independent cases or a system of time-dependent, non-linear differential algebraic equations (DAEs) in the time-dependent cases. The unknowns in these systems are the coefficients of the B-spline basis functions. When the collocation and boundary conditions are ordered appropriately, the resulting linear systems will have what is known as an almost block diagonal (ABD) structure. A system of this structure contains mostly zeros apart from pairs of non-zero blocks which occur along the main diagonal; further discussion on this structure in the context of collocation methods can be found in [16]. Matrices of this structure can be solved more efficiently than ones having a fully dense structure by using software tools such as COLROW [12] which allow for lesser time and space complexity than would be obtained if the linear systems were assumed to be dense.

For the time-independent cases, the non-linear equations are solved using software that solves the equations to obtain B-spline coefficients that are accurate to within a given tolerance. In the time-dependent cases, the DAE solver can enable temporal error control of the collocation solution. The error of a collocation solution in a time-dependent problem class will consist of both a spatial and temporal component. (This will be further discussed in section 2.3). By using a DAE solver with error control, the temporal error in the collocation solution (i.e., the error in the B-spline coefficients) can be controlled, which then requires just the spatial error to be controlled to achieve full error control.

2.2.1 One-dimensional B-spline Gaussian Collocation

When computing one-dimensional collocation solutions there are two input parameters which will affect the accuracy of the approximation. These are the degree, p, of the piecewise polynomials that represent the collocation solution, and the mesh that divides the spatial domain, which within this thesis is taken as a uniform partitioning of the spatial domain, creating N subintervals based on $N+1$ mesh points. The mesh points will be represented as $x_i : i = 1, 2, \ldots, N+1$, where $x_1 = A$, $x_{N+1} = B$. The points are sorted in increasing order.

To define B-spline basis functions, $b_i(x)$, of degree p over N subintervals and having C^1-continuity imposed, we need to specify $NCPTS = N(p-1)+2$ knots across the spatial domain. These knots take on the same values as the mesh points, but will contain repetitions of the mesh points. The knot points, $\kappa_i : i = 1, 2, \ldots, NCPTS$, are defined such that x_1 and x_{N+1} appear $p+1$ times while the remaining mesh points appear $p-1$ times each, with the knots appearing in increasing order of x-value. From these knots, the B-spline basis functions, $b_i(x) : i = 1, 2, \ldots, NCPTS$, are then defined. See [11] for further details.

The basis functions, along with coefficients, $c_i : i = 1, 2, \ldots, NCPTS$, in the BVODE case, or $c_i(t) : i = 1, 2, \ldots, NCPTS$, in the PPDE case, define the collocation solution, and allow for it to be evaluated at any point in the spatial domain in the BVODE case, or the spatial-temporal domain in the PPDE case.

In the BVODE case the collocation solution, $U(x)$, is given by,

$$U\left(x\right) = \sum_{i=1}^{NCPTS} c_i b_i(x), \tag{2.24}$$

while in the PPDE case, the collocation solution, $U(x,t)$, has the form,

$$U\left(x,t\right) = \sum_{i=1}^{NCPTS} c_i(t) b_i(x). \tag{2.25}$$

To discretize the spatial dependence of the collocation solution, we then define the collocation points, $\gamma_i : i = 1, 2, \ldots, NCPTS - 2$, such that each of the N subintervals contains $p-1$ collocation points. For each subinterval the collocation points will be the mapping of the set of $(p-1)$ Gauss-Legendre points from $[-1,1]$ to $[x_i, x_{i+1}], i = 1, 2, \ldots, N$. At each of these collocation points we will require that the collocation solution exactly satisfy the ODE or PDE.

In the BVODE case these collocation conditions will be of form,

$$f\left(\gamma_i, U\left(\gamma_i\right), U_x\left(\gamma_i\right)\right) - U_{xx}\left(\gamma_i\right) = 0 : i = 1, 2, \ldots, NCPTS - 2, \tag{2.26}$$

while in the PPDE case they are of form,

$$f\left(\gamma_i, t, U\left(\gamma_i, t\right), U_x\left(\gamma_i, t\right), U_{xx}\left(\gamma_i, t\right)\right) - U_t\left(\gamma_i, t\right) = 0 : i = 1, 2, \ldots, NCPTS - 2. \tag{2.27}$$

The remaining two conditions to determine the coefficients for the $NCPTS$ B-spline basis functions are obtained by requiring that the collocation solution satisfy the boundary conditions. These conditions will have the following form in the BVODE case,

$$U(A) - \alpha = 0, \quad U(B) - \beta = 0, \tag{2.28}$$

while in the PPDE case, they will have the form,

$$U(A,t) - \alpha(t) = 0, \quad U(B,t) - \beta(t) = 0. \tag{2.29}$$

As mentioned earlier, a system with an ABD structure can allow for greater efficiency and as such it is desired that we construct the system of non-linear algebraic equations in such a way that this structure will arise. Due to the locality of the B-spline basis functions, shown in Figures 2.9 and 2.10, where there are only $p+1$ potentially non-zero basis functions on each subinterval, a given collocation or boundary condition will depend on at most $p+1$ B-spline coefficients.

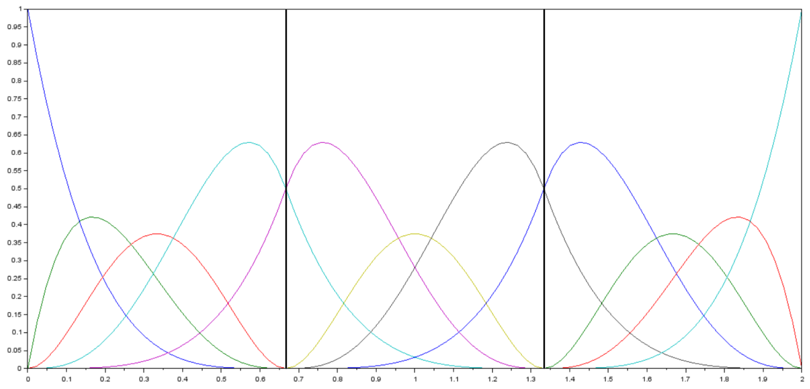


Figure 2.9: B-Spline basis functions for a collocation solution with $p = 4$ and $N = 3$. At most 5 (i.e, $p+1$), basis functions are non-zero on any subinterval.

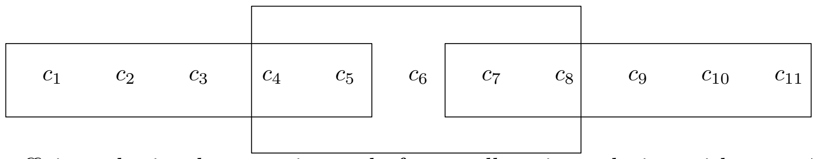


Figure 2.10: Coefficient sharing between intervals for a collocation solution with $p = 4$, $N = 3$. The three rectangles represent the three subintervals, and the solution on a given subinterval can depend on the coefficients within the corresponding rectangle.

Thus by ordering the equations in the non-linear algebraic or differential algebraic system so that the conditions appear in increasing order of the x-value at which the collocation solution is evaluated, an ABD structure will arise. The system ordered this way will be as follows in the BVODE case,

$$\left[\begin{array}{c} U\left(A\right)-\alpha \\ f\left(\gamma_1,U\left(\gamma_1\right),U_x\left(\gamma_1\right)\right)-U_{xx}\left(\gamma_1\right) \\ \vdots \\ f\left(\gamma_{NCPTS-2},U\left(\gamma_{NCPTS-2}\right),U_x\left(\gamma_{NCPTS-2}\right)\right)-U_{xx}\left(\gamma_{NCPTS-2}\right) \\ U\left(B\right)-\beta \end{array}\right]=\underline{0}. \qquad (2.30)$$

In the PPDE case, the system of boundary and collocation conditions has the form,

$$\left[\begin{array}{c} U\left(A,t\right)-\alpha\left(t\right) \\ f\left(\gamma_1,t,U\left(\gamma_1,t\right),U_x\left(\gamma_1,t\right),U_{xx}\left(\gamma_1,t\right)\right)-U_t\left(\gamma_1,t\right) \\ \vdots \\ f\left(\gamma_{NCPTS-2},t,U\left(\gamma_{NCPTS-2},t\right),U_x\left(\gamma_{NCPTS-2},t\right),U_{xx}\left(\gamma_{NCPTS-2},t\right)\right)-U_t\left(\gamma_{NCPTS-2},t\right) \\ U\left(B,t\right)-\beta\left(t\right) \end{array}\right]=\underline{0}. \qquad (2.31)$$

A number of software packages that employ B-spline Gaussian collocation to obtain approximate solutions for BVODE or PPDE problems have been developed. An early example of B-spline Gaussian collocation software for BVODE problems is the COLSYS software [4]. An early example of B-spline Gaussian collocation software for the PPDE case is PDECOL [17], which led to EPDCOL [15]. Another more recent example of B-spline Gaussian collocation software for the PPDE case is the BACOL software [22]. Both the COLSYS and BACOL packages compute error controlled collocation solutions. We discuss this later in the thesis.

2.2.2 Two-dimensional B-spline Gaussian Collocation

The approach discussed in this section employs a tensor-product of the B-spline basis functions, and as such has much in common with a one-dimensional approach. In this case we will have degree p piecewise polynomials in x, and degree q piecewise polynomials in y. We will also have a mesh in x and y, defined by N and M subintervals across the x and y spatial domains. These mesh points will be represented as, $x_i : i = 1, 2, \dots N + 1$, and $y_i : i = 1, 2, \dots, M + 1$, in x and y, respectively.

The B-spline basis functions will be determined by $NCPTX = N(p-1)+2$ knots in x, and $NCPTY = M(q-1)+2$ knots in y. These knots will be constructed in the same manner as in the one-dimensional case, where the exterior mesh points are repeated $p+1$ or $q+1$ times while the interior points are repeated $p-1$ or $q-1$ times. The resulting knots, $k_i : i = 1, 2, \ldots, NCPTX$, and $l_i : i = 1, 2, \ldots, NCPTY$, will be sorted in increasing order of x and y value. These knots will then be used to define the B-spline basis functions in x, $b_i(x) : i = 1, 2, \ldots, NCPTX$, and in y, $d_i(y) : i = 1, 2, \ldots, NCPTY$.

The basis functions, along with coefficients, $c_{i,j}$ or $c_{i,j}(t) : i = 1, 2, \ldots, NCPTX, j = 1, 2, \ldots, NCPTY$, allow us to define the collocation solution.

In the EPDE case, the collocation solution $U(x, y)$ is given by,

$$U(x,y) = \sum_{i=1}^{NCPTX} \sum_{j=1}^{NCPTY} c_{i,j} b_i(x) d_j(y). \tag{2.32}$$

In the 2DPPDE case, the collocation solution $U(x, y, t)$ is given by,

$$U(x,y,t) = \sum_{i=1}^{NCPTX} \sum_{j=1}^{NCPTY} c_{i,j}(t) b_i(x) d_j(y). \tag{2.33}$$

The spatial domain is then discretized by defining the collocation points in x,
$\gamma_i : i = 1, 2, \ldots, NCPTX-2$, and y, $\delta_j : j = 1, 2, \ldots, NCPTY-2$, such that each of the N or M subintervals contains $(p-1)$ or $(q-1)$ collocation points. These points will be the mapping of the set of $(p-1)$ Gauss-Legendre points from $[-1, 1]$ to $[x_i, x_{i+1}] : i = 1, 2, \ldots, N$, for the collocation points in x, and the set of $(q-1)$ Gauss-Legendre points mapped from $[-1, 1]$ to $[y_j, y_{j+1}] : j = 1, 2, \ldots, M$, for the collocation points in y. The collocation solution will be required to satisfy the DE at every combination of collocation points in x and y to obtain the collocation conditions. In the EPDE case these will be,

$$f(\gamma_i, \delta_j, U(\gamma_i, \delta_j), U_x(\gamma_i, \delta_j), U_y(\gamma_i, \delta_j)) - U_{xx}(\gamma_i, \delta_j) - U_{yy}(\gamma_i, \delta_j) = 0, \tag{2.34}$$

$$i = 1, 2, \ldots, NCPTX - 2, \quad j = 1, 2, \ldots, NCPTY - 2.$$

In the 2DPPDE case these will be

$$f\left(\gamma_i, \delta_j, t, U\left(\gamma_i, \delta_j, t\right), U_x\left(\gamma_i, \delta_j, t\right), U_{xx}\left(\gamma_i, \delta_j, t\right), U_y\left(\gamma_i, \delta_j, t\right), U_{yy}\left(\gamma_i, \delta_j, t\right), U_{xy}\left(\gamma_i, \delta_j, t\right)\right)$$
$$-U_t\left(\gamma_i, \delta_j, t\right) = 0, \quad (2.35)$$

$$i = 1, 2, \ldots, NCPTX - 2, \quad j = 1, 2, \ldots, NCPTY - 2.$$

The remaining conditions are obtained by requiring that the collocation solution satisfy the boundary conditions at certain points. The boundary conditions require the evaluation of the collocation solution at the projection of the collocation points in x onto the upper and lower bounds of the y-domain, the projection of the collocation points in y onto the upper and lower bounds of the x-domain, and at each of the four corners of the spatial domain. A pictoral representation of where the various boundary conditions are evaluated can be seen in Figure 2.11. The boundary conditions will be of the following form in the EPDE case,

$$0 = U(A, C) - \alpha(C), \quad 0 = U(A, D) - \beta(A), \quad 0 = U(B, C) - \zeta(B), \quad 0 = U(B, D) - \eta(D),$$
$$0 = U(A, \delta_j) - \alpha(\delta_j), \quad 0 = U(B, \delta_j) - \eta(\delta_j), \quad 0 = U(\gamma_i, C) - \zeta(\gamma_i), \quad 0 = U(\gamma_i, D) - \beta(\gamma_i),$$
$$i = 1, 2, \ldots, NCPTX - 2, \quad j = 1, 2, \ldots, NCPTY - 2.$$

In the 2DPPDE case,

$$0 = U(A, C, t) - \alpha(C, t), \quad 0 = U(A, D, t) - \beta(A, t), \quad 0 = U(B, C, t) - \zeta(B, t),$$
$$0 = U(B, D, t) - \eta(D, t), \quad 0 = U(A, \delta_j, t) - \alpha(\delta_j, t), \quad 0 = U(B, \delta_j, t) - \eta(\delta_j, t),$$
$$0 = U(\gamma_i, C, t) - \zeta(\gamma_i, t), \quad 0 = U(\gamma_i, D, t) - \beta(\gamma_i, t),$$
$$i = 1, 2, \ldots, NCPTX - 2, \quad j = 1, 2, \ldots, NCPTY - 2.$$

As in the one-dimensional case, we also desire the resulting system of equations to be of block diagonal form. While the B-splines have their locality within each spatial dimension, the tensor product causes any given evaluation to be dependant on up to $(p + 1)(q + 1)$ coefficients. The dependency of coefficients per subinterval can be seen in Figure 2.12.

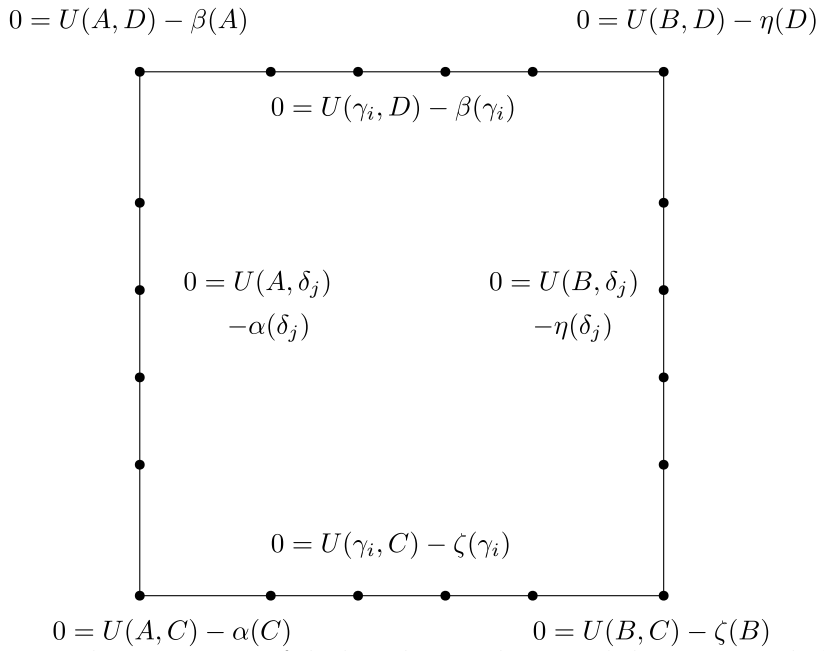
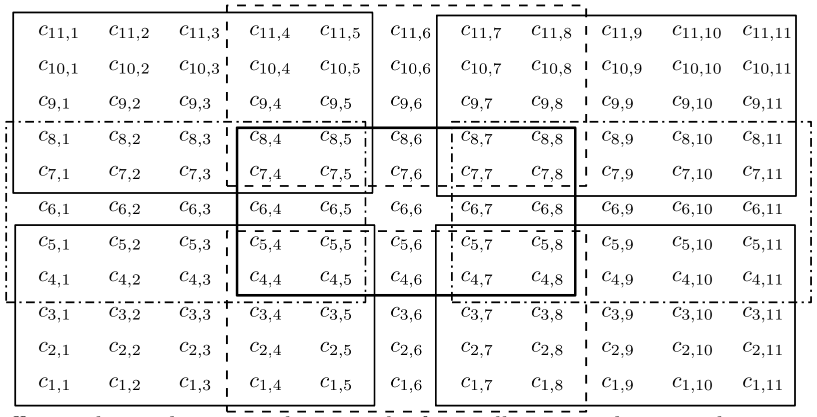


Figure 2.11: A pictorial representation of the boundary conditions and the points at which they evaluate the collocation solution, for the case $N = M = 1$, $p = q = 5$.

As the coefficients will be stored in a 1D array indexed by, $c_i : i = 1, 2, \ldots, NCPTX \times NCPTY$, the manner in which they get mapped into this one dimensional representation will determine how the non-linear system needs to be ordered so that an ABD structure arises. In this thesis we will use an x-major approach, meaning that given the coefficients, $c_{i,j} : i = 1, 2, \ldots, NCPTX, j = 1, 2, \ldots, NCPTY$, they will be represented one-dimensionally in the order $[c_{1,1}, c_{1,2}, \ldots, c_{NCPTX,NCPTY-1}, c_{NCPTX,NCPTY}]$. Using this mapping, we can then create a non-linear system of ABD structure by ordering it such that the first conditions evaluate the collocation solution at the minimal value of x, sorted in increasing order of the y point at which they require evaluation of the collocation solution. This x-major pattern is followed to order the remaining conditions. This will result in the system, $\underline{F} = \underline{0}$, for the EPDE case, where $\underline{F}$ is as given in Figure 2.13. In the 2DPPDE case, the system of boundary and collocation conditions has the form $\underline{F} = \underline{0}$, where $\underline{F}$ is given in Figure 2.14.



$c_{11,1}$ $c_{11,2}$ $c_{11,3}$ $c_{11,4}$ $c_{11,5}$ $c_{11,6}$ $c_{11,7}$ $c_{11,8}$ $c_{11,9}$ $c_{11,10}$ $c_{11,11}$
$c_{10,1}$ $c_{10,2}$ $c_{10,3}$ $c_{10,4}$ $c_{10,5}$ $c_{10,6}$ $c_{10,7}$ $c_{10,8}$ $c_{10,9}$ $c_{10,10}$ $c_{10,11}$
$c_{9,1}$ $c_{9,2}$ $c_{9,3}$ $c_{9,4}$ $c_{9,5}$ $c_{9,6}$ $c_{9,7}$ $c_{9,8}$ $c_{9,9}$ $c_{9,10}$ $c_{9,11}$
$c_{8,1}$ $c_{8,2}$ $c_{8,3}$ $c_{8,4}$ $c_{8,5}$ $c_{8,6}$ $c_{8,7}$ $c_{8,8}$ $c_{8,9}$ $c_{8,10}$ $c_{8,11}$
$c_{7,1}$ $c_{7,2}$ $c_{7,3}$ $c_{7,4}$ $c_{7,5}$ $c_{7,6}$ $c_{7,7}$ $c_{7,8}$ $c_{7,9}$ $c_{7,10}$ $c_{7,11}$
$c_{6,1}$ $c_{6,2}$ $c_{6,3}$ $c_{6,4}$ $c_{6,5}$ $c_{6,6}$ $c_{6,7}$ $c_{6,8}$ $c_{6,9}$ $c_{6,10}$ $c_{6,11}$
$c_{5,1}$ $c_{5,2}$ $c_{5,3}$ $c_{5,4}$ $c_{5,5}$ $c_{5,6}$ $c_{5,7}$ $c_{5,8}$ $c_{5,9}$ $c_{5,10}$ $c_{5,11}$
$c_{4,1}$ $c_{4,2}$ $c_{4,3}$ $c_{4,4}$ $c_{4,5}$ $c_{4,6}$ $c_{4,7}$ $c_{4,8}$ $c_{4,9}$ $c_{4,10}$ $c_{4,11}$
$c_{3,1}$ $c_{3,2}$ $c_{3,3}$ $c_{3,4}$ $c_{3,5}$ $c_{3,6}$ $c_{3,7}$ $c_{3,8}$ $c_{3,9}$ $c_{3,10}$ $c_{3,11}$
$c_{2,1}$ $c_{2,2}$ $c_{2,3}$ $c_{2,4}$ $c_{2,5}$ $c_{2,6}$ $c_{2,7}$ $c_{2,8}$ $c_{2,9}$ $c_{2,10}$ $c_{2,11}$
$c_{1,1}$ $c_{1,2}$ $c_{1,3}$ $c_{1,4}$ $c_{1,5}$ $c_{1,6}$ $c_{1,7}$ $c_{1,8}$ $c_{1,9}$ $c_{1,10}$ $c_{1,11}$

Figure 2.12: Coefficient sharing between sub-rectangles for a collocation solution with $p = q = 4$, $N = M = 3$. The rectangles represent the various subrectangles, and contain within them the coefficients which correspond to the potential non-zero B-spline basis functions.

$$
\left[\begin{array}{c}
U(A,C)-\alpha(C)\\
U(A,\delta_1)-\alpha(\delta_1)\\
U(A,\delta_2)-\alpha(\delta_2)\\
\vdots\\
U(A,\delta_{NCPTY-2})-\alpha(\delta_{NCPTY-2})\\
U(A,D)-\beta(A)\\
\hdashline
U(\gamma_1,C)-\zeta(\gamma_1)\\
f\left(\gamma_1,\delta_1,U\left(\gamma_1,\delta_1\right),U_x\left(\gamma_1,\delta_1\right),U_y\left(\gamma_1,\delta_1\right)\right)-U_{xx}\left(\gamma_1,\delta_1\right)-U_{yy}\left(\gamma_1,\delta_1\right)\\
f\left(\gamma_1,\delta_2,U\left(\gamma_1,\delta_2\right),U_x\left(\gamma_1,\delta_2\right),U_y\left(\gamma_1,\delta_2\right)\right)-U_{xx}\left(\gamma_1,\delta_2\right)-U_{yy}\left(\gamma_1,\delta_2\right)\\
\vdots\\
f\left(\gamma_1,\delta_{NCPTY-2},U\left(\gamma_1,\delta_{NCPTY-2}\right),U_x\left(\gamma_1,\delta_{NCPTY-2}\right),U_y\left(\gamma_1,\delta_{NCPTY-2}\right)\right)\\
-U_{xx}\left(\gamma_1,\delta_{NCPTY-2}\right)-U_{yy}\left(\gamma_1,\delta_{NCPTY-2}\right)\\
U(\gamma_1,D)-\beta(\gamma_1)\\
\hdashline
\vdots\\
\hdashline
U(\gamma_{NCPTX-2},C)-\zeta(\gamma_{NCPTX-2})\\
f\left(\gamma_{NCPTX-2},\delta_1,U\left(\gamma_{NCPTX-2},\delta_1\right),U_x\left(\gamma_{NCPTX-2},\delta_1\right),U_y\left(\gamma_{NCPTX-2},\delta_1\right)\right)\\
-U_{xx}\left(\gamma_{NCPTX-2},\delta_1\right)-U_{yy}\left(\gamma_{NCPTX-2},\delta_1\right)\\
f\left(\gamma_{NCPTX-2},\delta_2,U\left(\gamma_{NCPTX-2},\delta_2\right),U_x\left(\gamma_{NCPTX-2},\delta_2\right),U_y\left(\gamma_{NCPTX-2},\delta_2\right)\right)\\
-U_{xx}\left(\gamma_{NCPTX-2},\delta_2\right)-U_{yy}\left(\gamma_{NCPTX-2},\delta_2\right)\\
\vdots\\
f\big(\gamma_{NCPTX-2},\delta_{NCPTY-2},U\left(\gamma_{NCPTX-2},\delta_{NCPTY-2}\right),U_x\left(\gamma_{NCPTX-2},\delta_{NCPTY-2}\right),\\
U_y\left(\gamma_{NCPTX-2},\delta_{NCPTY-2}\right)\big)-U_{xx}\left(\gamma_{NCPTX-2},\delta_{NCPTY-2}\right)-U_{yy}\left(\gamma_{NCPTX-2},\delta_{NCPTY-2}\right)\\
U(\gamma_{NCPTX-2},D)-\beta(\gamma_{NCPTX-2})\\
\hdashline
U(B,C)-\alpha(C)\\
U(B,\delta_1)-\eta(\delta_1)\\
U(B,\delta_2)-\eta(\delta_2)\\
\vdots\\
U(B,\delta_{NCPTY-2})-\eta(\delta_{NCPTY-2})\\
U(B,D)-\beta(B)
\end{array}\right]
$$

Figure 2.13: The ordering of boundary and collocation conditions for a general EPDE problem in order to achieve an ABD structure.

$$\left[\begin{array}{c}
U(A,C,t)-\alpha(C,t)\\
U(A,\delta_1,t)-\alpha(\delta_1,t)\\
U(A,\delta_2,t)-\alpha(\delta_2,t)\\
\vdots\\
U(A,\delta_{NCPTY-2},t)-\alpha(\delta_{NCPTY-2},t)\\
U(A,D,t)-\beta(A,t)\\
\hdashline
U(\gamma_1,C,t)-\zeta(\gamma_1,t)\\
f\big(\gamma_1,\delta_1,t,U\left(\gamma_1,\delta_1,t\right),U_x\left(\gamma_1,\delta_1,t\right),U_{xx}\left(\gamma_1,\delta_1,t\right),U_y\left(\gamma_1,\delta_1,t\right),U_{yy}\left(\gamma_1,\delta_1,t\right),\\
U_{xy}\left(\gamma_1,\delta_1,t\right)\big)-U_t\left(\gamma_1,\delta_1,t\right)\\
f\big(\gamma_1,\delta_2,t,U\left(\gamma_1,\delta_2,t\right),U_x\left(\gamma_1,\delta_2,t\right),U_{xx}\left(\gamma_1,\delta_2,t\right),U_y\left(\gamma_1,\delta_2,t\right),U_{yy}\left(\gamma_1,\delta_2,t\right),\\
U_{xy}\left(\gamma_1,\delta_2,t\right)\big)-U_t\left(\gamma_1,\delta_2,t\right)\\
\vdots\\
f\big(\gamma_1,\delta_{NCPTY-2},t,U\left(\gamma_1,\delta_{NCPTY-2},t\right),U_x\left(\gamma_1,\delta_{NCPTY-2},t\right),U_{xx}\left(\gamma_1,\delta_{NCPTY-2},t\right),\\
U_y\left(\gamma_1,\delta_{NCPTY-2},t\right),U_{yy}\left(\gamma_1,\delta_{NCPTY-2},t\right),\\
U_{xy}\left(\gamma_1,\delta_{NCPTY-2},t\right)\big)-U_t\left(\gamma_1,\delta_{NCPTY-2},t\right)\\
U(\gamma_1,D,t)-\beta(\gamma_1,t)\\
\hdashline
\vdots\\
\hdashline
U(\gamma_{NCPTX-2},C,t)-\zeta(\gamma_{NCPTX-2},t)\\
f\big(\gamma_{NCPTX-2},\delta_1,t,U\left(\gamma_{NCPTX-2},\delta_1,t\right),U_x\left(\gamma_{NCPTX-2},\delta_1,t\right),U_{xx}\left(\gamma_{NCPTX-2},\delta_1,t\right),\\
U_y\left(\gamma_{NCPTX-2},\delta_1,t\right),U_{yy}\left(\gamma_{NCPTX-2},\delta_1,t\right),\\
U_{xy}\left(\gamma_{NCPTX-2},\delta_1,t\right)\big)-U_t\left(\gamma_{NCPTX-2},\delta_1,t\right)\\
f\big(\gamma_{NCPTX-2},\delta_2,t,U\left(\gamma_{NCPTX-2},\delta_2,t\right),U_x\left(\gamma_{NCPTX-2},\delta_2,t\right),U_{xx}\left(\gamma_{NCPTX-2},\delta_2,t\right),\\
U_y\left(\gamma_{NCPTX-2},\delta_2,t\right),U_{yy}\left(\gamma_{NCPTX-2},\delta_2,t\right),U_{xy}\left(\gamma_{NCPTX-2},\delta_2,t\right)\big)-U_t\left(\gamma_{NCPTX-2},\delta_2,t\right)\\
\vdots\\
f\big(\gamma_{NCPTX-2},\delta_{NCPTY-2},t,U\left(\gamma_{NCPTX-2},\delta_{NCPTY-2},t\right),U_x\left(\gamma_{NCPTX-2},\delta_{NCPTY-2},t\right),\\
U_{xx}\left(\gamma_{NCPTX-2},\delta_{NCPTY-2},t\right),U_y\left(\gamma_{NCPTX-2},\delta_{NCPTY-2},t\right),U_{yy}\left(\gamma_{NCPTX-2},\delta_{NCPTY-2},t\right),\\
U_{xy}\left(\gamma_{NCPTX-2},\delta_{NCPTY-2},t\right)\big)-U_t\left(\gamma_{NCPTX-2},\delta_{NCPTY-2},t\right)\\
U(\gamma_{NCPTX-2},D,t)-\beta(\gamma_{NCPTX-2},t)\\
\hdashline
U(B,C,t)-\alpha(C,t)\\
U(B,\delta_1,t)-\eta(\delta_1,t)\\
U(B,\delta_2,t)-\eta(\delta_2,t)\\
\vdots\\
U(B,\delta_{NCPTY-2},t)-\eta(\delta_{NCPTY-2},t)\\
U(B,D,t)-\beta(B,t)
\end{array}\right]$$

Figure 2.14: The ordering of boundary and collocation conditions for a general 2DPPDE problem in order to achieve an ABD structure.

Some examples of B-spline Gaussian collocation software for the two spatial dimension case are the ELLPACK environment [20] which is able to approximate the solution to an EPDE problem and the BACOL2D software [16] which approximates solutions to 2DPPDE problems.

2.3 Error Estimation and Control

As a collocation solution is an approximation to the true solution of a DE, it is guaranteed there will be a difference between the collocation solution and the true solution. That is, when calculating any approximation with numerical software there will be an error associated with the approximation. This raises the question of what the error is. In most practical uses of numerical methods for DEs the true solution is not known, and as such the true error can not be calculated. Instead an error estimate must be calculated in an attempt to answer the above question.

While returning an error estimate along with the approximate solution allows the user to know the quality of the approximation, it is more desirable that the user be able to specify a tolerance to be applied to the error estimate. This concept is error control, which when employed by numerical software, will require the software to return an approximation such that the corresponding error estimate satisfies a user given tolerance. Error control is desirable in numerical software as it can allow for the software to compute an approximation in less computational time. An example of this can be seen in the BACOLI software [19], when solving the one-layer Burgers equation, where the spatial mesh is adaptively refined in order to be able to follow a layer of difficult solution behaviour as it moves across the spatial domain as time progresses. This is the essence of the spatial error control algorithm employed by BACOLI. The package also employs a temporal error control DAE solver, DASSL [18], to compute the B-spline coefficients. This allows BACOLI to compute an approximate solution whose error estimate satisfies a given tolerance using lower computational costs than would be required if a fixed and much finer spatial mesh was employed. (A finer mesh across the entire spatial domain would be required in the non-adaptive algorithm in order to obtain a solution whose error estimate satisfies the tolerance).

2.3.1 Error of a Collocation Solution

The error of a Gaussian collocation solution is a research question that has been heavily explored. In the BVODE case it has been proven that the error of a collocation solution based on a piecewise polynomial of degree p will be $O(h^{p+1})$ where h is the length of the longest subinterval, while at the mesh points the error will be $O(h^{2(p-1)})$ [10]. In the PPDE case these results have also been proven [9], [13]. Furthermore in the BVODE case, it has been shown that there are points within each subinterval where the collocation error is $O(h^{p+2})$; see [5]. In the PPDE case, these points have been experimentally shown to have an error that is $O(h^{p+2})$; see, for example, [2]. These points will be referred to as non-mesh super-convergent points (NMSCPs) in this thesis.

These aforementioned errors are the errors of the collocation solution associated with the spatial domain. In the time-dependant cases there is also the temporal error associated with the computation of the B-spline coefficients. As mentioned above, by using a DAE solver with error control the temporal error in the collocation solution can be controlled. This allows the development of error controlled algorithms for calculating collocation solutions by having the DAE solver control the temporal error to be within a user specified tolerance, and then applying a spatially adaptive collocation algorithm so that the estimated spatial error meets the user specified tolerance.

2.3.2 Error Estimation

When calculating an error estimate for an approximation, a more accurate solution is often used in place of the true solution. In other words, an error estimate is often calculated by looking at the difference between a given approximation, and one of greater accuracy. One method of acquiring the more accurate solution is through Richardson extrapolation, an example of which can be seen in the COLSYS software [4]. When A_h represents a collocation solution calculated on a mesh with a subinterval length of h and A represents the true solution, the error in A_h can be expressed as,

$$A - A_h = Kh^{p+1} + O(h^{p+2}), \tag{2.36}$$

where K is some constant. Then a second approximation is calculated on a mesh obtained by halving each subinterval of the original mesh. For this latter mesh, the subinterval size is $\frac{h}{2}$ and the corresponding approximate solution is $A_{h/2}$. In this case the error is,

$$A - A_{h/2} = K\left(\frac{h}{2}\right)^{p+1} + O(h^{p+2}). \tag{2.37}$$

Then

$$\begin{aligned} &\left|A_h - A_{h/2}\right|, \\ &= \left|A - A_h - \left(A - A_{h/2}\right)\right|, \\ &= \left|Kh^{p+1} - K\left(\tfrac{h}{2}\right)^{p+1} + O\left(h^{p+2}\right)\right|, \\ &= \left|Kh^{p+1}\left(1 - \left(\tfrac{1}{2}\right)^{p+1}\right)\right| + O(h^{p+2}). \end{aligned}$$

Thus

$$\frac{\left|A_h - A_{h/2}\right|}{\left(1 - \left(\frac{1}{2}\right)^{p+1}\right)} = Kh^{p+1} + O(h^{p+2}), \tag{2.38}$$

is an estimate of the leading order term in the error for A_h.

Another example of how an error estimate can be computed is shown in the BACOL software [22] which calculates a collocation solution of degree p as the returned approximation, along with a second collocation solution of degree $p+1$ that is used to estimate the error. The difference between the two gives an error estimate for the lower order approximate solution.

While being able to calculate an error estimate is very desirable, calculating an error estimate will have a computational cost. In the case of the BACOL software the cost of calculating the error estimate is greater than the cost of calculating the returned solution. This extra computational cost is undesirable, and it is important to be able to efficiently calculate an error estimate. An alternative which can be less computationally expensive than calculating a second approximation of greater accuracy, is to instead calculate one of lower accuracy than the returned solution. This method is referred to as local extrapolation. While this method may require less computational cost, the error estimate may not be as accurate.

2.3.3 Interpolation-Based Error Estimation

The superconvergent interpolant (SCI) [1] addresses the issue associated with BACOL in which a second higher order collocation solution is computed. In order to obtain an error estimate, the SCI replaces the higher order collocation solution with a piecewise polynomial Hermite-Birkhoff interpolant. A higher order interpolant is obtained by taking advantage of the increased accuracy at the mesh points, as well as at the NMSCPs. The SCI will be a piecewise polynomial where the degree is $p + 1$ on each of the subintervals. Enough interpolation points are chosen so that the interpolation error is dominated by the spatial error of the solution and derivative approximations at the points which are interpolated. The SCI is determined on each subinterval by interpolating the collocation solution and derivative approximations at the end points of a given subinterval, as well as the collocation solution at the $p-3$ NMSCP within the subinterval, and the closest NMSCP from each adjacent subinterval (or the two closest if there is only one adjacent subinterval). This is a total of $p+3$ data values. The points where only the solution value is interpolated will be denoted $w_i : i = 1, 2, \ldots, p-1$. The solution and derivative values are interpolated at the mesh points, s_1, s_2, bounding the subinterval.

The lower order interpolant (LOI) [3] replaces the lower order collocation solution computed by BACOL with an interpolant. As with the SCI, the LOI is also constructed as a piecewise polynomial Hermite-Birkhoff interpolant. In this case it is desired that the interpolation error dominates the data error. To do so, the LOI will interpolate only $p + 1$ data values. The solution value is interpolated at what would be the NMSCP within the subinterval for a collocation solution of degree $p-1$; these points are denoted as $w_i : i = 1, 2, \ldots, p-4$. The solution and derivative values are interpolated at the mesh points, s_1, s_2, which bound the subinterval.

The general form used to evaluate the SCI and LOI is derived from the Hermite-Birkhoff form presented by Finden [14], where s is the set of points which the solution and derivative values are interpolated, and w are the points where just the solution values are interpolated. The general form of the piecewise polynomial Hermite-Birkhoff interpolant upon which the SCI and LOI are based is,

$$q\left(x\right)=\sum_{i=1}^{|s|}H_i\left(x\right)u\left(s_i\right)+\sum_{i=1}^{|s|}\bar{H}_i\left(x\right)u_x\left(s_i\right)+\sum_{j=1}^{|w|}G_j\left(x\right)u\left(w_j\right), \tag{2.39}$$

where,

$$H_i\left(x\right)=\left(1-\left(x-s_i\right)\lambda_i\right)\hat{H}_i\left(x\right),\quad \bar{H}_i\left(x\right)=\left(x-s_i\right)\hat{H}_i\left(x\right),$$
$$\hat{H}_i\left(x\right)=\tfrac{\psi_i^2(x)\phi(x)}{\psi_i^2(s_i)\phi(s_i)},\quad G_j\left(x\right)=\tfrac{\psi^2(x)\phi_j(x)}{\psi^2(s_i)\phi_j(s_i)},$$

and where,

$$\psi_i\left(x\right)=\prod_{\substack{k=1\\k\neq i}}^{|s|}\left(x-s_k\right),\ \ \psi\left(x\right)=\prod_{k=1}^{|s|}\left(x-s_k\right),$$
$$\phi_j\left(x\right)=\prod_{\substack{k=1\\k\neq j}}^{|w|}\left(x-w_k\right),\ \ \phi\left(x\right)=\prod_{k=1}^{|w|}\left(x-w_k\right).$$

Both the SCI and LOI interpolation schemes are used within error estimation methods that make use of a quadrature rule to estimate the L^2 error of the collocation solution. (The L^2 norm, e, of a function, $f(x)$, on the domain $A \leq x \leq B$, is calculated as, $e=\sqrt{\int_A^B f(x)^2dx}$).

This approach can be used for estimating the error across the entire spatial domain, which will be referred to as the global error, or for a single subinterval. We will use x_α and x_ω as the lower and upper bounds, respectively, of the region across which the error is being estimated. We will also use p as the degree of the collocation solution. The following formula is used to calculate an error estimate for the subinterval $[x_\alpha, x_\omega]$,

$$E=\sqrt{\int_{x_\alpha}^{x_\omega}\left(\frac{|U\left(x\right)-q\left(x\right)|}{atol+rtol|U\left(x\right)|}\right)^2dx}, \tag{2.40}$$

where *atol* is the absolute error tolerance, *rtol* is the relative error tolerance, $U(x)$ is the collocation solution, and $q(x)$ is the appropriate interpolant. For the PPDE case we replace $U(x)$ and $q(x)$ with $U(x,t)$ and $q(x,t)$.

Chapter 3

Interpolants for Error Estimation

The main purpose of this thesis is to explore potential interpolants to be used for spatial error estimation of a two-dimensional collocation solution. Within this chapter we present three interpolants, two being adaptations of the previously developed SCI and LOI 1D interpolants to two dimensions. The third interpolant, although inspired by the LOI, is a new type of interpolant for this application, and it has not been previously published.

3.1 SCI and LOI Interpolants in Two-Dimensions

Here we consider the 2D non-time-dependent case. The SCI and LOI interpolants, as described earlier, estimate the spatial error for a collocation solution of a PDE with one spatial dimension. In this thesis we present a generalization of the SCI and LOI schemes to two spatial dimensions using a tensor-product approach. Given a collocation solution, $U(x, y)$, of degree p in x, and degree q in y, a desired point of evaluation, (x, y), and subintervals such that $x \in [x_\alpha, x_\omega]$, and $y \in [y_\alpha, y_\omega]$, the 2D SCI or LOI can be evaluated through the following procedure. The points $s_i : i = 1, 2$, and $t_i : i = 1, 2$, are the x and y values at which solution and derivative values are interpolated; these are set such that $s_1 = x_\alpha, s_2 = x_\omega, t_1 = y_\alpha, t_2 = y_\omega$. The points at which only the solution value is interpolated, w_i in x and v_i in y, are chosen following the same method outlined in Chapter 2. Then the interpolant can be constructed as follows:

$$q(x,y) = \sum_{i=1}^{2}(H_i(x)\eta(s_i,y)) + \sum_{i=1}^{2}(\bar{H}_i(x)\zeta(s_i,y)) + \sum_{j=1}^{|w|}(G_j(x)\eta(w_j,y)), \tag{3.1}$$

where

$$H_i\left(x\right) = \left(1-\left(x-s_i\right)\lambda_i\right)\hat{H}_i\left(x\right), \quad \bar{H}_i\left(x\right) = \left(x-s_i\right)\hat{H}_i\left(x\right), \quad \hat{H}_i\left(x\right) = \tfrac{\psi_i^2(x)\phi(x)}{\psi_i^2(s_i)\phi(s_i)},$$
$$G_j\left(x\right) = \tfrac{\psi^2(x)\phi_j(x)}{\psi^2(s_i)\phi_j(s_i)}, \quad \eta(x,y) = \sum_{j=1}^{2}(F_j(y)U(x,t_j)) + \sum_{j=1}^{2}(\bar{F}_j(y)U_y(x,t_j)) + \sum_{j=1}^{|v|}(E_j(y)U(x,v_j)),$$
$$\zeta(x,y) = \sum_{j=1}^{2}(F_j(y)U_x(x,t_j)) + \sum_{j=1}^{|v|}(E_j(y)U_x(x,v_j)),$$

and

$$\psi_i\left(x\right) = \prod_{\substack{k=1\\k\neq i}}^{|s|}\left(x-s_k\right), \quad \psi\left(x\right) = \prod_{k=1}^{|s|}\left(x-s_k\right), \quad \phi_j\left(x\right) = \prod_{\substack{k=1\\k\neq j}}^{|w|}\left(x-w_k\right), \quad \phi\left(x\right) = \prod_{k=1}^{|w|}\left(x-w_k\right),$$
$$\lambda_i = \sum_{j=1}^{|w|}\tfrac{1}{s_i-w_j} + 2\sum_{\substack{j=1\\j\neq i}}^{2}\tfrac{1}{s_i-s_j}, \quad F_j(y) = (1-(y-t_j)\Lambda_j)\hat{F}_j(y), \quad \bar{F}_j(y) = (y-t_j)\hat{F}_j(y),$$
$$\hat{F}_j(y) = \tfrac{\Psi_i^2(x)\Phi(x)}{\Psi_i^2(s_i)\Phi(s_i)},$$

and

$$\Psi_i\left(y\right) = \prod_{\substack{k=1\\k\neq i}}^{|t|}\left(y-t_k\right), \quad \Psi\left(y\right) = \prod_{k=1}^{|t|}\left(y-t_k\right), \quad \Phi_j\left(y\right) = \prod_{\substack{k=1\\k\neq j}}^{|v|}\left(y-v_k\right), \quad \Phi\left(y\right) = \prod_{k=1}^{|v|}\left(y-v_k\right),$$
$$\Lambda_i = \sum_{j=1}^{|v|}\tfrac{1}{t_i-v_j} + 2\sum_{\substack{j=1\\j\neq i}}^{2}\tfrac{1}{t_i-t_j}.$$

The above formulas are the result of a direct tensor product of the one-dimensional interpolant form, with the $\bar{H}_i(x)\bar{F}_j(y)$ terms removed. These terms are removed as the terms containing $\bar{H}_i(x)$ are those which interpolate the spatial derivative of the collocation solution in x, while the $\bar{F}_i(x)$ terms interpolate the spatial derivative with respect to y. As such, a term containing $\bar{H}_i(x)\bar{F}_j(y)$ would be interpolating the cross derivative, which is not expected to have sufficient convergence for use with the SCI and LOI schemes. A visual representation of what values are interpolated at which points is given below in Figures 3.1, 3.2, 3.3, for the case of the LOI interpolant with $p = 7$.

Furthermore, the way in which the SCI and LOI are used to calculate an error estimate must also be generalized to two dimensions. To calculate an error estimate on the domain $x_\alpha \leq x \leq x_\omega, y_\alpha \leq y \leq y_\omega$, with the assumption $p = q$, the following formula is used:

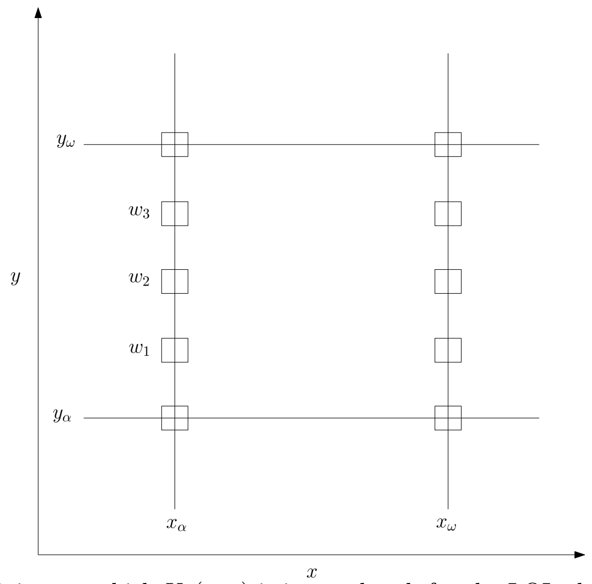


Figure 3.1: Points at which $U_x(x,y)$ is interpolated, for the LOI when $p = q = 7$.

$$E = \sqrt{\int_{x_\alpha}^{x_\omega} \int_{y_\alpha}^{y_\omega} \left(\frac{|U(x,y) - q(x,y)|}{atol + rtol|U(x,y)|} \right)^2 dy\, dx}, \tag{3.2}$$

where $q(x, y)$ is the interpolant being used for error estimation.

3.2 LOI2 Interpolant

Due to reasons which will be discussed later in thesis, it was desired that we have an alternate interpolant which does not make use of derivative values. The result of this is the LOI2 [21] which replaces a lower order solution when calculating an error estimate in the same way as the LOI, although the LOI2 aims to match the error polynomial of the higher order collocation solution, while the LOI matches that of the lower order solution it is replacing.

The LOI2 is a standard Lagrange interpolant with the interpolation points chosen such that the difference between the leading order term in the error of the higher order collocation solution and the interpolation error of the LOI2 interpolant is minimized in the L^2 norm. For a given degree of collocation solution the leading order term in the collocation error has a known form [3]. For a Lagrange interpolant, the interpolation error has a well known form which depends on the interpolation points. We choose the interpolation points for a given degree of collocation solution

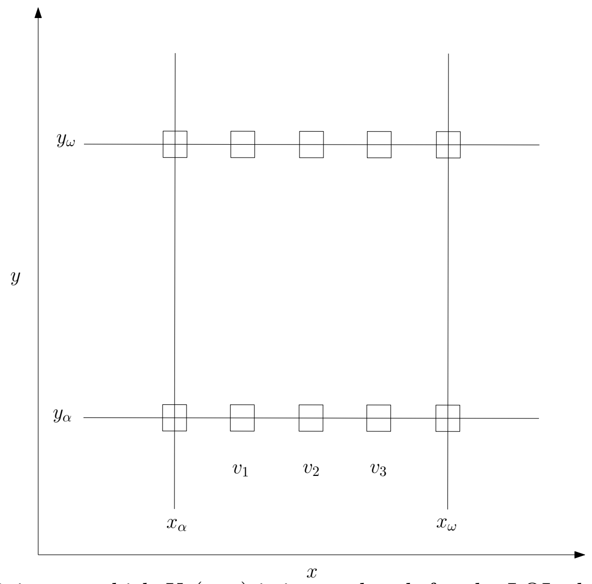


Figure 3.2: Points at which $U_y(x,y)$ is interpolated, for the LOI when $p = q = 7$.

by minimizing the L^2 norm between the collocation and interpolation errors. The interpolation points are required to include the end points of the subinterval, while the remaining $p-2$ points are determined by the Scilab *optim* function. The resulting points for degrees four through seven, on the interval $[0,1]$, are given in Figure 3.4.

3.2.1 LOI2 in One-dimension

Here we consider the non-time-dependent case. To evaluate the LOI2 in one dimension, the first Barycentric form of the Lagrange interpolant is used [6]. Given a collocation solution, $U(x)$, of degree, p, the desired point of evaluation, x, the subinterval such that $x \in [x_\alpha, x_\omega]$, and the points to interpolate, $w_i : i = 1, 2, \ldots, p$, the LOI2 can be evaluated as follows,

$$q\left(x\right) = L\left(x\right) \sum_{i=1}^{p} \frac{\mu_i}{\left(x - w_i\right)} U\left(w_i\right), \tag{3.3}$$

where

$$L\left(x\right) = \prod_{i=1}^{p} \left(x - w_i\right), \quad \mu_i = \prod_{\substack{k=1 \\ k \neq i}}^{p} \left(w_i - w_k\right).$$

In the PPDE case, in equation 3.3, we replace $q(x)$ with $q(x,t)$ and $U(w_i)$ is replaced by $U(w_i,t)$.

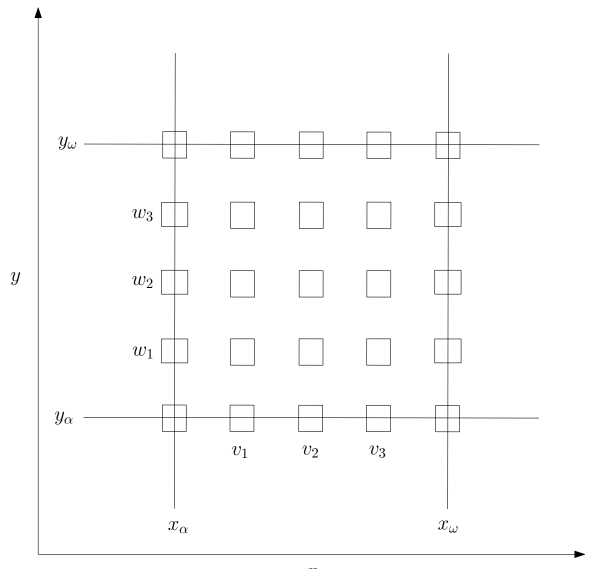


Figure 3.3: Points at which $U(x, y)$ is interpolated, for the LOI when $p = q = 7$.

Degree (p)	**4**	**5**	**6**	**7**
w_1	0	0	0	0
w_2	0.302331973224813	0.179424182143275	0.143465814421734	0.107235231524833
w_3	0.697668026790731	0.5	0.37051884018424	0.283676545203967
w_4	1	0.820575567392312	0.629481160240031	0.5
w_5		1	0.856534185886017	0.716323434111384
w_6			1	0.892764777407028
w_7				1

Figure 3.4: The relative values, w_i, on $[0, 1]$ at which the LOI2 interpolates the collocation solution for various values of p.

3.2.2 LOI2 in Two-dimensions

Here we consider the non-time-dependent case. To evaluate the LOI2 in two-dimensions, we use a Lagrange interpolant generalized to two dimensions based on a tensor product approach. Note that a Barycentric form is not used in this case, although it would be straightforward to do so. Given a collocation solution, $U(x, y)$, of degree p in x, and degree q in y, a desired point of evaluation (x, y), subintervals such that $x \in [x_\alpha, x_\omega]$ and $y \in [y_\alpha, y_\omega]$, and the points to interpolate, $w_i : i = 1, 2, \ldots, p$, $v_i : i = 1, 2, \ldots, q$, the LOI2 can be evaluated as follows,

$$q\left(x, y\right) = \sum_{i=1}^{p} \sum_{j=1}^{q} G_i\left(x\right) E_j\left(y\right) U\left(w_i, v_j\right), \tag{3.4}$$

where

$$G_i(x) = \frac{\phi_i(x)}{\phi_i(w_i)}, \quad E_j(y) = \frac{\Phi_j(y)}{\Phi_j(v_j)},$$

and

$$\phi_i = \prod_{\substack{k=1 \\ k \neq i}}^{p} (x - w_i), \quad \Phi_j = \prod_{\substack{k=1 \\ k \neq j}}^{q} (y - v_j).$$

Chapter 4

Implementation

In this chapter we provide an overview of the Scilab software we have developed. The source code for the Scilab scripts is given in the Appendix of this thesis.

4.1 Overview of Software

As opposed to a single piece of software for each of the four problem classes, numerous Scilab scripts have been created so that common functions may be reused across the problem classes. These scripts are divided into three main levels: common, dimension, and problem class. The hierarchy of these classes can be seen in Figure 4.1. The common level contains functions that are required in all of the problem classes, while the dimension level contains the functions required for computing a collocation solution and error estimate in either one or two spatial dimensions. Finally the problem class contains the functions that are required for each problem class. A driver that loads the required scripts is provided for each problem class.

4.1.1 Common Scripts

The common level contains two scripts, core.sci, which contain the common functions for calculating a collocation solution, and err.sci, which contains the common functions used for calculating an error estimate. All of the functions within these scripts are internal functions and are not intended

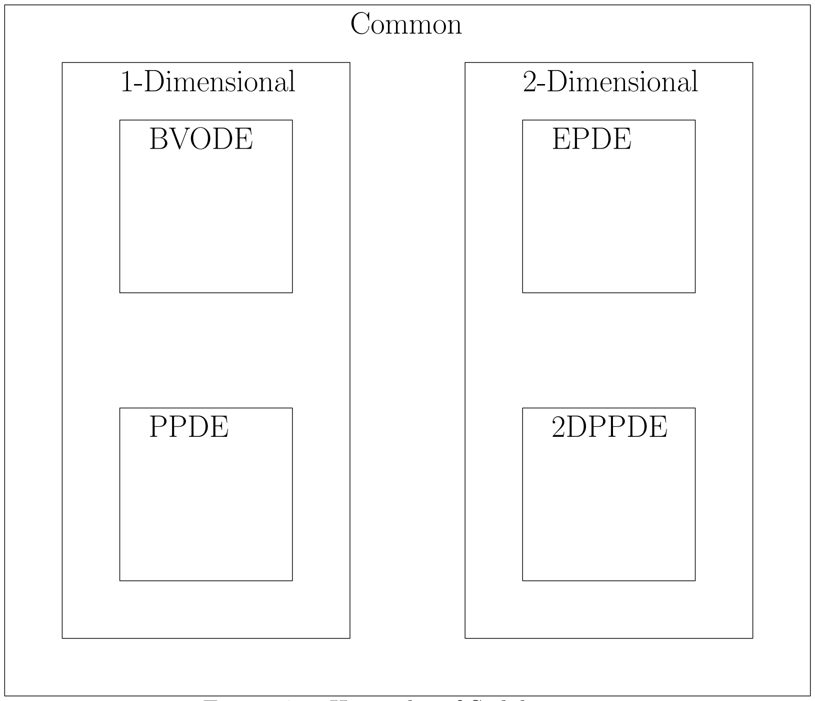


Figure 4.1: Hierarchy of Scilab scripts

to be called by the user. Within core.sci most of the functions are for calculating the B-splines, including Fortran source code for the B-spline package, which is compiled and linked upon execution of the script. The functions within err.sci provide access to the interpolation points used for the various error estimation schemes, as well as the points and weights required for Gaussian quadrature. This script also contains the supporting functions which are used to evaluate the Hermite-Birkhoff interpolants.

4.1.2 Dimensional Scripts

As with the common scripts, the dimensional level is also broken into the scripts used for calculating a collocation solution and the scripts for calculating an error estimate. In the one spatial dimension classes these are core1D.sci and err1D.sci, while in the two spatial dimensions classes these are core2D.sci and err2D.sci. The core scripts within this level handle the evaluation of the collocation solution. The error scripts within this level contain functions to evaluate Hermite-Birkhoff, Lagrange, and Barycentric-Lagrange interpolants, along with functions to evaluate these interpolants in order to implement one of the error estimation schemes introduced in Chapters 2 and 3. Furthermore, these scripts contain functions to perform Gaussian quadrature and return an error estimate using the error expressions presented in Chapters 2 and 3.

4.1.3 Problem Class Scripts

This level will contain two scripts for each problem class, the core script which contains functions to calculate a collocation solution, and the problems script which contains a definition of the problem(s) to be solved. The core script contains functions which represent the system of equations which will be solved for the coefficients of the collocation solution, as well as a collocate function to be called by the user which will calculate a collocation solution of a specified degree with a specified number of subintervals.

4.2 Collocation Procedure

Within this subsection we will step through the process of how a collocation solution is computed for each of the problem classes. We first outline the global variables which are used within the problem classes. All problem classes have the following global variables which must be set by the user before calculating a collocation solution: double $atol$, the absolute error tolerance, double $rtol$, the relative error tolerance, and integer $probNum$, which specifies to the script which problem to use. There are also global variables which will be set by the scripts, including *coeffs*, a vector containing the B-spline coefficients of the collocation solution. There are also variables associated with specifying the collocation method on the x domain: degree of collocation solution, integer p, number of subintervals, integer N, the lower and upper bounds of the spatial domain, integers A and B, the mesh points, double $meshX$, a vector of size $N + 1$, the B-spline knots, double $knotsX$, a vector, and collocation points, double $colX$, a matrix where $colX(i, j)$ represents the j^{th} collocation point on the i^{th} subinterval. For the two spatial dimension problem classes we will have an equivalent set of variables for the y domain; these will be q, M, C and D, $meshY$, $knotsY$, and $colY$. Furthermore in the one spatial dimension case we have $NCPTS = N(p - 1) + 2$, or in the two spatial dimension case we have $NCPTX = N(p - 1) + 2$, $NCPTY = M(q - 1) + 2$, and

$NCPTS = NCPTX \times NCPTY.$

4.2.1 BVODE

A problem is defined in this class by the following three functions.

$$y = g(x, u, u_x, u_{xx}),$$

$$y = bndxa(u(A)),$$

$$y = bndxb(u(B)),$$

where $g(\dots) = f(\dots) - u_{xx}$, and $f(\dots)$ represents the right hand side of the DE, (2.1). The *bndxa* and *bndxb* functions return the boundary conditions at the boundaries of the spatial domain.

To calculate a collocation solution for this problem case, one must first set the global variables outlined at the beginning of this subsection and then call the provided collocate function. The collocate function takes as input the degree of the collocation solution, p, and the number of subintervals, N. The function is called as follows:

$$info = collocate(p, N).$$

collocate sets up the collocation and boundary conditions, then uses *fsolve* to solve the systems for the B-spline coefficients. The *fsolve* function is a non-linear system solver with error control that is included with Scilab, which returns the integer flag *info*.

The *collocate* function will begin by setting the spatial mesh and B-spline knot points using the one-dimensional common *buildMesh* function. The *buildMesh* function requires no input, and will return the spatial mesh as well as setting the global knots variable. After this, the collocation points are set by iterating over the subintervals and calling the common *getGaussPoints* function. The calling syntax is as follows.

$$pts = getGaussPts(l, r, numP),$$

where l and r are the lower and upper bounds of the subinterval, and $numP$ is the number of Gauss points requested which, is $p - 1$ in this case. The vector *pts* is returned containing $numP$ Gauss points mapped onto the interval $[l, r]$.

Then, after this initialization, the *fsolve* function can be called to calculate the B-spline coefficients of the collocation solution. The calling syntax for fsolve is as follows,

$$[\mathit{coeffs}, v, \mathit{info}] = \mathit{fsolve}(\mathit{fg}, \mathit{res}, \mathit{fjac}, \mathit{atol}),$$

where *fg* is a vector of length $NCPTS$ containing an initial guess for the B-spline coefficients. The function res represents the system of equations. The user provided optional function, *fjac*, provides the Jacobian of the system. *atol* is the previously mentioned tolerance. The returned *coeffs* is a vector of length $NCPTS$ which contains the B-spline coefficients of the collocation solution. The returned vector v is treated as a dummy variable, while the integer *info* contains information from *fsolve*, which will allow the user to know if the solver was unable to converge or meet the specified tolerance.

The residual function takes an array of B-spline coefficients as its only input, which are then used for any required evaluations of the collocation solution. The residual function returns a matrix of size equal to the coefficients containing the residuals of the boundary and collocation conditions. The residuals of the boundary conditions are calculated by the *bndxa* and *bndxb* functions. The residual of the collocation conditions are calculated by evaluating the function g at each collocation point. The form of the residual function is as follows.

```
function $y = residual(\mathit{coeff})$
    $y(1) = bndxa(u(A))$
    for $i = 1 : N$
        for $j = 1 : (p-1)$
            $x = colX(i, j)$
            $y((i-1)(p-1) + j + 1) = g(x, u(x), u_x(x), u_{xx}(x))$
        end
    end
    $y(NCPTS) = bndxb(u(B))$
endfunction
```

4.2.2 PPDE

A problem is defined in this class by the following four functions:

$$y = g(t, x, u, u_x, u_{xx}, u_t),$$

$$y = bndxa(t, u, u_x),$$

$$y = bndxb(t, u, u_x),$$

$$y = uinit(x),$$

where $g(\dots) = f(\dots) - u_t$, and $f(\dots)$ represents the right hand side of the DE, (2.6). The function $bndxa(t, u, u_x)$ represents the boundary condition at $x = A$ evaluated at time t with u and u_x evaluated at $x = A$. The function $bndxb(t, u, u_x)$ is similarly defined except with $x = B$. The function $uinit(x)$ gives the initial solution at point x, and time, t_0.

In this problem class, the collocate function has four inputs: the degree of collocation solution, p, number of subintervals, N, the beginning time, t_0, and the output time, t_f. As in the BVODE case, the *collocate* function begins by calculating the mesh and collocation points, and the size of the system of equations which will be solved. The collocate function then calculates the initial values for the B-spline coefficients based on the initial solution $u(x, t_0)$, which is defined by *uinit.* This is done by projecting the initial solution onto the B-spline basis. This is done by requiring that the collocation solution equal the initial solution at the two end points of the spatial domain, and at the collocation points within each subinterval. The *fsolve* function is used to solve for these initial B-spline coefficients.

The initial collocation solution together with the right hand side of the PDE is then used to obtain the temporal derivative of the collocation solution at t_0, which is then projected onto the B-spline basis functions to obtain the initial values for the derivatives of the B-spline coefficients with respect to time, at t_0. The *fsolve* function is again used to calculate the initial temporal derivatives of the B-spline coefficients.

The system which *fsolve* uses to evaluate the boundary and collocation conditions is represented by a residual function. The residual function takes three inputs: the current time, t, the B-spline coefficients, c, and their temporal derivatives, c_t, all of which are used to perform any required evaluations of the collocation solution within the residual function. The output of the residual is a matrix of size $NCPTS$ where the first and last elements are the evaluations of the boundary conditions at the current time t, with the collocation solution defined by the given B-spline coefficients. The remaining elements are the collocation conditions, ordered in increasing magnitude of x. The general form of this residual function is as follows:

```
function y = residual($t$, $c$, $c_t$)
    $y(1) = bndxa(t, u, u_x)$
    for $i = 1 : N$
        for $j = 1 : (p-1)$
            $x = colX(i, j)$
            $y((i-1)(p-1)+j+1) = g(t, x, u(x), u_x(x), u_{xx}(x), u_t(x))$
        end
    end
    $y(NCPTS) = bndxb(t, u, u_x)$
endfunction,
```

where t is the current point in the temporal domain, c and c_t are the B-spline coefficients and their temporal derivatives at time t. These inputs are used for any required evaluation of the collocation solution.

After the initial B-spline coefficients and their temporal derivatives have been computed, the Scilab *daskr* function is called (this is a Scilab function which links to the Fortran code of *daskr*, [8], [7]) with the following parameters: a matrix containing the initial B-spline coefficients and their derivatives, the initial time, the output time, the tolerance, which is specified as a tenth of the global *atol* value, the residual function, and the *info* list, which controls the operation of *daskr*. *daskr* is a DAE solver featuring error control and root finding. The error control within *daskr* controls the temporal error associated with the calculation of the B-spline coefficients. The root finding feature is not required in our case, and is ignored.

4.2.3 EPDE

A problem is defined in this class by the following five functions.

$$z = g(x, y, u_x, u_y, u_{xx}, u_{yy}),$$

$$z = bndxa(y, u, u_x, u_y),\ z = bndxb(y, u, u_x, u_y),\ z = bndyc(x, u, u_x, u_y),\ z = bndyd(x, u, u_x, u_y),$$

where $g(\dots) = f(\dots) - u_{xx} - u_{yy}$, and $f(\dots)$ represents the right hand side of the DE, (2.13). The function *bndxa* represents the boundary condition along $x = A$, and the other boundary conditions follow this same pattern.

The collocate function for the EPDE problem case takes in two parameters, the degree of the collocation solution and the number of subintervals of the spatial meshes in x and y. These inputs can be vectors, in which case the first elements will be used for the degree and number of subintervals in x, while the second elements will be used for those in y. These inputs can also be scalars, in which case the degree and number of subintervals will be the same in x and in y. Then the mesh points, collocation points, and knot sequence are calculated. Due to the rapidly growing computational costs in the two-dimensional collocation solutions, all B-spline evaluations are saved for later use.

In this case, for simplicity, we use B-spline initial coefficients of zero and then call the *fsolve* function. The *fsolve* function takes as input, the residual function, an optional function which estimates the Jacobian, and the absolute tolerance. The residual function is ordered as outlined in Figure 2.13.

4.2.4 2DPPDE

A problem is defined in this class by the following six functions.

$$z = g(t, x, y, u_x, u_y, u_{xx}, u_{yy}),$$

$$z = bndxa(y, t, u, u_x, u_y), \quad z = bndxb(y, t, u, u_x, u_y),$$

$$z = bndyc(x, t, u, u_x, u_y), \quad z = bndyd(x, t, u, u_x, u_y),$$

$$z = uinit(x, y),$$

where $g(\dots) = f(\dots) - u_t$, and $f(\dots)$ represents the right hand side of the DE, (2.20). The boundary conditions correspond to the point along their line at which they are evaluated. For example, *bndxa* returns the value of the boundary condition at time t, $x = A$, and y. Finally the $\mathit{uinit}(x, y)$ function returns the value of the initial solution at point (x, y), and at time t_0.

For this problem class the collocate function takes four inputs. These are the degrees of the B-spline bases, the number of subintervals in the x and y domains, the initial time, and the output time. The degree and number of subintervals are treated in the same manner as for the EPDE collocate function. Then the mesh points, collocation points, and knot sequence in x and y are calculated. As is done in the EPDE case, the B-spline evaluations are saved for re-use.

The next step in the collocate function is setting the initial values, starting with the B-spline coefficients. This is done in an approach that is the obvious generalization of the approach used for the PPDE case. The initial temporal derivatives of the B-spline coefficients are also obtained using the same approach as in the PPDE case.

The residual function which represents the system in this case takes as input the time, the B-spline coefficients, and their temporal derivatives. These are used to evaluate the boundary and collocation conditions which are ordered as outlined in Figure 2.14.

After these initial values are set, the *daskr* function is called with the following input: a matrix containing the initial B-spline coefficients and their temporal derivatives, the initial time, the output time, the tolerance, which is specified as a tenth of the global absolute tolerance, the residual function, and the *info* list which controls the operation of *daskr*.

Chapter 5

Numerical Results

This chapter contains results investigating experimentally the order of convergence of collocation solutions and the interpolants introduced within this thesis; the latter are also evaluated for the accuracy of their error estimates. We will begin by introducing the methods used for these tests, as well as the format in which the results are presented. For the time-dependent problem classes, the output time used is the end of the temporal domain as introduced in Chapter 2.

The sections of this chapter are divided by problem class and within each of these problem class sections there are two subsections, one for convergence results, and one for error estimation results. The convergence results subsections will contain a table for each of the following: collocation solution, SCI, LOI, and LOI2, which are labelled as GE, $SCI-E$, $LOI-E$, $LOI2-E$. These tables contain convergence results for various combinations of degree (p), and subintervals ($nint = N = M$). In the case of the SCI table and a one-dimensional problem class, this error would be,

$$(SCI - E)_n = \max_{A \le x \le B} \left(|SCI(x) - u(x)| \right), \tag{5.1}$$

where $SCI(x)$ evaluates the SCI calculated from a collocation solution with $nint = N = 2^n$ subintervals and $u(x)$ represents the true solution. For a two-dimensional problem class, this error would instead be,

$$(SCI - E)_n = \max_{\substack{A \le x \le B \\ C \le y \le D}} \left(|SCI(x,y) - u(x,y)| \right), \tag{5.2}$$

where the SCI is calculated from a collocation solution with $nint = N = M = 2^n$.

It is important to note that these errors are not computed exactly, but instead are approximated by evaluating the error at 1000 evenly spaced points across the spatial domain and then taking the maximum of those evaluations. In the two dimensional cases, we instead evaluate at the Cartesian product of 100 evenly spaced points within each spatial domain for a total of 10000 evaluation points.

Adjacent to the error columns there is also a *rate* column which will give the rate of convergence. The rate value is calculated with the following formula,

$$rate_n = log_2((SCI - E)_{n-1}/(SCI - E)_n), \tag{5.3}$$

where $n \geq 2$. As n approaches 5 or $nint$ approaches 32 it is expected that the *rate* will begin to fall below what is expected. This is caused by errors introduced from the solvers used (*fsolve* or *daskr*) to calculate the B-spline coefficients, resulting in a lower bound on the error which can be achieved. Furthermore for values of n close to 1 or $nint$ close to 2, we may not see the expected rate either as the subinterval size may not be sufficiently small for the lower order terms of the error expression to be dominant. The region in between these two phenomenon is the asymptotic region, where the errors behave as expected for a change of the subinterval size.

The error estimation subsections differ slightly in one and two dimensions. These subsections will contain results from the SCI, LOI and LOI2 schemes, as well as a scaled LOI and LOI2. As the LOI and LOI2 provide an error estimate for a collocation solution of an order less than the returned collocation solution, the LOI and LOI2 error estimates are multiplied by the subinterval size, h, so that the leading order terms are of the same order. These scaled versions will be labelled as LOI-S, and LOI2-S, for the scaled versions of the LOI and LOI2.

We will first discuss the one-dimensional classes. Figure 5.1 shows the error estimates produced from the various interpolants, as well as the true error of the collocation solution, for one of the test cases. Note that the error estimates are plotted per subinterval, and that the points on the plots correspond to the error estimate on each subinterval in the spatial domain. The lines connecting these points are plotted to allow easier visualization.

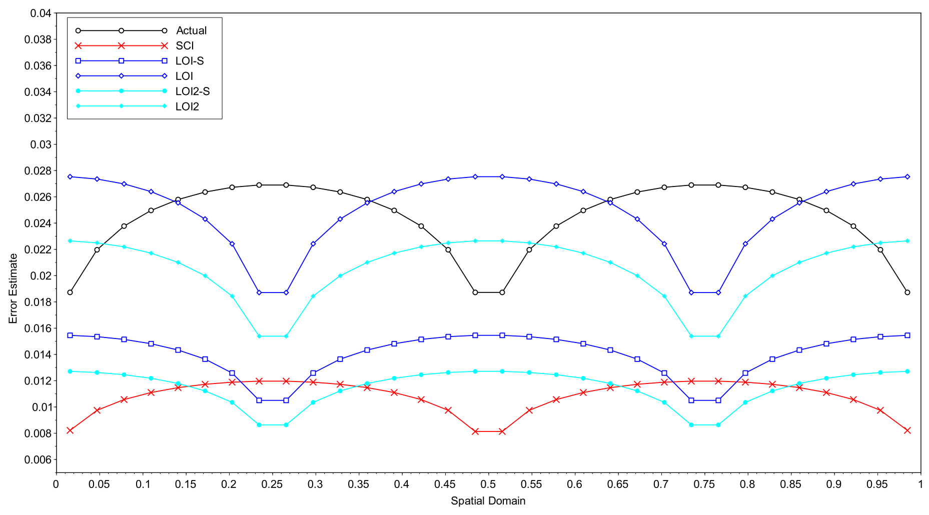


Figure 5.1: Actual and estimated subinterval errors for a collocation solution to PPDE problem 2 at $t = 1$ with $p = 5$ and $nint = 32$. SCI, LOI, and LOI2 are the SCI, LOI and LOI2 error estimates. LOI-S and LOI2-S are the scaled LOI and LOI2 error estimates.

When evaluating an error estimation method, we are looking to see if it will provide an error estimate that is at least of the right order of magnitude. This idea is also phrased as desiring the error estimate to be bounded by a small multiple of the true error. Figure 5.1 does not allow one to easily see if the error estimates are of the right order of magnitude, and therefore the error estimates will instead be presented as

$$log_{10}(\text{estimated error/actual error}), \tag{5.4}$$

where the actual error is the L^2 norm of the difference between collocation solution and the true solution. The L^2 norm is used as the SCI, LOI, and LOI2 have been developed for use within an L^2 error estimate. Figure 5.2 shows an example of this type of plot, for the same example that was considered in Figure 5.1. The left subplot shows the error estimate in terms of equation (5.4), which we will call the error log ratio, across the spatial domain, while the right subplot shows the occurrence rate of the error log ratios. An error log ratio of -1 corresponds to an error estimate of one order less than the true error, while 1 would be one order above, and 0 would correspond to the true error. Thus an accurate error estimate will have an error log ratio close to 0. The right subplot is created by dividing the interval $[min(-1, \text{error log ratios}), max(1, \text{error log ratios})]$ into subintervals of width 0.1, and then counting how many error log ratios fall within each of these subintervals. These counts are then divided by the total number of error log ratios to calculate the relative occurrence rate of each of the error log ratio intervals. In Figure 5.2 one can look at the SCI in the left subplot and see that it has a consistent error log ratio of about -0.3; this can be seen in the right subplot by the single point at occurrence rate 1 around the error log ratio of -3. With the other interpolants one can see that they are less consistent by noting on the right subplot that they have an occurrence plot that is more spread out. The plots of this type for the one-dimensional problem classes all have $nint = 32$, and span degrees 4 through 7 for each of the test problems.

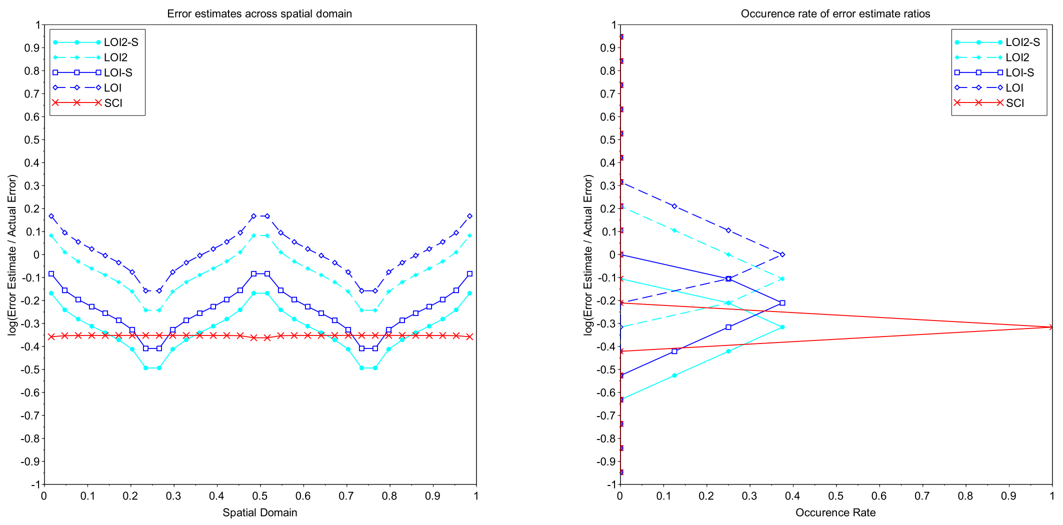


Figure 5.2: The log of the ratio of estimated error to actual error for each subinterval, and occurrence rate of error log ratios for a collocation solution to PPDE problem 2 at $t = 1$ with $p = 5$ and $nint = 32$.

An example of an error estimation plot for the two dimensional problem classes can be seen in Figure 5.3. The top left subplot shows the error log ratios for the SCI over the spatial domain, while the bottom left shows the error log ratios for the LOI and LOI2. The top right subplot shows the error log ratios for the scaled LOI and LOI2, while the bottom right shows the occurrence rates of the error log ratios for each case. For the three-dimensional plots it is important to note that the error estimates are computed for each subrectangle, and this error estimate is plotted at the center of the subrectangle. The surface plotted is the interpolation of these points, and serves only to show any trends in the error estimates. The plots given for the two-dimensional problem classes all have $nint = 16$ or $N = M = 16$, and span degrees 4 through 7, with $p = q$, for each of the test problems.

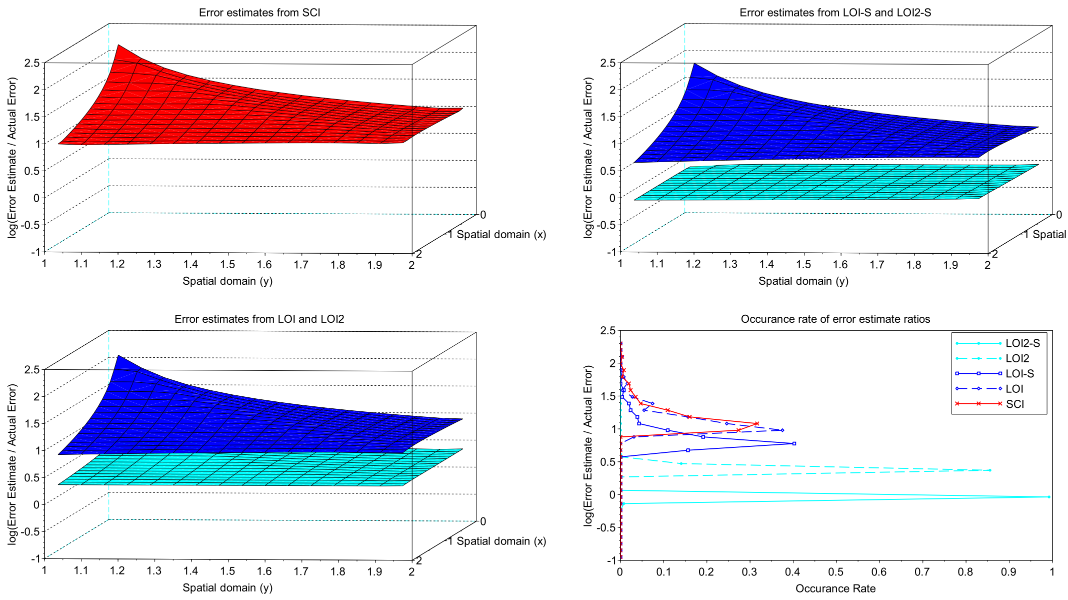


Figure 5.3: Error estimate results for a collocation solution to EPDE problem 2 with $p = q = 4$ and $nint = 16$.

5.1 BVODEs

5.1.1 Convergence Results

Within this section the expected convergence rates are $p + 1$ for GE, $p + 2$ for $SCI - E$, and p for LOI and $LOI2$. From Table 5.1 we see that the collocation solution does have the expected convergence rate of $p+1$ for most of the entries, apart from the rates at $nint = 32$ for $p = 6, 7$ where we see the error associated with the accuracy of the *fsolve* computations dominate as the errors reach a minimum of about 10^{-13}. The SCI, LOI, and LOI2 results are very similar, as can be seen in Tables 5.2, 5.3, 5.4, respectively, where most of the entries agree with their expected convergence rates apart from some of the entries for higher degrees and larger $nint$ values.

	Degree (p)							
	4		5		6		7	
nint	GE	$rate$	GE	$rate$	GE	$rate$	GE	$rate$
Problem 1								
2	9.51×10^{-3}		5.31×10^{-4}		2.83×10^{-5}		1.25×10^{-6}	
4	4.08×10^{-4}	4.54	1.26×10^{-5}	5.4	3.09×10^{-7}	6.51	6.87×10^{-9}	7.5
8	1.49×10^{-5}	4.78	2.39×10^{-7}	5.71	2.85×10^{-9}	6.76	3.19×10^{-11}	7.75
16	4.99×10^{-7}	4.9	4.13×10^{-9}	5.86	2.41×10^{-11}	6.88	1.60×10^{-13}	7.64
32	1.60×10^{-8}	4.96	6.72×10^{-11}	5.94	6.09×10^{-13}	5.31	3.06×10^{-13}	-0.94
Problem 2								
2	2.54×10^{-3}		1.04×10^{-4}		3.73×10^{-6}		1.26×10^{-7}	
4	9.77×10^{-5}	4.7	2.16×10^{-6}	5.59	3.77×10^{-8}	6.63	6.44×10^{-10}	7.61
8	3.39×10^{-6}	4.85	3.89×10^{-8}	5.8	3.35×10^{-10}	6.81	2.89×10^{-12}	7.8
16	1.12×10^{-7}	4.92	6.51×10^{-10}	5.9	2.81×10^{-12}	6.9	5.64×10^{-14}	5.68
32	3.57×10^{-9}	4.97	1.05×10^{-11}	5.96	2.20×10^{-13}	3.67	1.51×10^{-13}	-1.42

Table 5.1: Global errors (GE) and convergence rates of collocation solutions for varying BVODE problem and degree.

	Degree (p)							
	4		5		6		7	
nint	$SCI-E$	$rate$	$SCI-E$	$rate$	$SCI-E$	$rate$	$SCI-E$	$rate$
Problem 1								
2	4.50×10^{-3}		1.84×10^{-4}		8.36×10^{-6}		3.28×10^{-7}	
4	1.09×10^{-4}	5.36	2.00×10^{-6}	6.52	4.69×10^{-8}	7.48	9.21×10^{-10}	8.48
8	2.19×10^{-6}	5.64	2.01×10^{-8}	6.64	2.18×10^{-10}	7.75	2.16×10^{-12}	8.74
16	3.90×10^{-8}	5.81	1.78×10^{-10}	6.82	8.82×10^{-13}	7.95	9.33×10^{-14}	4.53
32	6.50×10^{-10}	5.91	1.49×10^{-12}	6.9	5.20×10^{-13}	0.76	3.07×10^{-13}	-1.72
Problem 2								
2	4.62×10^{-4}		9.90×10^{-6}		2.16×10^{-7}		5.83×10^{-9}	
4	8.85×10^{-6}	5.71	9.13×10^{-8}	6.76	1.04×10^{-9}	7.69	1.39×10^{-11}	8.71
8	1.51×10^{-7}	5.87	7.68×10^{-10}	6.89	4.50×10^{-12}	7.86	3.38×10^{-14}	8.69
16	2.47×10^{-9}	5.94	6.20×10^{-12}	6.95	1.27×10^{-13}	5.14	5.02×10^{-14}	-0.57
32	3.94×10^{-11}	5.97	5.33×10^{-14}	6.86	2.07×10^{-13}	-0.7	1.53×10^{-13}	-1.61

Table 5.2: SCI errors ($SCI - E$) and convergence rates for varying BVODE problem and degree.

	Degree (p)							
	4		5		6		7	
nint	$LOI-E$	$rate$	$LOI-E$	$rate$	$LOI-E$	$rate$	$LOI-E$	$rate$
Problem 1								
2	1.36×10^{-1}		7.94×10^{-3}		5.61×10^{-4}		2.73×10^{-5}	
4	1.33×10^{-2}	3.36	3.66×10^{-4}	4.44	1.27×10^{-5}	5.46	3.02×10^{-7}	6.5
8	1.04×10^{-3}	3.68	1.39×10^{-5}	4.71	2.40×10^{-7}	5.73	2.81×10^{-9}	6.75
16	7.27×10^{-5}	3.84	4.81×10^{-7}	4.86	4.13×10^{-9}	5.86	2.40×10^{-11}	6.87
32	4.80×10^{-6}	3.92	1.58×10^{-8}	4.93	6.74×10^{-11}	5.94	4.00×10^{-13}	5.91
Problem 2								
2	6.58×10^{-2}		2.24×10^{-3}		1.06×10^{-4}		3.65×10^{-6}	
4	5.42×10^{-3}	3.6	9.14×10^{-5}	4.61	2.17×10^{-6}	5.61	3.72×10^{-8}	6.62
8	3.90×10^{-4}	3.8	3.28×10^{-6}	4.8	3.89×10^{-8}	5.8	3.33×10^{-10}	6.81
16	2.62×10^{-5}	3.9	1.10×10^{-7}	4.9	6.52×10^{-10}	5.9	2.80×10^{-12}	6.89
32	1.70×10^{-6}	3.95	3.54×10^{-9}	4.95	1.05×10^{-11}	5.96	1.64×10^{-13}	4.09

Table 5.3: LOI errors ($LOI-E$) and convergence rates for varying BVODE problem and degree.

	Degree (p)							
	4		5		6		7	
nint	$LOI2-E$	$rate$	$LOI2-E$	$rate$	$LOI2-E$	$rate$	$LOI2-E$	$rate$
Problem 1								
2	2.83×10^{-2}		2.42×10^{-3}		1.78×10^{-4}		9.88×10^{-6}	
4	2.54×10^{-3}	3.47	1.14×10^{-4}	4.41	3.91×10^{-6}	5.51	1.08×10^{-7}	6.52
8	1.92×10^{-4}	3.73	4.34×10^{-6}	4.71	7.28×10^{-8}	5.75	9.96×10^{-10}	6.75
16	1.32×10^{-5}	3.86	1.49×10^{-7}	4.86	1.24×10^{-9}	5.87	8.45×10^{-12}	6.88
32	8.54×10^{-7}	3.95	4.91×10^{-9}	4.93	1.98×10^{-11}	5.98	3.34×10^{-13}	4.66
Problem 2								
2	1.25×10^{-2}		7.11×10^{-4}		3.25×10^{-5}		1.30×10^{-6}	
4	1.00×10^{-3}	3.64	2.87×10^{-5}	4.63	6.59×10^{-7}	5.63	1.32×10^{-8}	6.62
8	7.09×10^{-5}	3.82	1.02×10^{-6}	4.81	1.17×10^{-8}	5.81	1.18×10^{-10}	6.81
16	4.71×10^{-6}	3.91	3.41×10^{-8}	4.91	1.96×10^{-10}	5.91	1.01×10^{-12}	6.86
32	3.01×10^{-7}	3.97	1.10×10^{-9}	4.95	3.07×10^{-12}	5.99	1.55×10^{-13}	2.7

Table 5.4: LOI2 errors ($LOI2-E$) and convergence rates for varying BVODE problem and degree.

5.1.2 Error Estimation

For the BVODE problem class the SCI performs very well, providing consistent error log ratios in most of the results presented. In Figure 5.11 we see the greatest underestimate of all the interpolants from the SCI, with an error log ratio of about -0.4 and an occurrence rate of about 0.15, which remains within a small multiple of the true error. The SCI does not overestimate the error for these BVODE examples, and tends to produce a consistent error log ratio between -0.3 and 0.

The LOI and LOI2 are quite consistent for the BVODE problem class, although they seem to be impacted by the choice of p. In Figures 5.4 and 5.8, where $p = 4$, the LOI and LOI2 have error log ratios mostly around 0.5 and 0.4 respectively, while their scaled versions have error log ratios around 0.3 and 0.1, respectively. For $p = 7$, in Figures 5.7 and 5.11, the LOI and LOI2 error log ratios tend towards -0.1 and -0.2, while the scaled versions have error log ratios around -0.2 and -0.3. Apart from this downward tendency as p increases, the error log ratios remain within $[-0.5, 0.5]$ and thus are a small multiple of the true error.

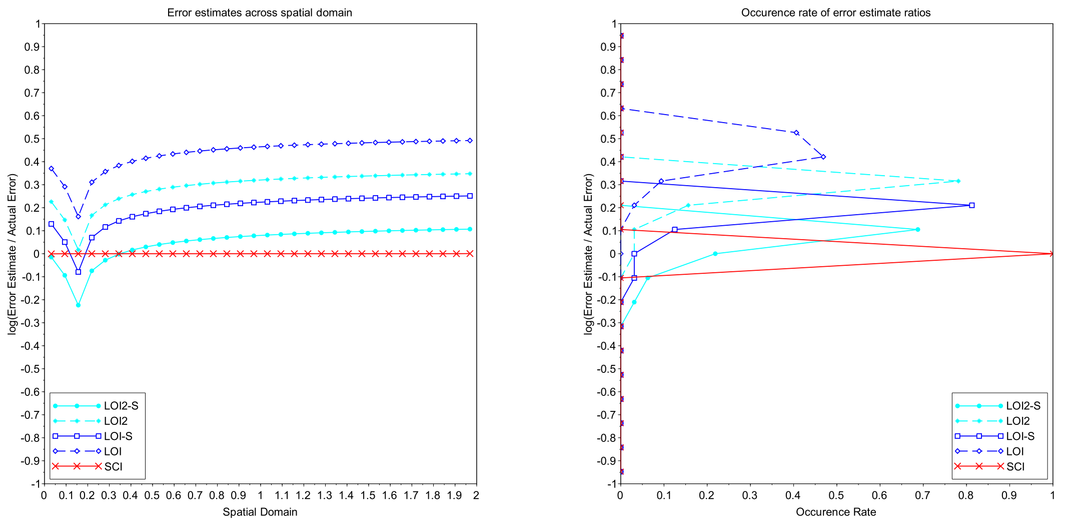


Figure 5.4: The log of the ratio of estimated error to actual error for each subinterval for a collocation solution to BVODE problem 1 with $p = 4$ and $nint = 32$.

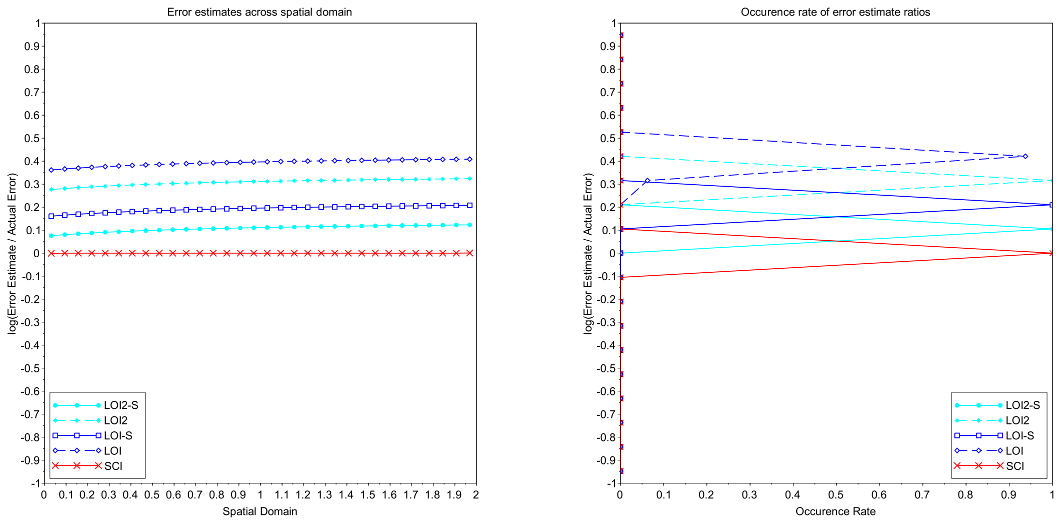


Figure 5.5: The log of the ratio of estimated error to actual error for each subinterval for a collocation solution to BVODE problem 1 with $p = 5$ and $nint = 32$.

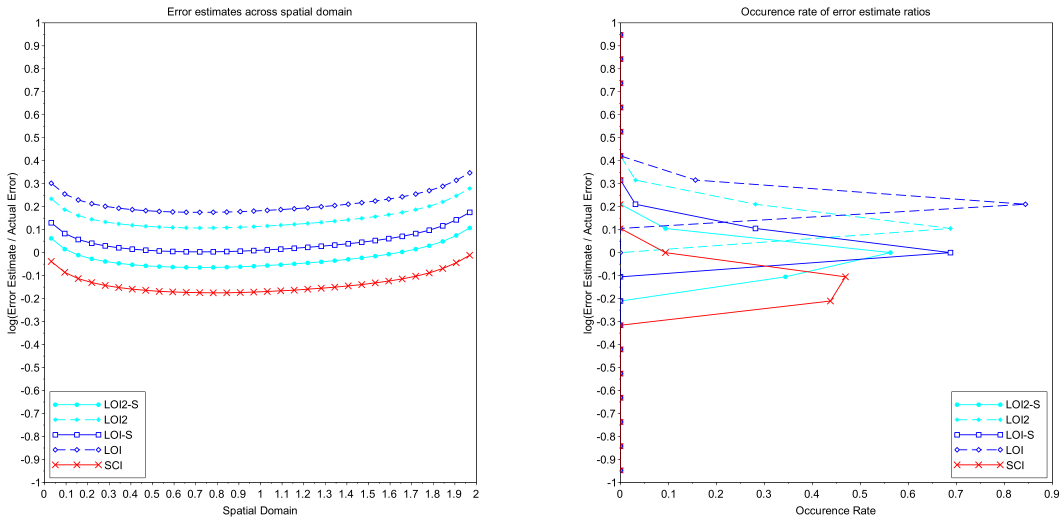


Figure 5.6: The log of the ratio of estimated error to actual error for each subinterval for a collocation solution to BVODE problem 1 with $p = 6$ and $nint = 32$.

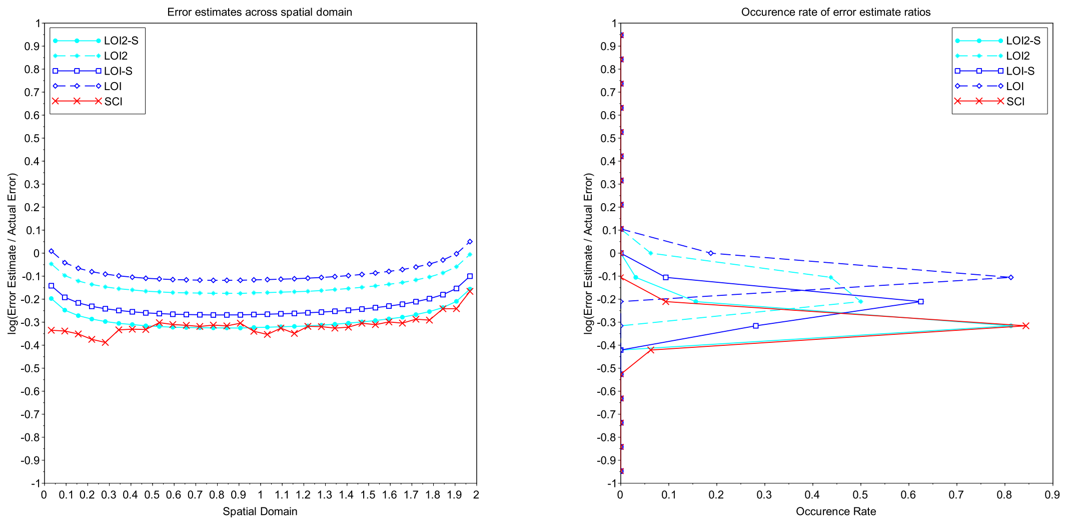


Figure 5.7: The log of the ratio of estimated error to actual error for each subinterval for a collocation solution to BVODE problem 1 with $p = 7$ and $nint = 32$.

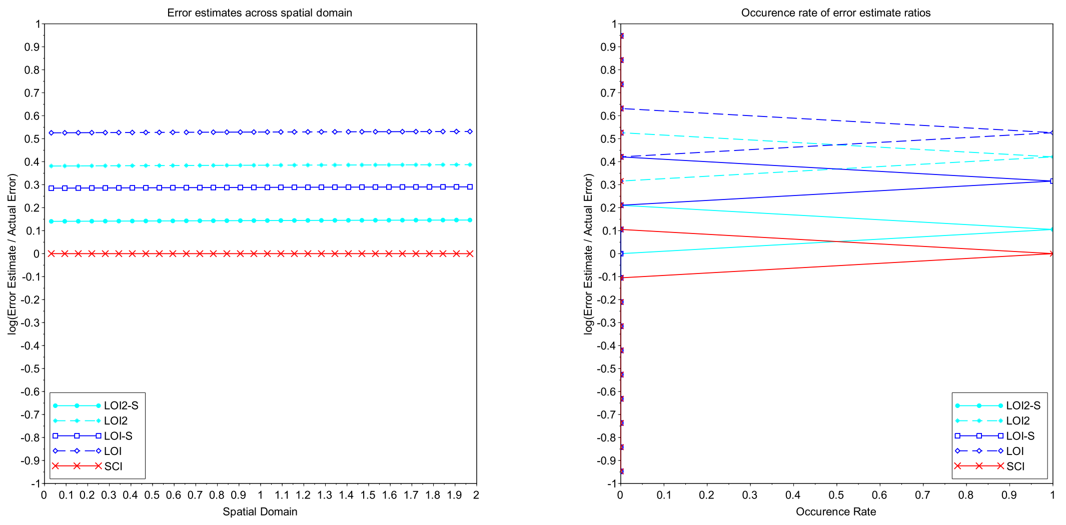


Figure 5.8: The log of the ratio of estimated error to actual error for each subinterval for a collocation solution to BVODE problem 2 with $p = 4$ and $nint = 32$.

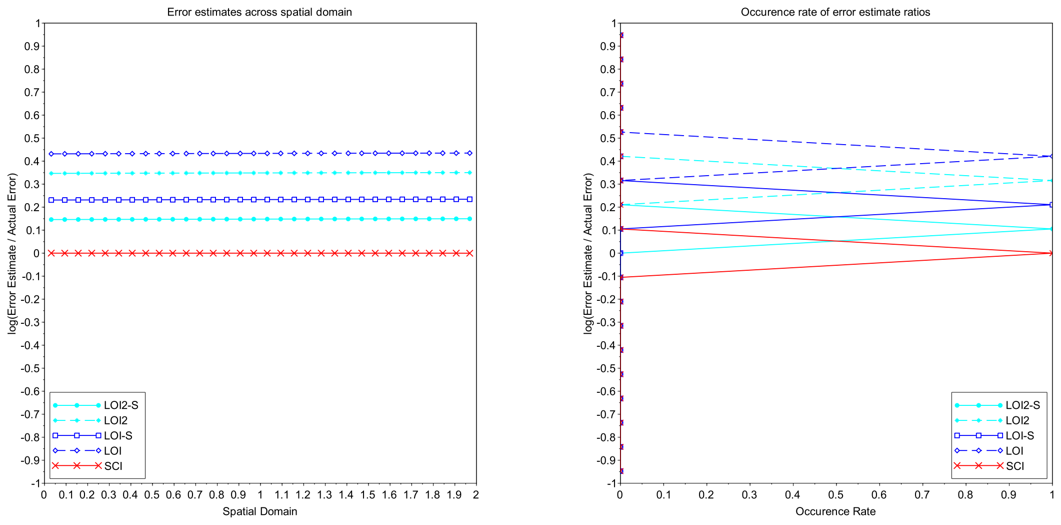


Figure 5.9: The log of the ratio of estimated error to actual error for each subinterval for a collocation solution to BVODE problem 2 with $p = 5$ and $nint = 32$.

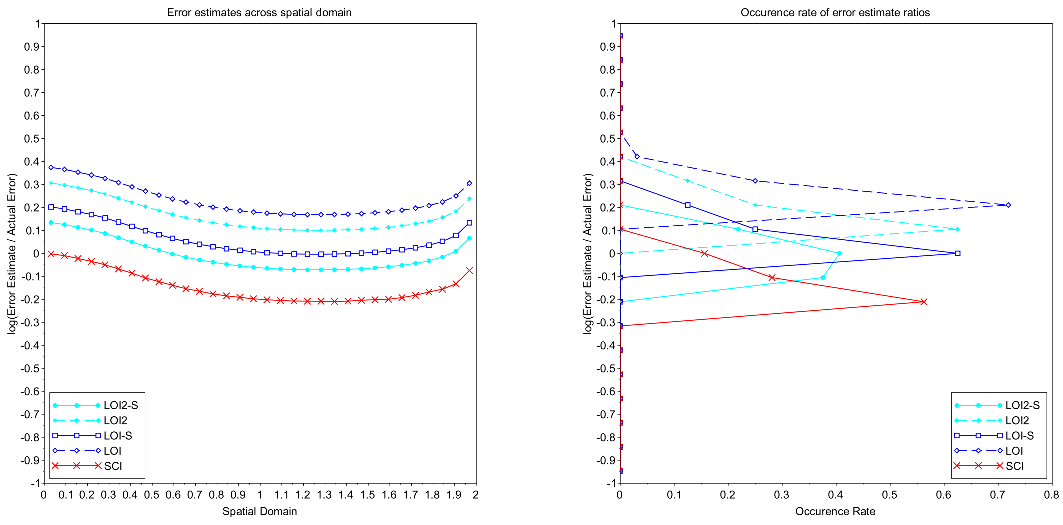


Figure 5.10: The log of the ratio of estimated error to actual error for each subinterval for a collocation solution to BVODE problem 2 with $p = 6$ and $nint = 32$.

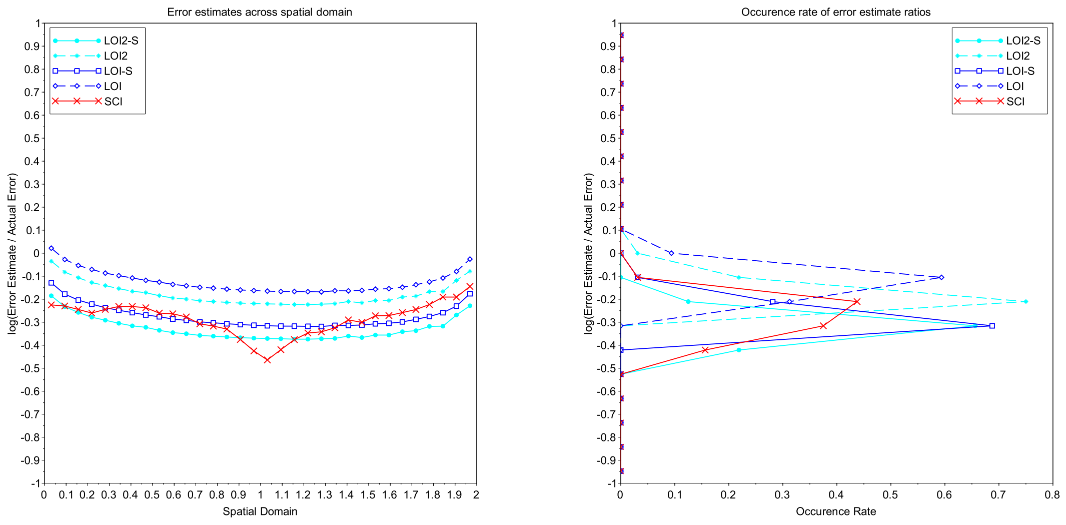


Figure 5.11: The log of the ratio of estimated error to actual error for each subinterval for a collocation solution to BVODE problem 2 with $p = 7$ and $nint = 32$.

5.2 PPDEs

5.2.1 Convergence Results

Within this section the expected convergence rates are $p+1$ for GE, $p+2$ for $SCI-E$, and p for LOI and $LOI2$. Within Table 5.5 we see that the collocation solution does have the expected convergence rate of $p+1$ for most of the entries, although we see more entries outside of the asymptotic region when compared to the BVODE results, including no results within the asymptotic region for $p = 7$. We note that the minimum error achievable is about 10^{-10}, which corresponds to the accuracy delivered by the *daskr* solver. The SCI, LOI, and LOI2 results are very similar, as can be seen in Tables 5.6, 5.7, 5.8, respectively, where most of the entries agree with their expected convergence rates, apart from some of the entries for higher degrees and $nint$ values. For $p = 7$, we do see a few entries within the asymptotic region from the LOI and $LOI2$, for problem 1.

nint	Degree (p)							
	4		5		6		7	
	GE	$rate$	GE	$rate$	GE	$rate$	GE	$rate$
Problem 1								
2	9.05×10^{-3}		8.54×10^{-4}		7.62×10^{-4}		6.90×10^{-5}	
4	1.47×10^{-4}	5.94	5.00×10^{-5}	4.09	5.72×10^{-6}	7.06	5.98×10^{-7}	6.85
8	1.22×10^{-5}	3.59	1.13×10^{-6}	5.47	4.17×10^{-8}	7.1	7.18×10^{-9}	6.38
16	5.47×10^{-7}	4.48	1.93×10^{-8}	5.87	1.38×10^{-9}	4.92	3.32×10^{-10}	4.44
32	1.81×10^{-8}	4.91	1.29×10^{-9}	3.9	4.13×10^{-10}	1.74	1.02×10^{-9}	-1.62
Problem 2								
2	4.13×10^{-3}		6.46×10^{-4}		9.39×10^{-6}		4.64×10^{-6}	
4	1.00×10^{-4}	5.36	5.74×10^{-6}	6.81	2.82×10^{-7}	5.06	1.25×10^{-8}	8.54
8	3.59×10^{-6}	4.8	1.13×10^{-7}	5.66	2.87×10^{-9}	6.62	2.01×10^{-9}	2.64
16	1.17×10^{-7}	4.94	3.98×10^{-9}	4.83	2.40×10^{-9}	0.26	1.89×10^{-10}	3.41
32	4.04×10^{-9}	4.86	2.18×10^{-9}	0.87	6.48×10^{-10}	1.89	4.29×10^{-10}	-1.18

Table 5.5: Global errors (GE) and convergence rates of collocation solutions for varying PPDE problem and degree.

nint	Degree (p)							
	4		5		6		7	
	$SCI-E$	$rate$	$SCI-E$	$rate$	$SCI-E$	$rate$	$SCI-E$	$rate$
Problem 1								
2	1.04×10^{-2}		3.12×10^{-3}		7.49×10^{-4}		2.12×10^{-4}	
4	2.26×10^{-4}	5.53	4.37×10^{-5}	6.16	2.43×10^{-6}	8.27	5.30×10^{-7}	8.64
8	5.94×10^{-6}	5.25	3.26×10^{-7}	7.06	1.01×10^{-8}	7.91	1.16×10^{-9}	8.83
16	9.84×10^{-8}	5.92	2.34×10^{-9}	7.12	9.24×10^{-10}	3.45	3.24×10^{-10}	1.84
32	1.72×10^{-9}	5.84	1.01×10^{-9}	1.22	4.13×10^{-10}	1.16	1.02×10^{-9}	-1.65
Problem 2								
2	4.93×10^{-3}		1.57×10^{-4}		1.28×10^{-5}		2.69×10^{-7}	
4	2.02×10^{-5}	7.93	9.57×10^{-7}	7.35	2.37×10^{-8}	9.08	1.61×10^{-9}	7.39
8	2.53×10^{-7}	6.32	6.47×10^{-9}	7.21	5.26×10^{-10}	5.49	1.99×10^{-9}	-0.31
16	8.65×10^{-9}	4.87	2.75×10^{-9}	1.24	2.39×10^{-9}	-2.19	1.89×10^{-10}	3.4
32	1.81×10^{-9}	2.25	2.16×10^{-9}	0.35	6.48×10^{-10}	1.89	4.29×10^{-10}	-1.18

Table 5.6: SCI errors ($SCI-E$) and convergence rates for varying PPDE problem and degree.

nint	Degree (p)							
	4		5		6		7	
	$LOI - E$	$rate$	$LOI - E$	$rate$	$LOI - E$	$rate$	$LOI - E$	$rate$
Problem 1								
2	1.32×10^{-2}		9.73×10^{-3}		5.74×10^{-4}		6.93×10^{-4}	
4	4.52×10^{-3}	1.55	9.10×10^{-5}	6.74	5.14×10^{-5}	3.48	5.15×10^{-6}	7.07
8	2.59×10^{-4}	4.12	1.02×10^{-5}	3.15	1.14×10^{-6}	5.5	4.04×10^{-8}	6.99
16	2.03×10^{-5}	3.68	5.05×10^{-7}	4.34	1.96×10^{-8}	5.86	7.74×10^{-10}	5.71
32	1.29×10^{-6}	3.97	1.82×10^{-8}	4.79	7.20×10^{-10}	4.77	1.02×10^{-9}	-0.4
Problem 2								
2	3.44×10^{-2}		6.46×10^{-4}		4.52×10^{-4}		4.64×10^{-6}	
4	1.84×10^{-3}	4.23	8.41×10^{-5}	2.94	5.45×10^{-6}	6.37	2.64×10^{-7}	4.14
8	1.54×10^{-4}	3.58	3.47×10^{-6}	4.6	1.13×10^{-7}	5.59	3.26×10^{-9}	6.34
16	1.03×10^{-5}	3.91	1.16×10^{-7}	4.91	3.62×10^{-9}	4.96	1.91×10^{-10}	4.09
32	6.51×10^{-7}	3.98	4.19×10^{-9}	4.79	6.75×10^{-10}	2.42	4.29×10^{-10}	-1.16

Table 5.7: LOI errors ($LOI - E$) and convergence rates for varying PPDE problem and degree.

nint	Degree (p)							
	4		5		6		7	
	$LOI2 - E$	$rate$	$LOI2 - E$	$rate$	$LOI2 - E$	$rate$	$LOI2 - E$	$rate$
Problem 1								
2	8.42×10^{-3}		3.53×10^{-3}		7.32×10^{-4}		2.74×10^{-4}	
4	9.28×10^{-4}	3.18	5.46×10^{-5}	6.02	1.80×10^{-5}	5.34	2.06×10^{-6}	7.05
8	5.39×10^{-5}	4.11	3.44×10^{-6}	3.99	3.56×10^{-7}	5.66	1.46×10^{-8}	7.15
16	3.71×10^{-6}	3.86	1.59×10^{-7}	4.44	6.57×10^{-9}	5.76	4.70×10^{-10}	4.96
32	2.30×10^{-7}	4.01	6.17×10^{-9}	4.68	4.98×10^{-10}	3.72	1.02×10^{-9}	-1.12
Problem 2								
2	8.56×10^{-3}		6.46×10^{-4}		1.41×10^{-4}		4.64×10^{-6}	
4	3.81×10^{-4}	4.49	2.70×10^{-5}	4.58	1.72×10^{-6}	6.36	9.38×10^{-8}	5.63
8	2.81×10^{-5}	3.76	1.08×10^{-6}	4.64	3.39×10^{-8}	5.66	2.32×10^{-9}	5.34
16	1.84×10^{-6}	3.93	3.61×10^{-8}	4.9	2.85×10^{-9}	3.57	1.90×10^{-10}	3.61
32	1.18×10^{-7}	3.97	2.44×10^{-9}	3.89	6.57×10^{-10}	2.12	4.29×10^{-10}	-1.17

Table 5.8: LOI2 errors ($LOI2 - E$) and convergence rates for varying PPDE problem and degree.

5.2.2 Error Estimation

The SCI for the PPDE class is not as consistent as for the BVODE class, although it still provides an accurate error estimate in most cases. In Figure 5.19 we see the greatest underestimation from the SCI in the PPDE class with an error log ratio of about -0.75 for all of the subintervals. We also see a similar underestimate in Figures 5.18 and 5.15 although these have occurrence rates of only about 0.1. The SCI does not overestimate the error in any of the tested situations, and tends to provide an error log ratio between -0.5 and -0.1 while also following a slight downward trend as p increases.

The LOI and LOI2 perform moderately well for this class. We once again see a downward trend in the error log ratio as p increases, although it is more pronounced in this case. With $p = 4$, in Figures 5.12 and 5.16, we see the LOI and LOI2 producing error log ratios of about 0.5 and 0.3 respectively, while their scaled versions produce error log ratios of around 0.1 and 0 respectively. The LOI tends to produce error log ratios between -0.5 and 0.5, with its scaled version producing error log ratios between -0.7 and 0.3. The LOI2 produces error log ratios between -0.5 and 0.3, with the scaled version producing error log ratios between -0.8 and 0. The scaling in this class tends to cause the LOI and LOI2 to further underestimate the error.

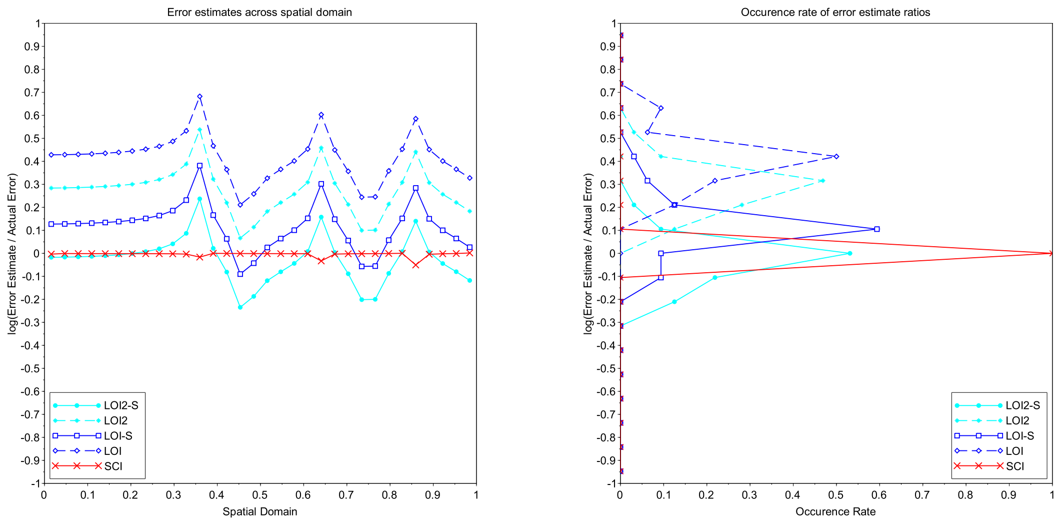


Figure 5.12: The log of the ratio of estimated error to actual error of each subinterval for a collocation solution to PPDE problem 1 with $p = 4$ and $nint = 32$.

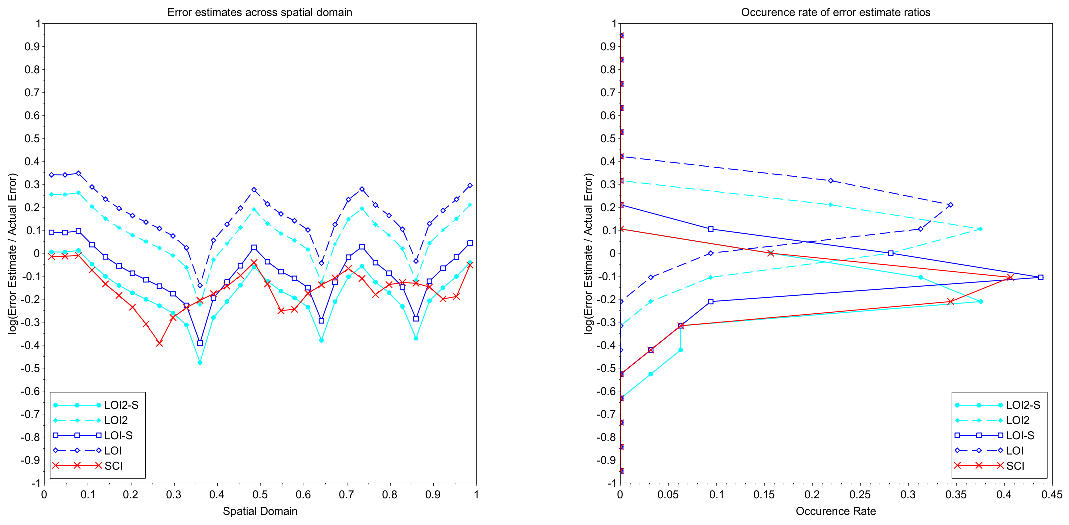


Figure 5.13: The log of the ratio of estimated error to actual error of each subinterval for a collocation solution to PPDE problem 1 with $p = 5$ and $nint = 32$.

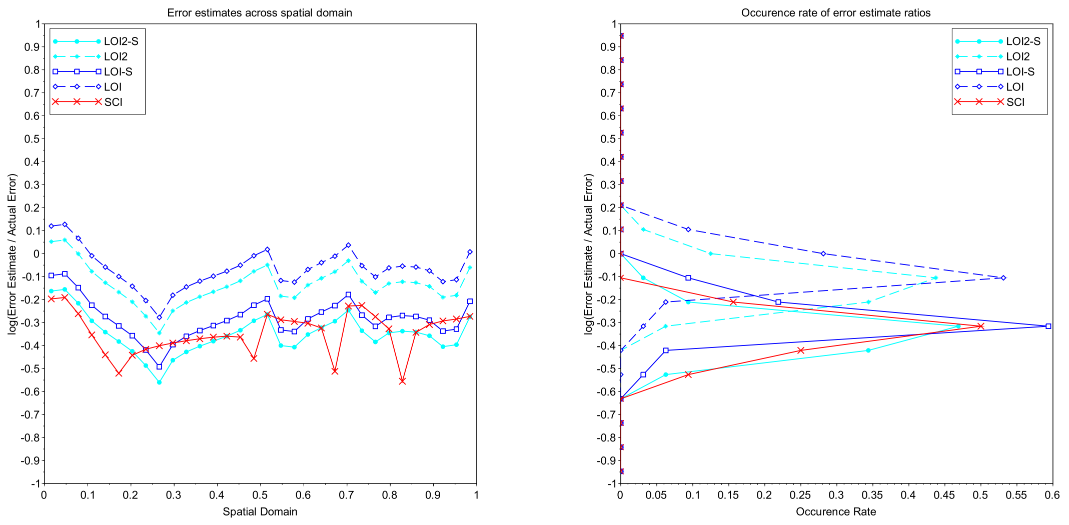


Figure 5.14: The log of the ratio of estimated error to actual error of each subinterval for a collocation solution to PPDE problem 1 with $p = 6$ and $nint = 32$.

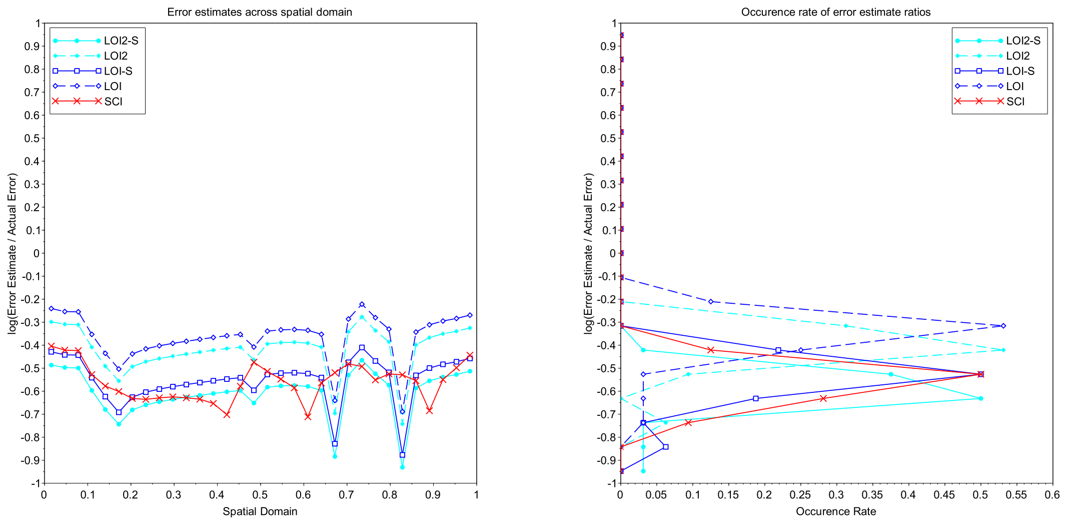


Figure 5.15: The log of the ratio of estimated error to actual error of each subinterval for a collocation solution to PPDE problem 1 with $p = 7$ and $nint = 32$.

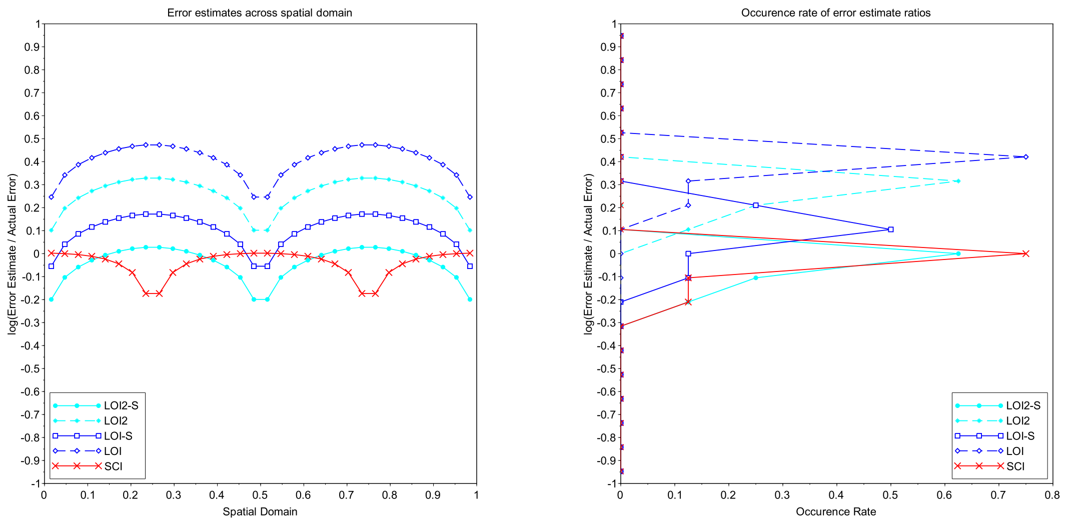


Figure 5.16: The log of the ratio of estimated error to actual error of each subinterval for a collocation solution to PPDE problem 2 with $p = 4$ and $nint = 32$.

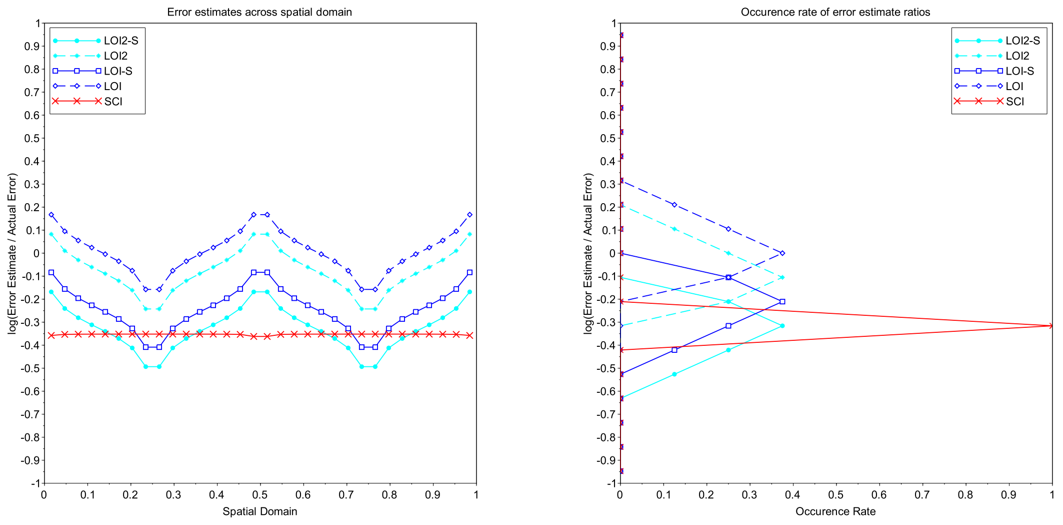


Figure 5.17: The log of the ratio of estimated error to actual error of each subinterval for a collocation solution to PPDE problem 2 with $p = 5$ and $nint = 32$.

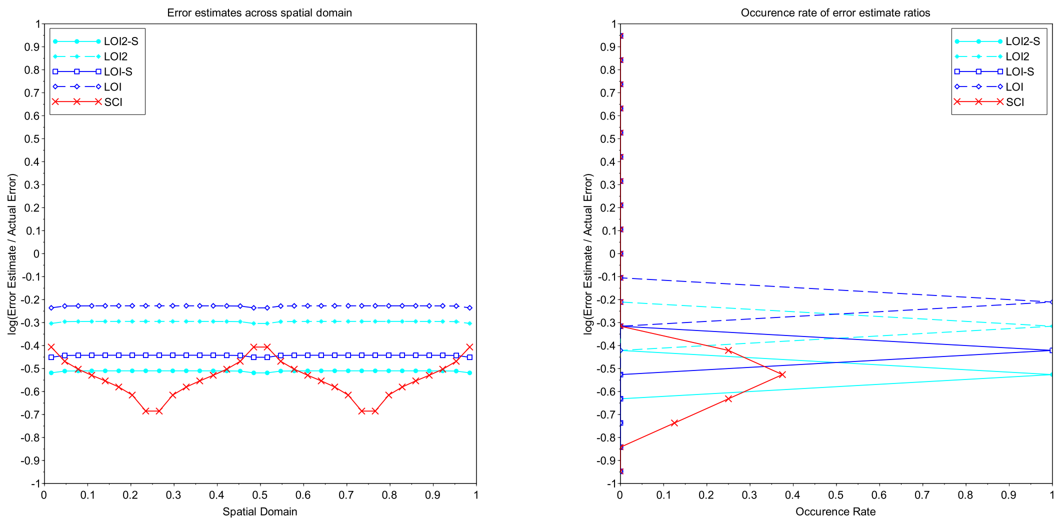


Figure 5.18: The log of the ratio of estimated error to actual error of each subinterval for a collocation solution to PPDE problem 2 with $p = 6$ and $nint = 32$.

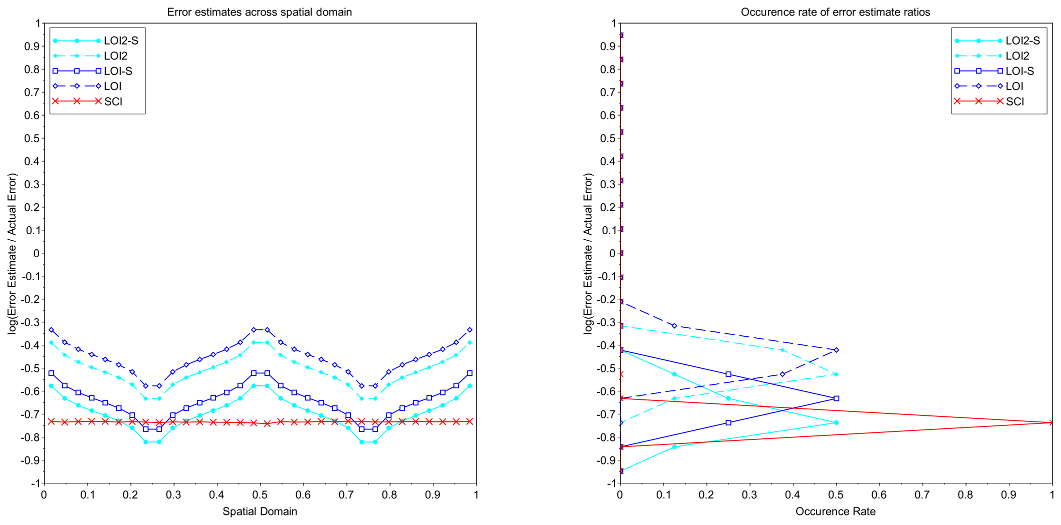


Figure 5.19: The log of the ratio of estimated error to actual error of each subinterval for a collocation solution to PPDE problem 2 with $p = 7$ and $nint = 32$.

5.3 EPDEs

5.3.1 Convergence Results

Within this section the expected convergence rates are $p + 1$ for GE, $p + 2$ for $SCI - E$, and p for LOI and $LOI2$. In Table 5.9 we can see that the collocation solution does not agree exactly with the expected convergence rate of $p + 1$, but the results are still reasonable. For the rates associated with $p = 4$ we see that the asymptotic region is being approached and the expected convergence rate is being approached, but it is not achieved for the values of $nint$ tested. For each higher p value, once again the *rate* begins to be impacted by the lower bound which is again around 10^{-13} due to the accuracy delivered by *fsolve*.

The SCI and LOI do not agree at all with the expected convergence rates as can be seen in Tables 5.10 and 5.11. The expected cause of this is that the SCI and LOI in two dimensions would require sufficient convergence of collocation solution derivative values at points where it is not expected to occur. This is also supported by seeing that the LOI2, which does not rely on solution derivative values, delivers the expected convergence results.

On the other hand, the LOI2 error agrees very closely with the expected convergence rate of p as can be seen in Table 5.12, and contains many entries in or just approaching the asymptotic region. There are a few entries which are impacted by the lower bound on the error, mostly those where $p = 7$ but also sometimes when $p = 6$.

	Degree (p)							
	4		5		6		7	
nint	GE	$rate$	GE	$rate$	GE	$rate$	GE	$rate$
Problem 1								
2	9.94×10^{-6}		1.89×10^{-7}		3.03×10^{-9}		4.61×10^{-11}	
4	3.63×10^{-7}	4.774	3.49×10^{-9}	5.754	2.71×10^{-11}	6.802	2.11×10^{-13}	7.769
8	1.22×10^{-8}	4.892	5.93×10^{-11}	5.88	2.23×10^{-13}	6.927	2.98×10^{-14}	2.829
16	3.67×10^{-10}	5.06	9.65×10^{-13}	5.941	2.04×10^{-13}	0.126	1.02×10^{-13}	-1.773
32	1.17×10^{-11}	4.969	7.77×10^{-14}	3.635	5.99×10^{-13}	-1.552	-	-
Problem 2								
2	2.99×10^{-4}		1.09×10^{-5}		3.27×10^{-7}		1.02×10^{-8}	
4	1.69×10^{-5}	4.149	2.80×10^{-7}	5.279	3.93×10^{-9}	6.379	6.20×10^{-11}	7.361
8	7.21×10^{-7}	4.546	5.69×10^{-9}	5.622	4.67×10^{-11}	6.396	3.32×10^{-13}	7.545
16	2.49×10^{-8}	4.858	1.20×10^{-10}	5.563	4.63×10^{-13}	6.656	9.64×10^{-14}	1.785
32	8.58×10^{-10}	4.859	3.99×10^{-13}	8.236	5.43×10^{-13}	-0.232	3.15×10^{-13}	-1.708

Table 5.9: Global errors (GE) and convergence rates of collocation solutions for varying EPDE problem and degree.

	Degree (p)							
	4		5		6		7	
nint	$SCI-E$	$rate$	$SCI-E$	$rate$	$SCI-E$	$rate$	$SCI-E$	$rate$
Problem 1								
2	2.42×10^{-2}		6.15×10^{-3}		7.19×10^{-3}		4.70×10^{-3}	
4	6.85×10^{-3}	1.817	1.84×10^{-3}	1.741	2.16×10^{-3}	1.733	1.48×10^{-3}	1.668
8	1.85×10^{-3}	1.888	5.30×10^{-4}	1.796	6.03×10^{-4}	1.842	4.21×10^{-4}	1.811
16	3.93×10^{-4}	2.237	1.42×10^{-4}	1.899	1.59×10^{-4}	1.92	1.12×10^{-4}	1.906
32	8.50×10^{-5}	2.208	1.32×10^{-6}	6.746	4.10×10^{-5}	1.96	-	-
Problem 2								
2	7.86×10^{-2}		1.14×10^{-2}		1.74×10^{-2}		7.27×10^{-3}	
4	3.19×10^{-2}	1.302	5.04×10^{-3}	1.175	7.52×10^{-3}	1.211	4.38×10^{-3}	0.731
8	1.09×10^{-2}	1.552	2.43×10^{-3}	1.051	3.12×10^{-3}	1.269	2.01×10^{-3}	1.123
16	2.79×10^{-3}	1.964	8.63×10^{-4}	1.493	1.03×10^{-3}	1.603	6.97×10^{-4}	1.53
32	6.28×10^{-4}	2.15	9.38×10^{-6}	6.525	2.96×10^{-4}	1.793	5.22×10^{-5}	3.738

Table 5.10: SCI errors ($SCI - E$) and convergence rates of collocation solutions for varying EPDE problem and degree. The expected convergence rates are not achieved.

	Degree (p)							
	4		5		6		7	
nint	$LOI-E$	$rate$	$LOI-E$	$rate$	$LOI-E$	$rate$	$LOI-E$	$rate$
Problem 1								
2	1.99×10^{-2}		1.16×10^{-2}		2.49×10^{-3}		1.57×10^{-3}	
4	4.07×10^{-3}	2.289	3.00×10^{-3}	1.946	5.83×10^{-4}	2.094	3.97×10^{-4}	1.982
8	8.84×10^{-4}	2.203	7.25×10^{-4}	2.05	1.44×10^{-4}	2.015	1.00×10^{-4}	1.987
16	1.91×10^{-4}	2.207	1.80×10^{-4}	2.007	2.29×10^{-5}	2.656	1.27×10^{-5}	2.979
32	3.18×10^{-5}	2.59	2.88×10^{-5}	2.644	4.12×10^{-8}	9.118	-	-
Problem 2								
2	1.53×10^{-1}		7.45×10^{-2}		2.33×10^{-2}		1.16×10^{-2}	
4	3.34×10^{-2}	2.194	2.07×10^{-2}	1.847	5.24×10^{-3}	2.15	3.03×10^{-3}	1.941
8	7.35×10^{-3}	2.182	5.39×10^{-3}	1.941	1.23×10^{-3}	2.091	7.85×10^{-4}	1.948
16	1.56×10^{-3}	2.237	1.41×10^{-3}	1.935	1.98×10^{-4}	2.633	9.51×10^{-5}	3.045
32	2.66×10^{-4}	2.552	2.25×10^{-4}	2.647	3.81×10^{-7}	9.025	2.50×10^{-5}	1.927

Table 5.11: LOI errors ($LOI-E$) and convergence rates of collocation solutions for varying EPDE problem and degree. The expected convergence rates are not achieved.

	Degree (p)							
	4		5		6		7	
nint	$LOI2-E$	$rate$	$LOI2-E$	$rate$	$LOI2-E$	$rate$	$LOI2-E$	$rate$
Problem 1								
2	1.23×10^{-4}		3.06×10^{-6}		6.11×10^{-8}		1.10×10^{-9}	
4	8.69×10^{-6}	3.825	1.08×10^{-7}	4.819	1.04×10^{-9}	5.873	9.70×10^{-12}	6.824
8	5.35×10^{-7}	4.022	3.61×10^{-9}	4.91	1.51×10^{-11}	6.107	6.44×10^{-14}	7.235
16	3.36×10^{-8}	3.993	1.16×10^{-10}	4.955	2.57×10^{-13}	5.881	1.02×10^{-13}	-0.659
32	5.40×10^{-10}	5.959	3.69×10^{-12}	4.977	6.00×10^{-13}	-1.225	-	-
Problem 2								
2	3.07×10^{-3}		1.37×10^{-4}		6.79×10^{-6}		2.47×10^{-7}	
4	2.73×10^{-4}	3.487	5.76×10^{-6}	4.578	1.47×10^{-7}	5.529	2.70×10^{-9}	6.518
8	2.05×10^{-5}	3.738	2.20×10^{-7}	4.707	2.24×10^{-9}	6.038	1.68×10^{-11}	7.327
16	1.31×10^{-6}	3.965	8.46×10^{-9}	4.703	3.05×10^{-11}	6.199	1.49×10^{-13}	6.815
32	2.23×10^{-8}	5.876	2.94×10^{-10}	4.849	5.47×10^{-13}	5.799	3.15×10^{-13}	-1.077

Table 5.12: LOI2 errors ($LOI2-E$) and convergence rates of collocation solutions for varying EPDE problem and degree.

5.3.2 Error Estimation

As can be seen in Tables 5.10 and 5.11 the convergence of the SCI and LOI are not as expected, and this can be seen in the Figures within this section where the SCI and LOI consistently produce an error estimate that is an order of magnitude greater than the true error.

The LOI2 performs well in this problem class, producing very consistent error log ratios for a given degree. The decreasing trend in error log ratio as p increases is once again seen here and will decrease the error log ratio by about 0.5 when comparing $p = 4$ to $p = 7$. For degrees 4 and 5, the scaled LOI2 provides an error log ratio of about 0, while for degrees 6 and 7 the non-scaled LOI2 provides an error log ratio of about 0.

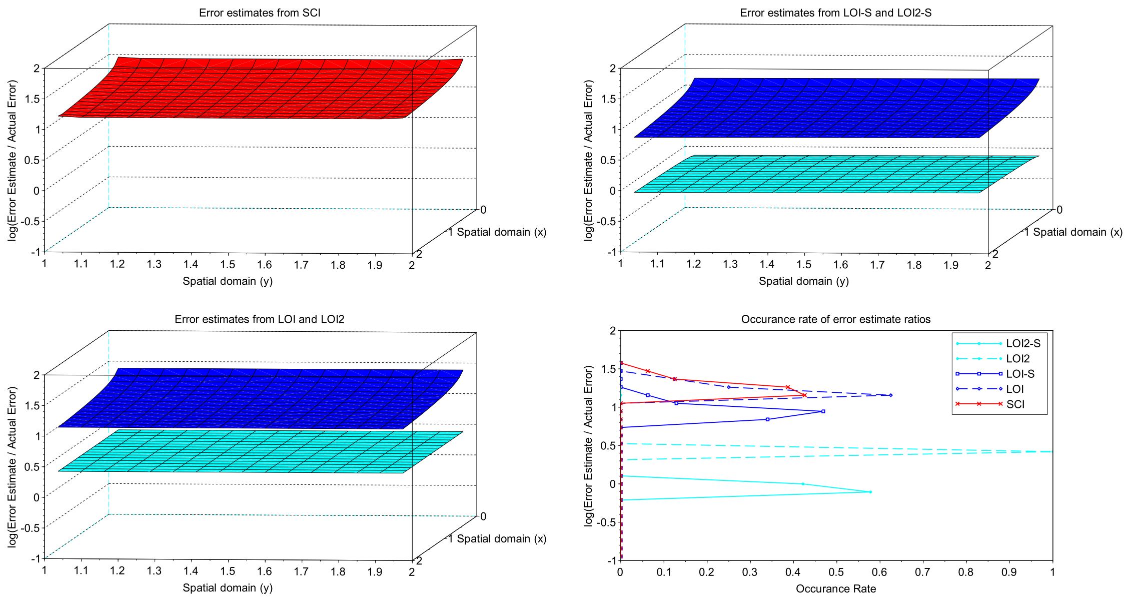


Figure 5.20: Error estimate results for a collocation solution to EPDE problem 1 with $p = q = 4$ and $nint = 16$.

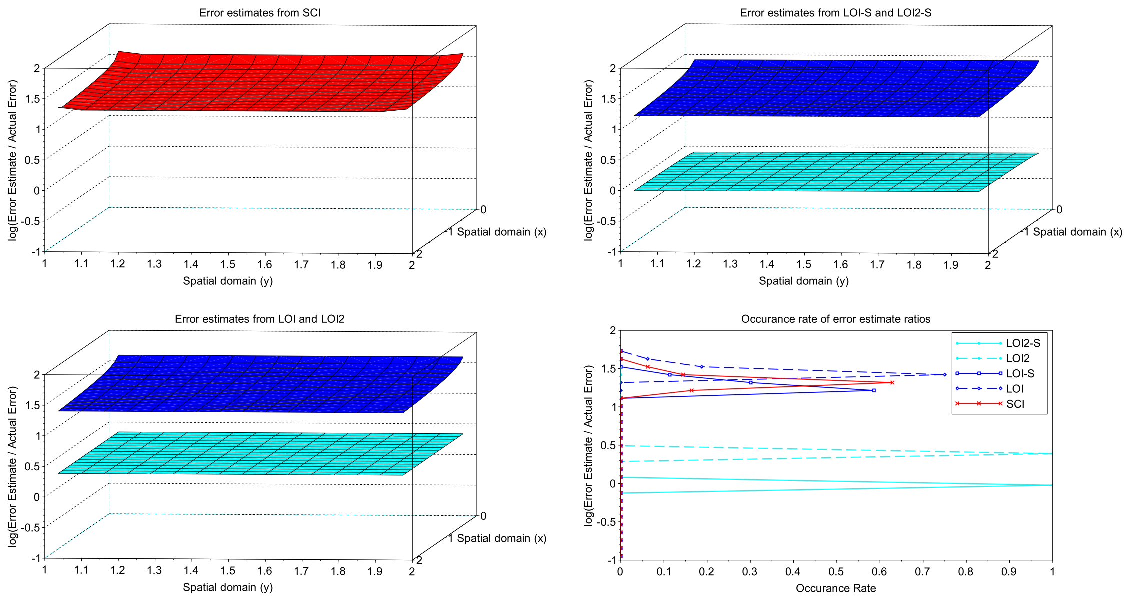


Figure 5.21: Error estimate results for a collocation solution to EPDE problem 1 with $p = q = 5$ and $nint = 16$.

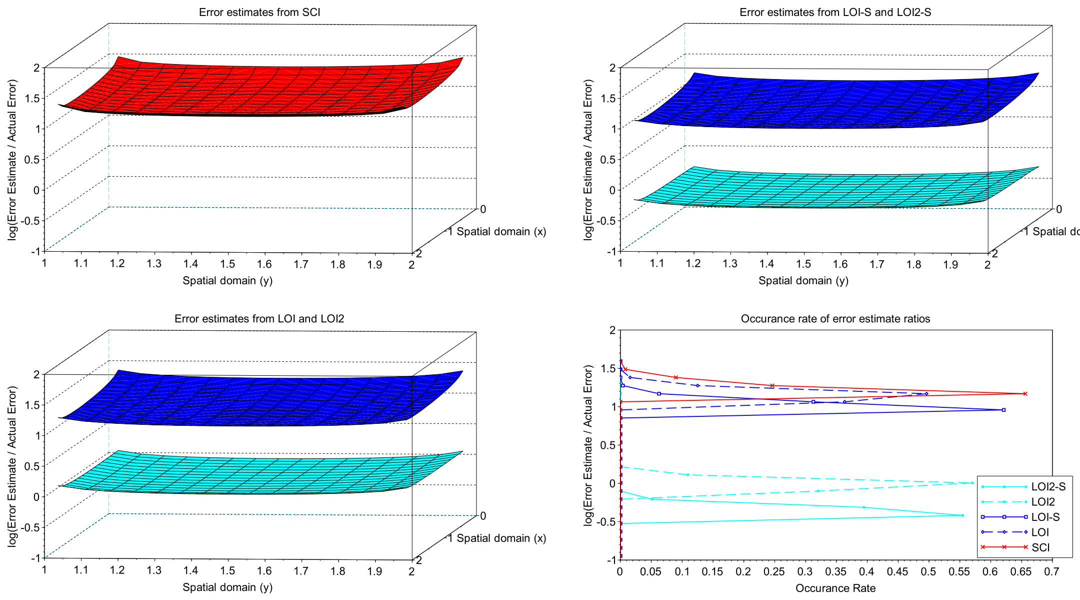


Figure 5.22: Error estimate results for a collocation solution to EPDE problem 1 with $p = q = 6$ and $nint = 16$.

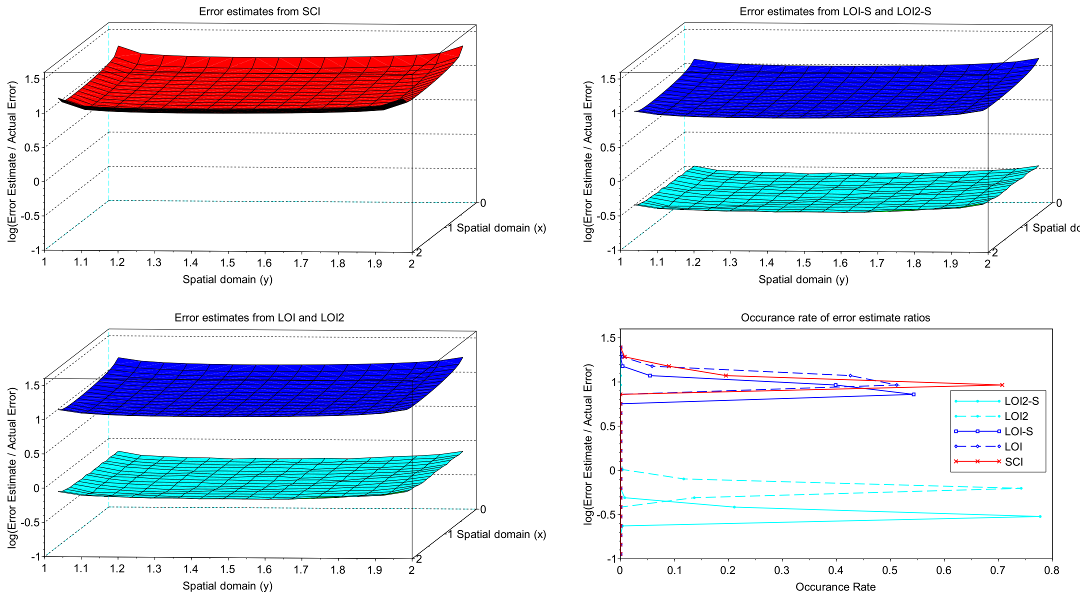


Figure 5.23: Error estimate results for a collocation solution to EPDE problem 1 with $p = q = 7$ and $nint = 16$.

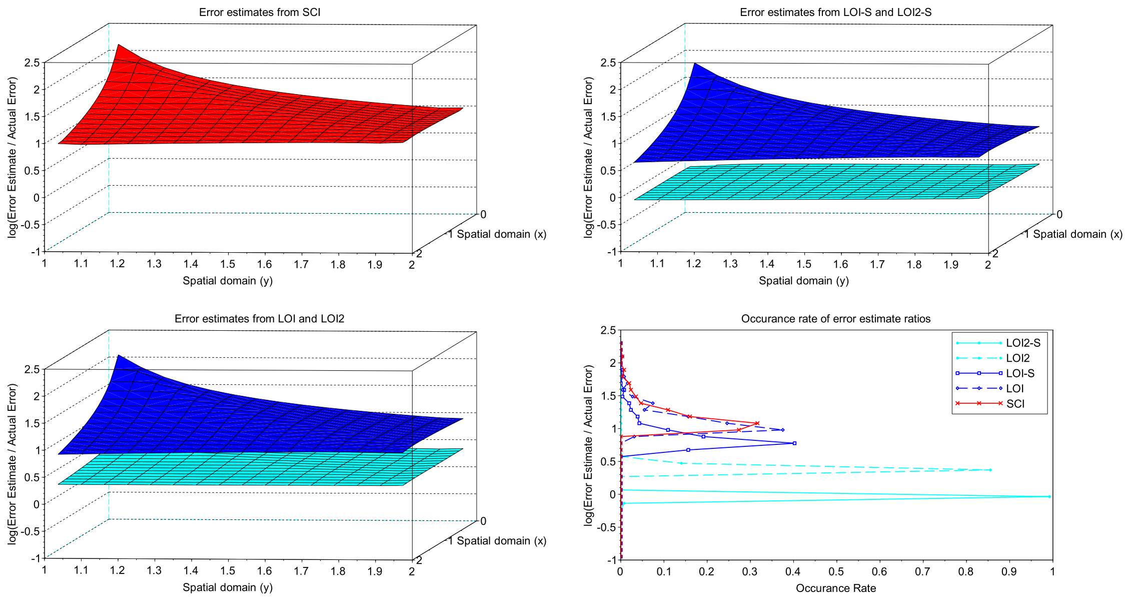


Figure 5.24: Error estimate results for a collocation solution to EPDE problem 2 with $p = q = 4$ and $nint = 16$.

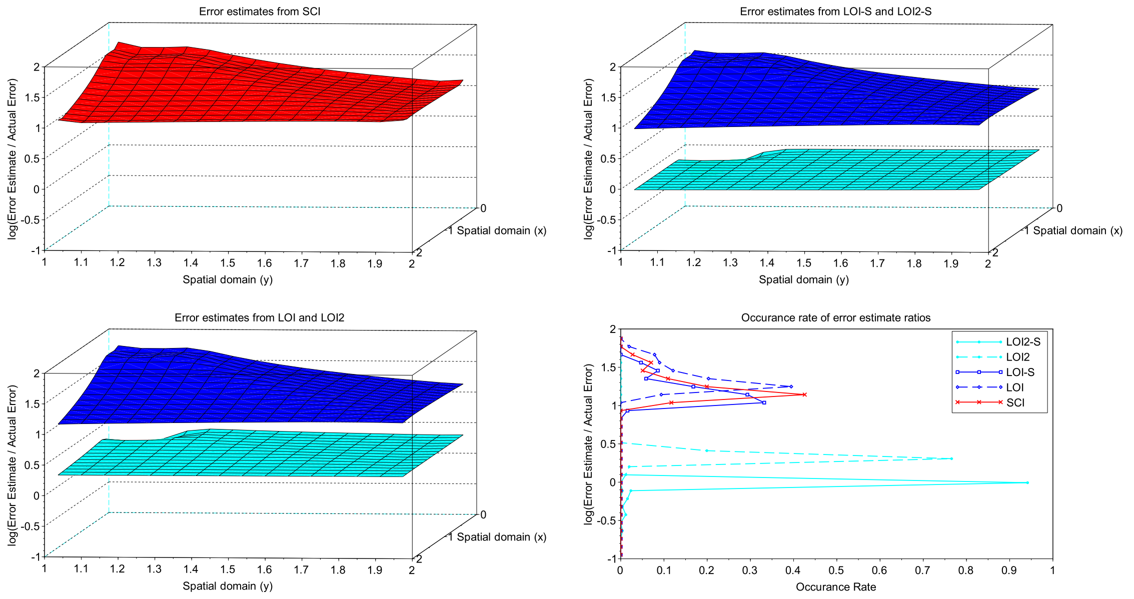


Figure 5.25: Error estimate results for a collocation solution to EPDE problem 2 with $p = q = 5$ and $nint = 16$.

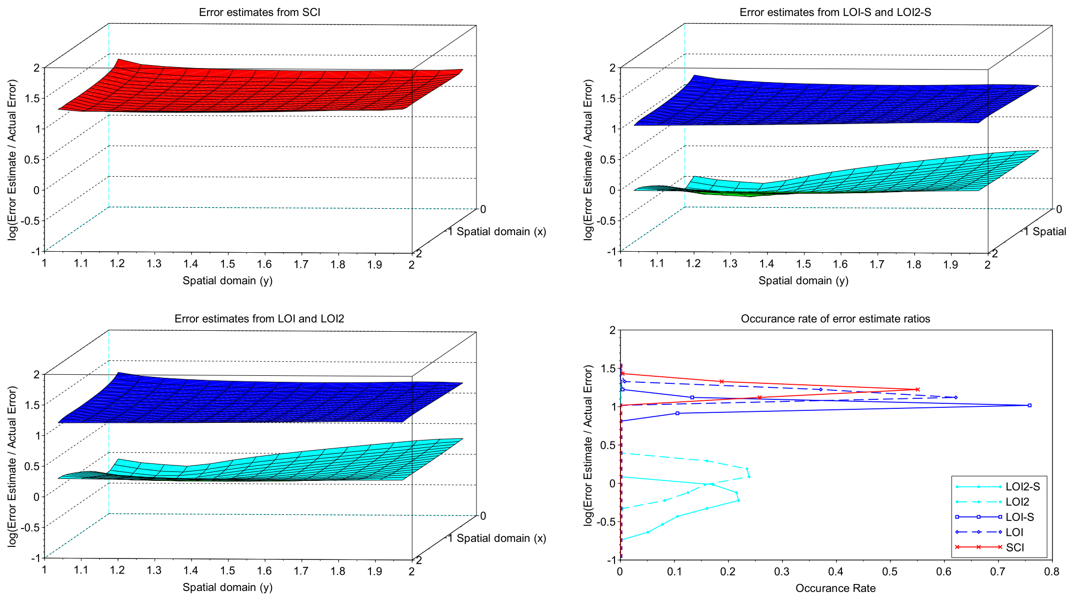


Figure 5.26: Error estimate results for a collocation solution to EPDE problem 2 with $p = q = 6$ and $nint = 16$.

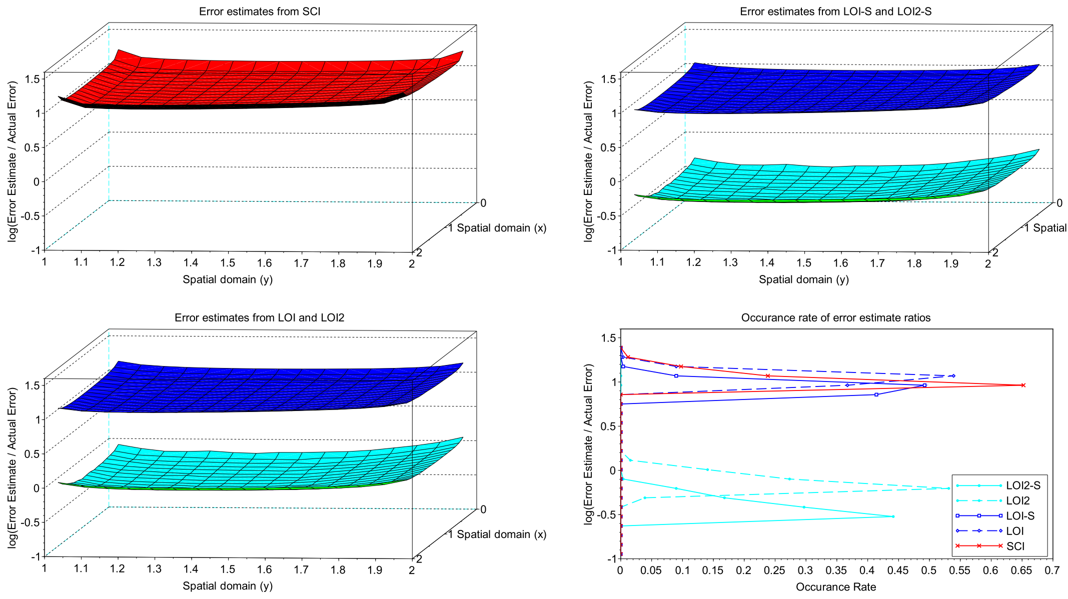


Figure 5.27: Error estimate results for a collocation solution to EPDE problem 2 with $p = q = 7$ and $nint = 16$.

5.4 2DPPDEs

5.4.1 Convergence Results

As is shown by Table 5.13, the expected convergence rates for the collocation solution are not achieved. It appears that there is an issue with the implementation for this case. As such we can not evaluate the performance of the various error estimation methods for this problem class.

	Degree (p)							
	4		5		6		7	
nint	GE	$rate$	GE	$rate$	GE	$rate$	GE	$rate$
Problem 1								
2	3.95×10^{-4}		6.03×10^{-5}		2.42×10^{-5}		2.81×10^{-4}	
4	1.61×10^{-5}	4.612	9.35×10^{-6}	2.689	1.99×10^{-6}	3.606	8.97×10^{-6}	4.972
8	9.97×10^{-7}	4.018	7.82×10^{-7}	3.58	4.29×10^{-6}	-1.11	2.19×10^{-6}	2.032
16	8.56×10^{-7}	0.219	2.01×10^{-5}	-4.681	1.57×10^{-5}	-1.871	6.65×10^{-5}	-4.923
32	1.68×10^{-5}	-4.299	8.41×10^{-6}	1.254	-	-	-	-
Problem 2								
2	2.29×10^{-3}		3.73×10^{-4}		1.17×10^{-4}		2.32×10^{-5}	
4	9.83×10^{-5}	4.541	2.20×10^{-5}	4.082	5.30×10^{-5}	1.147	9.67×10^{-6}	1.261
8	3.67×10^{-6}	4.745	1.14×10^{-4}	-2.373	3.15×10^{-7}	7.393	4.04×10^{-6}	1.26
16	9.90×10^{-6}	-1.433	2.67×10^{-5}	2.093	1.08×10^{-4}	-8.425	1.18×10^{-4}	-4.864
32	1.20×10^{-4}	-3.6	-	-	-	-	-	-

Table 5.13: Global errors (GE) and convergence rates of collocation solutions for varying 2DPPDE problem and degree.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Within this thesis we have discussed B-spline Gaussian Collocation for the problem classes, BVODE, PPDE, EPDE, and 2DPPDE. We have also presented numerical convergence results from two test problems from each of the BVODE, PPDE, and EPDE classes, which verify their expected convergence rates for the collocation solutions using the software introduced within this thesis. An overview of the source code and software itself are also included.

We have also introduced various interpolants for calculating an error estimate for a collocation solution. These included the SCI and LOI, which have been also generalized to two spatial dimensions, as well as a modified LOI, named LOI2, which can be used in one or two spatial dimensions. We examined the performance of these interpolants for the BVODE, PPDE, and EPDE problem classes. In the one spatial dimension classes, BVODE and PPDE, all of the interpolants were able to provide an error estimate of the right order, while being more accurate for collocation solutions of lower degree. For the EPDE case the SCI and LOI were unable to provide an error estimate of the right order, while the LOI2 was. We also attempted to investigate the use of collocation and interpolation based error estimates for the 2DPPDE problem class but ran into issues with the implementation of these methods for this problem class.

6.2 Future Work

There are two major next steps which present themselves from this thesis. The first is a further investigation into the 2DPPDE problem class, including further investigation of the software so that the expected convergence rates of the collocation solutions are seen. This would then be followed by an assessment of the performance of the various error estimation interpolants in the 2DPPDE problem class.

Another area for future work would be the development of more formal software in a more efficient and compiled language that performs the same functions as the software presented in this thesis. This is desirable as the calculation of a higher order collocation solution for a EPDE problem using the existing software can take multiple hours.

Bibliography

[1] T. Arsenault, T. Smith, and P. Muir. Superconvergent interpolants for efficient spatial error estimation in 1D PDE collocation solvers. *Can. Appl. Math. Q.*, 17(3):409–431, 2009.

[2] T. Arsenault, T. Smith, P. Muir, and P. Keast. Efficient interpolation-based error estimation for 1D time-dependent PDE collocation codes. *Saint Mary's University, Dept. of Mathematics and Computing Science Technical Report Series, Technical Report 2011_001*, 2011.

[3] T. Arsenault, T. Smith, P. Muir, and J. Pew. Asymptotically correct interpolation-based spatial error estimation for 1D PDE solvers. *Can. Appl. Math. Q.*, 20(3):307–328, 2012.

[4] U. Ascher, J. Christiansen, and R. D. Russell. Algorithm 569: COLSYS: Collocation software for boundary-value ODEs. *ACM Trans. Math. Softw.*, 7(2):223–229, 1981.

[5] U.M. Ascher, R.M.M. Mattheij, and R.D. Russell. *Numerical solution of boundary value problems for ordinary differential equations.* Classics in applied mathematics. Society for Industrial and Applied Mathematics (SIAM), United States, corr. republication. edition, 1995.

[6] J.P. Berrut and L.N. Trefethen. Barycentric Lagrange interpolation. *SIAM Rev.*, 46(3):501–517, 2004.

[7] P.N. Brown, A.C. Hindmarsh, and L.R. Petzold. Using Krylov methods in the solution of large-scale differential-algebraic systems. *SIAM J. Sci. Comput.*, 15(6):1467–1488, 1994.

[8] P.N. Brown, A.C. Hindmarsh, and L.R. Petzold. Consistent initial condition calculation for differential-algebraic systems. *SIAM J. Sci. Comput.*, 19(5):1495–1512, 1998.

[9] J.H. Cerutti and S.V. Parter. Collocation methods for parabolic partial differential equations in one space dimension. *Numer. Math.*, 26(3):227–254, 1976.

[10] C. de Boor. Collocation at Gaussian points. *SIAM J. Numer. Anal.*, 10(4):582–606, 1973.

[11] C. de Boor. Package for calculating with B-splines. *SIAM J. Numer. Anal.*, 14(3):441–472, 1973.

[12] J. Diaz, G. Fairweather, and P. Keast. COLROW and ARCECO: FORTRAN packages for solving certain almost block diagonal linear systems by modified alternate row and column elimination. *ACM Trans. Math. Softw.*, 9(3):376–380, 1983.

[13] J. Douglas and T. Dupoont. A finite element collocation method for quasilinear parabolic equations. *Math. Comp.*, 27(121):17–28, 1973.

[14] W.F. Finden. An error term and uniqueness for Hermite–Birkhoff interpolation involving only function values and/or first derivative values. *J. Comput. Appl. Math.*, 212(1):1 – 15, 2008.

[15] P. Keast and P. Muir. Algorithm 688: EPDCOL: A more efficient PDECOL code. *ACM Trans. Math. Softw.*, 17(2):153–166, 1991.

[16] Z. Li and P. Muir. B-spline Gaussian collocation software for two-dimensional parabolic PDEs. *Adv. Appl. Math. Mech.*, 5(4):528–547, 2013.

[17] N. K. Madsen and R. F. Sincovec. Algorithm 540: PDECOL, general collocation software for partial differential equations. *ACM Trans. Math. Softw.*, 5(3):326–351, 1979.

[18] L. R. Petzold. Description of DASSL: a differential/algebraic system solver. *Technical Report, Sandia Labs, Livermore, CA*, 1982.

[19] J. Pew, Z. Li, and P. Muir. Algorithm 962: BACOLI: B-spline adaptive collocation software for pdes with interpolation-based spatial error control. *ACM Trans. Math. Softw.*, 42(3):25:1–25:17, 2016.

[20] J. Rice and R. Boisvert. *Solving elliptec problems using ELLPACK*. Springer Series in Computational Mathematics. Springer-Verlag, New York, 1985.

[21] C. Tannahill, P. Muir, and A. Fraser. Unpublished research. 2019.

[22] R. Wang, P. Keast, and P. Muir. BACOL: B-spline adaptive collocation software for 1-D parabolic PDEs. *ACM Trans. Math. Softw.*, 30(4):454–470, 2004.

Appendix

README.txt

```
Before these scripts can be run, the following setup must be done:
In each of the main scripts, BVODE.sce, PPDE.sce, EPDE.sce, 2DPPDE.sce, the variable
    codeDir
must be set to the directory where these files are. (example. codeDir = "C:\Users\
    name\Documents\SciCol")
There are also the verbose and jacMode control parameters which are described in each
     of the main scripts.

These scripts require the Scilab MinGw toolbox (to compile included Fortran code),
which requires the Equation solution Compiler gcc-6.2.0 package
the 32 bit version - http://atoms.scilab.org/toolboxes/mingw/0.10.0/files/gcc
    -6.2.0-32.exe
the 64 bit version - http://atoms.scilab.org/toolboxes/mingw/0.10.0/files/gcc
    -6.2.0-64.exe

After this has been downloaded and installed, in the Scilab console you can now
    install the MinGw toolbox with the following command
        atomsinstall("mingw")

Once this has completed you must log out of your Windows account, and back in to
    enable it.
Now when Scilab is launched, there should be a message stating that MinGw has loaded,
     this may take a bit.

Finally, the main scripts can be executed with the following command
        exec("C:\Users\name\Documents\SciCol\BVODE.sce",-1)
Where within the quotes should be the complete location to the desired main script.
This will compile the included Fortran and run the required provided Scilab code, and
     now the collocate function can be called to calculate a collocation solution.
```

BVODE.sce

```
global codeDir verbose jacMode
codeDir = "" // This must be set to the directory where the Scilab scripts are
verbose = 0 // Determines the verbosity of the collocation procedure, 0 is no output,
     1 outputs information about the progress, while 2 also includes timing of
    various stages.
jacMode = 0 // Determines if fsolve/daskr will calculate the Jacobian (0), or if it
    will be approximated using knowledge of the structure and finite difference
    methods (1). Setting to 1 may speed up computation, or may also result in fsolve/
    daskr failing to converge.

chdir(codeDir);


exec (codeDir+"\core.sci");
exec (codeDir+"\err.sci");

exec (codeDir+"\core1D.sci");
exec (codeDir+"\err1D.sci");

exec (codeDir+"\coreBVODE.sci");
exec (codeDir+"\probsBVODE.sci");
```

PPDE.sce

```
global codeDir verbose jacMode
codeDir = "" // This must be set to the directory where the Scilab scripts are
verbose = 0 // Determines the verbosity of the collocation procedure, 0 is no output,
     1 outputs information about the progress, while 2 also includes timing of
    various stages.
jacMode = 0 // Determines if fsolve/daskr will calculate the Jacobian (0), or if it
    will be approximated using knowledge of the structure and finite difference
    methods (1). Setting to 1 may speed up computation, or may also result in fsolve/
    daskr failing to converge.

chdir(codeDir);

exec (codeDir+"\core.sci");
exec (codeDir+"\err.sci");

exec (codeDir+"\core1D.sci");
exec (codeDir+"\err1D.sci");

exec (codeDir+"\corePPDE.sci");
exec (codeDir+"\probsPPDE.sci");
```

EPDE.sce

```
global codeDir verbose jacMode
codeDir = "" // This must be set to the directory where the Scilab scripts are
verbose = 0 // Determines the verbosity of the collocation procedure, 0 is no output,
     1 outputs information about the progress, while 2 also includes timing of
    various stages.
jacMode = 0 // Determines if fsolve/daskr will calculate the Jacobian (0), or if it
    will be approximated using knowledge of the structure and finite difference
    methods (1). Setting to 1 may speed up computation, or may also result in fsolve/
    daskr failing to converge.

chdir(codeDir)

exec (codeDir+"\core.sci");
exec (codeDir+"\err.sci");

exec (codeDir+"\core2D.sci");
exec (codeDir+"\err2D.sci");

exec (codeDir+"\coreEPDE.sci");
exec (codeDir+"\probsEPDE.sci");
```

2DPPDE.sce

```
global codeDir verbose jacMode
codeDir = "" // This must be set to the directory where the Scilab scripts are
verbose = 0 // Determines the verbosity of the collocation procedure, 0 is no output,
     1 outputs information about the progress, while 2 also includes timing of
    various stages.
jacMode = 0 // Determines if fsolve/daskr will calculate the Jacobian (0), or if it
    will be approximated using knowledge of the structure and finite difference
    methods (1). Setting to 1 may speed up computation, or may also result in fsolve/
    daskr failing to converge.

chdir(codeDir);

exec (codeDir+"\core.sci");
exec (codeDir+"\err.sci");

exec (codeDir+"\core2D.sci");
exec (codeDir+"\err2D.sci");

exec (codeDir+"\core2DPPDE.sci");
exec (codeDir+"\probs2DPPDE.sci");
```

core.sci

```
// Global variables which are required by all PDE types.
global A // Lower x bound of the domain
global B // Upper x bound of the domain
global probNum // Specifies which problem is 'active'
probNum = 1;
global ncpts // The number of equations total in the system
global p // Degree of the solution in x
global N // Number of intervals in x
global nconti // Number of continuity conditions imposed on the b-splines at mesh
    points
nconti = 2;
global coeffs // Stores the coefficients of b-splines representing the solution
global atol rtol // Absolute and relative tolerance used for error estimation
atol = 1.d-4;
rtol = 0;
global meshX // The mesh points in x
global knotsX // The knots or breakpoint sequence in x used for the creation of b-
    splines
global colX // The collocation points in x where the PDE must be satisfied
global dy // difference used for finite difference approximations
dy = 2 * sqrt(%eps);

// Builds a knot sequence from meshP to allow for a b-spline basis
// of degree deg over nint intervals with global nconti internal
// continuity conditions.
function knots = buildKnots(meshP, deg, nint)

    // Check if calling from 1D case
    if argn(2) == 1 then
        deg = p
        nint = N
    end

    // Find ncpt for this dimension
    ncpt = (deg-1) * nint + nconti

    // Set the end knots
    for i = 1:(deg-1+nconti)
        knots(i) = meshP(1)
        knots(i + ncpt) = meshP(nint+1)
    end

    // Set the internal knots
    for i = 2:nint
        ii = (i-2) * (deg-1) + (deg-1) + nconti
        for j = 1:(deg-1)
            knots(ii + j) = meshP(i)
        end
    end
endfunction


// Returns numP Gaussian points mapped between xL and xR
function x = getGaussPts(xL, xR, numP)

    // Calculate midpoint and radius of interval
    mid = (xR + xL)/2.0
    radius = abs(xR - mid)

    // p determines how many collocation points we have
    select numP
    case 2 then
        x(1) = -(1/sqrt(3)) * radius + mid
        x(2) = (1/sqrt(3)) * radius + mid
    case 3 then
        x(1) = -sqrt(3/5) * radius + mid
        x(2) = mid
        x(3) = sqrt(3/5) * radius + mid
    case 4 then
        x(1) = -sqrt((3/7)+(2/7)*sqrt(6/5)) * radius + mid
        x(2) = -sqrt((3/7)-(2/7)*sqrt(6/5)) * radius + mid
        x(3) = sqrt((3/7)-(2/7)*sqrt(6/5)) * radius + mid
        x(4) = sqrt((3/7)+(2/7)*sqrt(6/5)) * radius + mid
    case 5 then
        x(1) = -(1/3) * sqrt(5+2*sqrt(10/7)) * radius + mid
        x(2) = -(1/3) * sqrt(5-2*sqrt(10/7)) * radius + mid
        x(3) = mid
        x(4) = (1/3) * sqrt(5-2*sqrt(10/7)) * radius + mid
        x(5) = (1/3) * sqrt(5+2*sqrt(10/7)) * radius + mid
    case 6 then
        // Values taken from
        // https://pomax.github.io/bezierinfo/legendre-gauss.html
        x(1) = -0.9324695142031521 * radius + mid
        x(2) = -0.6612093864662645 * radius + mid
        x(3) = -0.2386191860831969 * radius + mid
        x(4) = 0.2386191860831969 * radius + mid
        x(5) = 0.6612093864662645 * radius + mid
        x(6) = 0.9324695142031521 * radius + mid
    case 7 then
        // Values taken from
        // https://pomax.github.io/bezierinfo/legendre-gauss.html
        x(1) = -0.9491079123427585 * radius + mid
        x(2) = -0.7415311855993945 * radius + mid
        x(3) = -0.4058451513773972 * radius + mid
        x(4) = mid
        x(5) = 0.4058451513773972 * radius + mid
        x(6) = 0.7415311855993945 * radius + mid
        x(7) = 0.9491079123427585 * radius + mid
    case 8 then
        x(1) = -0.9602898564975363 * radius + mid
        x(2) = -0.7966664774136267 * radius + mid
        x(3) = -0.5255324099163290 * radius + mid
        x(4) = -0.1834346424956498 * radius + mid
        x(5) = 0.1834346424956498 * radius + mid
        x(6) = 0.5255324099163290 * radius + mid
        x(7) = 0.7966664774136267 * radius + mid
        x(8) = 0.9602898564975363 * radius + mid
    case(9) then
        x(1) = -0.9681602395076261 * radius + mid
        x(2) = -0.8360311073266358 * radius + mid
        x(3) = -0.6133714327005904 * radius + mid
        x(4) = -0.3242534234038089 * radius + mid
        x(5) = mid
        x(6) = 0.3242534234038089 * radius + mid
        x(7) = 0.6133714327005904 * radius + mid
        x(8) = 0.8360311073266358 * radius + mid
        x(9) = 0.9681602395076261 * radius + mid
    else
        disp(string(numP) + " Gauss points requested. Only 2-9 is supported.")
        disp("Aborting execution.")
        abort
    end
endfunction

// Returns ileft as required by the B-spline basis functions. Input x is the point in
     a domain
// where we are evaluating, kts is the knot sequence of that domain, and boolean isY
    specifies
// if it is the x or y domain.
function i = getileft(x, kts, isY)
    // Default to being in x
    if argn(2) < 3 then
        isY = %F
    end

    if ~isY then
        ind = getInd(x, meshX, N)
        i = nconti + (p-1) * ind
    else
        ind = getInd(x, meshY, M)
        i = nconti + (q-1) * ind
    end

endfunction

// Simplified call to bsplvd which evaluates the nder-1th derivative
// of the degree p/q b-splines at pt associated with the global
// knotsX/Y. Set isY to true to evaluate a y value.
function y = bsplv(pt, nder, isY)
    if argn(2) < 3 then
        isY = %F
    end
    if isY then
        knots = knotsY
        deg = q
    else
        knots = knotsX
        deg = p
    end
    indS = (deg+1)*(nder-1)+1
    indE = indS + deg
    y = bsplvd(pt, nder, knots, deg, 2, isY)(indS:indE)
endfunction
```

```
// Calls Fortran bsplvd to evaluate the b-splines associated with
// knots kts, and nconti internal continitui conditions. Returns
// the nder-1th derivative of degree deg b-splines at x. Set isY to
// true if evaluating a Y value.
function y = bsplvd(x, nder, kts, deg, nconti, isY)
    if argn(2) == 5 then
        isY = %F
    end
    ileft = getileft(x, kts, isY)
    k = deg + nconti - 1
    vnikx = zeros(nder *k, 1)
    call('bsplvd',kts,1,'d',k,2,'i',x,3,'d',ileft,4,'i',..
    vnikx,5,'d',nder,6,'i')
    y = vnikx
endfunction

// Returns the mesh index of point val within mesh meshP, with max
// of nint.
function i = getInd(val, meshP, nint)
    if argn(2) == 1 then
        meshP = meshX
        nint = N
    end
    i = 1
    while val >= meshP(i+1) && i < nint
        i = i + 1
    end
endfunction

// Change to the temp directory and create bsplvd.f and bsplvn.f
cd(TMPDIR)

bd = ['      SUBROUTINE BSPLVD ( XT, K, X, ILEFT, VNIKX, NDERIV )'
'        implicit none'
'      INTEGER K,NDERIV,ILEFT'
'      DOUBLE PRECISION X'
'      DOUBLE PRECISION XT(*),VNIKX(K,NDERIV)'
'      INTEGER KO,IDERIV,IDERVM,KMD,JM1,IPKMD,JLOW'
'      DOUBLE PRECISION A(20,20)'
'      DOUBLE PRECISION FKMD,DIFF,V'
'      DOUBLE PRECISION ZERO, ONE'
'      PARAMETER (ZERO = 0.D0)'
'      PARAMETER (ONE  = 1.D0)'
'      INTEGER I,J,M,L'
'      KO = K + 1 - NDERIV'
'      CALL BSPLVN(XT,KO,1,X,ILEFT,VNIKX(NDERIV,NDERIV))'
'      IF (NDERIV .LE. 1) GO TO 130'
'      IDERIV = NDERIV'
'      DO 20 I=2,NDERIV'
'        IDERVM = IDERIV-1'
'        DO 10 J=IDERIV,K'
'          VNIKX(J-1,IDERVM) = VNIKX(J,IDERIV)'
'   10   CONTINUE'
'        IDERIV = IDERVM'
'        CALL BSPLVN(XT,0,2,X,ILEFT,VNIKX(IDERIV,IDERIV))'
'   20 CONTINUE'
'      DO 40 I=1,K'
'        DO 30 J=1,K'
'          A(I,J) = ZERO'
'   30   CONTINUE'
'        A(I,I) = ONE'
'   40 CONTINUE'
'      KMD = K'
'      DO 120 M=2,NDERIV'
'        KMD = KMD - 1'
'        FKMD =  DBLE(KMD)'
'        I = ILEFT'
'        J = K'
'   50   CONTINUE'
'        JM1 = J-1'
'        IPKMD = I + KMD'
'        DIFF = XT(IPKMD) -XT(I)'
'        IF (JM1 .NE. 0) THEN'
'          IF (DIFF .NE. ZERO) THEN'
'            DO 60 L=1,J'
'              A(L,J) = (A(L,J) - A(L,J-1))/DIFF*FKMD'
'   60       CONTINUE'
'          ENDIF'
'          J = JM1'
'          I = I - 1'
'          GO TO 50'
'         ENDIF'
'         IF (DIFF .NE. ZERO) THEN'
'           A(1,1) = A(1,1)/DIFF*FKMD'
'         ENDIF'
'         DO 110 I=1,K'
'           V = ZERO'
'           JLOW = MAX(I,M)'
'           DO 100 J=JLOW,K'
'             V = A(I,J)*VNIKX(J,M) + V'
'  100      CONTINUE'
'           VNIKX(I,M) = V'
'  110   CONTINUE'
'  120 CONTINUE'
'  130 RETURN'
'      END'];

bn = ['      SUBROUTINE BSPLVN ( XT, JHIGH, INDEX, X, ILEFT, VNIKX )'
'        implicit none'
'      DOUBLE PRECISION XT(*),X,VNIKX(*)'
'      INTEGER JHIGH,INDEX,ILEFT'
'      INTEGER IPJ,IMJP1,JP1,JP1ML'
'      DOUBLE PRECISION VMPREV,VM'
'      DOUBLE PRECISION ZERO, ONE'
'      PARAMETER (ZERO = 0.D0)'
'      PARAMETER (ONE  = 1.D0)'
'      DOUBLE PRECISION DELTAM(20),DELTAP(20)'
'      INTEGER J'
'      INTEGER L'
'      DATA J/1/,DELTAM/20*0.D+0/,DELTAP/20*0.D+0/'
'      IF(INDEX.EQ.1) THEN'
'        J = 1'
'        VNIKX(1) = ONE'
'        IF (J .GE. JHIGH) GO TO 40'
'      ENDIF'
'   20 CONTINUE'
'      IPJ = ILEFT+J'
'      DELTAP(J) = XT(IPJ) - X'
'      IMJP1 = ILEFT-J+1'
'      DELTAM(J) = X - XT(IMJP1)'
'      VMPREV = ZERO'
'      JP1 = J+1'
'      DO 30 L=1,J'
'        JP1ML = JP1-L'
'        VM = VNIKX(L)/(DELTAP(L) + DELTAM(JP1ML))'
'        VNIKX(L) = VM*DELTAP(L) + VMPREV'
'        VMPREV = VM*DELTAM(JP1ML)'
'   30 CONTINUE'
'      VNIKX(JP1) = VMPREV'
'      J = JP1'
'      IF (J .LT. JHIGH) GO TO 20'
'   40 RETURN'
'      END'];

mputl(bd, 'bsplvd.f');
mputl(bn, 'bsplvn.f');

// Link the newly created bsplvn.f and bsplvd.f
ilib_for_link(['bsplvd', 'bsplvn'], [..
'bsplvd.f', 'bsplvn.f'], [], "f");
exec loader.sce;
linked = %T;
```

core1D.sci

```
global Y_a
global Y_b

// Evaluates the collocation solution at point x with coefficients coef.
function y=Y(coef, x)
    y = beval(coef, x, 1)
endfunction

// Evaluates the first derivative of the collocation solution at point x
// with coefficients coef.
function y=Yx(coef, x)
    y = beval(coef, x, 2)
endfunction

// Evaluates the second derivative of the collocation solution at point x
// with coefficients coef.
function y=Yxx(coef, x)
    y = beval(coef, x, 3)
endfunction

// Evaluates the (d-1)th derivative of the collocation solution at point x
// with coefficients coef.
function y = beval(coef, x, d)
    basis = bsplv(x, d)
    sum = 0
    for i = 1:p+1
        sum = sum + coef(i) * basis(i)
    end
    y = sum
endfunction

// Creates and returns the mesh points in x. Contains deprecated uneven mesh creation
    .
function x = buildMesh(levels)
    // Calculate the mesh points
    x = linspace(A, B, N+1);

    // Set the knots
    global knotsX
    knotsX = buildKnots(x)
endfunction

// Returns the indexes within the Jacobian that are non-zero for the
// cIndth coefficient.
function inds = nonZeroInds(cInd)
    if cInd == 1 then
        bndInd  = 1
        ind = 1
    elseif cInd == ncpts
        bndInd = ncpts
        ind = N
    else
        bndInd = []
        ind = getInt(cInd)
    end
    colInds = colInds(ind)
    inds = cat(1, colInds, bndInd)
endfunction

// Returns the index of the collocation points in the xIndsth intervals.
function inds = colInds(xInds)
    if size(xInds)(1) == 2 then
        inds1 = colIndsI(xInds(1))
        inds2 = colIndsI(xInds(2))
    else
        inds1 = colIndsI(xInds)
        inds2 = []
    end
    inds = cat(1, inds1, inds2)
endfunction

// Returns the index of the collocation points on the xIndth interval.
function inds = colIndsI(xInd)
    startInd = 2 + (xInd - 1) * (p-1)
    endInd = startInd + (p - 2)
    inds = (startInd:endInd)'
endfunction

// Returns the intervals which are affected by the xith coefficient.
function ind = getInt(xi)
    if xi <= p - 1 then
        ind = 1
    elseif xi >= ncpts - 2
        ind = N
    else
        ind = 1
        xi = xi - (p-1)
        while xi > 0
            xi = xi - nconti
            if xi < 1 then
                if ind + 1 <= N then
                    ind(2) = ind + 1
                end
                break
            elseif xi <= (p + 1 - 2 * nconti) then
                ind = ind + 1
                break
            end
            xi = xi - (p + 1 - 2 * nconti)
            ind = ind + 1
        end
    end
endfunction

// Returns the index of the first coefficient for the ith interval.
function ind = firstCoef(i)
    ind = (p-1) * (i - 1) + 1
endfunction

// Returns the index of the last coefficient for the ith interval.
function ind = last(i)
    ind = firstCoef(i) + p
endfunction

// Evaluates the curent collocation solution at point x.
function y = U(x)
    // Determine which interval X is in
    index = getInd(x)

    // Calculate using the appropriate coeeficients
    y = Y(coeffs(firstCoef(index):last(index)), x)
endfunction

// Evaluates the first derivative of the collocation solution at point x.
function y = Ux(x)
    index = getInd(x)

    y = Yx(coeffs(firstCoef(index):last(index)), x)
endfunction

// Evaluates the second derivative of the collocation solution at point x.
function y = Uxx(x)
    index = getInd(x)

    y = Yxx(coeffs(firstCoef(index):last(index)), x)
endfunction
```

core2D.sci

```
// Global variables used for 2D collocation
global q M // Degree of solution and number of intervals in y
global C D // Lower and upper bound on y
global meshY // The mesh points in y
global knotsX knotsY // The knots in x and y used for creation of the b-splines
global colX colY // Collocation points in x and y
global ncpts ncptX ncptY
global bndAvals bndBvals bndCvals bndDvals
global colXBasisVals colYBasisVals
global meshXBasisVals meshYBasisVals
global colInds xBndInds yBndInds cornInds // The 1D index of conditions
    // colInds(i, j, :) is the index of the (p-1)*(q-1) col. conds. in
    // rectangle i, j.
    // x/yBndInds(i, j, k) is the index of the (q-1)/(p-1) boundary conditions
    // for the jth interval in x/y. i = 0 corresponds to x=A/y=C, and i = 1
    // corresponds to x=B/y=D

// Evaluates the tensor product of b-splines at (x, y).
// derX/Y designate the desired order -1 of derivative with respect
// to X/Y
function z = U(x, y, derX, derY)
    if argn(2) == 2 then
        derX = 1
        derY = 1
    end
    co = getCoefs(getInd(x, meshX, N), getInd(y, meshY, M))
    bv = getBasisVals(x, y, derX, derY)
    z = sum(co .* bv)
endfunction

// Retrieve the apropriate coefficients fpr the potentially non-zero
// b-spline basis functions in mesh square xInd, yInd. coefs is the
// vector containing all of the coefficients. co(i, j) contains the
// coefficient for the ith x b-spline multiplied by the jth b-spline.
function co = getCoefs(xInd, yInd)
    xStart = (xInd - 1) * (p - 1) + 1
    yStart = (yInd - 1) * (q - 1) + 1
    co = coefSq(xStart, yStart)
endfunction

// Calculate the product of the potential non-zero b-spline basis
// functions at point (x, y). derX/Y specifies which order of
// derivative with respect to X/Y. bv(i, j) will contain the product
// of the ith x b-spline multiplied by the jth b-spline.
function bv = getBasisVals(x, y, derX, derY)

    // Check if x is col pt or mesh val
    // If so then we can use the saved evaluation
    xInd = getInd(x, meshX, N)
    yInd = getInd(y, meshY, M)
    if x == meshX(xInd) then
        xVals = matrix(meshXBasisVals(xInd, :, derX), 1, p+1)'
    elseif x == meshX(xInd+1) then
        xVals = matrix(meshXBasisVals(xInd+1, :, derX), 1, p+1)'
    else
        for i = 1:(p-1)
            if x == colX(xInd, i) then
                xVals = matrix(colXBasisVals(xInd, i, :, derX), 1, p+1)'
                i = -1
            end
        end

        // Calculate it if it isn't a mesh or col pt
        if i ~= -1 then
            xVals = bsplv(x, derX)
        end
    end

    // Check if y is col pt or mesh val
    // If so then we can use the saved evaluation
    if y == meshY(yInd) then
        yVals = matrix(meshYBasisVals(yInd, :, derY), 1, q+1)'
    elseif y == meshY(yInd+1) then
        yVals = matrix(meshYBasisVals(yInd+1, :, derY), 1, q+1)'
    else
        for i = 1:(q-1)
            if y == colY(yInd, i) then
                yVals = matrix(colYBasisVals(yInd, i, :, derY), 1, q+1)'
                i = -1
            end
        end

        // Calculate it if it isn't a mesh or col pt
        if i ~= -1 then
            yVals = bsplv(y, derY, %T)
        end
    end
    bv = xVals * yVals'
endfunction

// Returns a square of coefficients (p+1) by (q+1)
// starting at firstX, firstY in x and y respectively.
function c = coefSq(firstX, firstY)
    c(p+1,q+1) = 0
    for i = 1:(p+1)
        os = ncptY * (firstX + i - 2)
        for j = 1:(q+1)
            c(i,j) = coeffs(os + (firstY + j - 1))
        end
    end
endfunction

// Sets the global meshes using the global A, B, C
// D, N, and M.
function prepareMesh()
    global meshX meshY
    meshX = linspace(A, B, N+1)
    meshY = linspace(C, D, M+1)
endfunction

// Sets the global knots variables using global meshX/Y, p, q, N, M.
function prepareKnots()
    global knotsX knotsY
    knotsX = buildKnots(meshX, p, N)
    knotsY = buildKnots(meshY, q, M)
endfunction

// Sets the global colX/Y variables using p, N, meshX, q, M, and
// meshY.
```

```
function prepareColP()
    global colX colY
    colX = zeros(N, p-1)
    colY = zeros(M, q-1)
    for i = 1:N
        colX(i,:) = getGaussPts(meshX(i), meshX(i+1), p-1)
    end
    for i = 1:M
        colY(i,:) = getGaussPts(meshY(i), meshY(i+1), q-1)
    end
endfunction

// Save the evaluations of basis functions which will be repeatedly used
// in colConds() to increase the efficiency.
function setSavedBasisVals()
    global colXBasisVals meshXBasisVals p q N M
    global colYBasisVals meshYBasisVals
    colXBasisVals = zeros(N, p-1, p+1, 3)
    colYBasisVals = zeros(M, q-1, q+1, 3)
    meshXBasisVals = zeros(N+1, p+1, 3)
    meshYBasisVals = zeros(M+1, q+1, 3)

    // Loop through the collocation points in x and save the basis values
    for i = 1:N
        for j = 1:(p-1)
            colXBasisVals(i, j, :, :) = bsplvSave(colX(i, j))
        end
    end

    // Loop through mesh points in x
    for i = 1:N+1
        meshXBasisVals(i, :, :) = bsplvSave(meshX(i))
    end

    // Loop through the collocation points in y and save the basis values
    for i = 1:M
        for j = 1:(q-1)
            colYBasisVals(i, j, :, :) = bsplvSave(colY(i, j), %T)
        end
    end

    // Loop through mesh points in y
    for i = 1:M+1
        meshYBasisVals(i, :, :) = bsplvSave(meshY(i), %T)
    end

endfunction

// Version of bsplv() to be called by setSavedBasisVals() which
// returns the solution value, first, and second derivatives.
function y = bsplvSave(pt, isY)
    if argn(2) < 2 then
        isY = %F
    end
    if isY then
        global q knotsY
        knots = knotsY
        deg = q
    else
        global p knotsX
        knots = knotsX
        deg = p
    end
    vec = bsplvd(pt, 3, knots, deg, 2, isY)
    y(:,1) = vec(1:deg+1)
    y(:,2) = vec(deg+2:2*deg+2)
    y(:,3) = vec(2*deg+3:3*deg+3)
endfunction

// Saves the boundary value at the collocation points into the global
// bnd_vals variables.
function saveBoundaryEvals()
    global bndAvals bndBvals bndCvals bndDvals
    for i = 1:M
        for j = 1:(q-1)
            bndAvals(i, j) = bndA(colY(i, j))
            bndBvals(i, j) = bndB(colY(i, j))
        end
    end
    for i = 1:N
        for j = 1:(p-1)
            bndCvals(i, j) = bndC(colX(i, j))
            bndDvals(i, j) = bndD(colX(i, j))
        end
    end
endfunction

// Converts the 1-dimensional index of a condition to a
// 2-dimensional coordinate
function [i, j] = toCoord(ind)
    i = ceil(ind/ncptY)
    j = pmodulo(ind-1, ncptY) + 1
endfunction

// Returns the indices of the jacobian which will be non zero for a given
// coefficient. Uses the pre-calculated indices provided.
function inds = nonZeroInds(cInd, colInds, xBndInds, yBndInds, cornInds)

    // Get the coordinate of the coefficient
    [xi, yi] = toCoord(cInd)

    // Determine what conditions are needed
    if xi == 1 then // On the low X boundary
        if yi == 1 then // On the low Y boundary
            cornInd = cornInds(1, 1)
            bndYinds = matrix(yBndInds(1, 1, :), 1, p-1)
            yInd = 1

        elseif yi == ncptY then // On the high Y boundary
            cornInd = cornInds(1, 2)
            bndYinds = matrix(yBndInds(2, 1, :), 1, p-1)
            yInd = M

        else // Not on a Y boundary
            bndYinds = []
            cornInd = []
            yInd = getYint(yi)
        end
        bndXinds = matrix(xBndInds(1, yInd, :), 1, (q-1)*size(yInd)(1))
        colInds = matrix(colInds(1, yInd, :), 1, (p-1)*(q-1)*size(yInd)(1))

    elseif xi == ncptX then // On the high X boundary
        if yi == 1 then // On the low Y boundary
            cornInd = cornInds(2, 1)
            bndYinds = matrix(yBndInds(1, N, :), 1, p-1)
            yInd = 1

        elseif yi == ncptY then // On the high Y boundary
            cornInd = cornInds(2, 2)
            bndYinds = matrix(yBndInds(2, N, :), 1, p-1)
            yInd = M

        else // Not on a Y boundary
            bndYinds = []
            cornInd = []
            yInd = getYint(yi)
        end
        bndXinds = matrix(xBndInds(2, yInd, :), 1, (q-1)*size(yInd)(1))
        colInds = matrix(colInds(N, yInd, :), 1, (p-1)*(q-1)*size(yInd)(1))

    else // Not on an X boundary
        xInd = getXint(xi)
        if yi == 1 then // On the low Y boundary
            bndYinds = matrix(yBndInds(1, xInd, :), 1, (p-1)*size(xInd)(1))
            yInd = 1

        elseif yi == ncptY then // On the high Y boundary
            bndYinds = matrix(yBndInds(2, xInd, :), 1, (p-1)*size(xInd)(1))
            yInd = M

        else // Not on a Y boundary
            bndYinds = []
            yInd = getYint(yi)
        end
        cornInd = []
        bndXinds = []
        colInds = matrix(colInds(xInd, yInd, :), 1, (p-1)*(q-1)*size(xInd)(1)*size(
            yInd)(1))
    end

    // Concatenate the various indices and return them
    inds = cat(1, cornInd, bndXinds', bndYinds', colInds')
endfunction

// Returns the interval(s) in X which coefficient xi will affect.
```

```
function indX = getXint(xi)
    if xi <= (p + 1 - nconti) then
        indX = 1
    elseif xi >= ncptX - nconti
        indX = N
    else
        indX = 1
        xi = xi - (p - 1)
        while xi > 0
            xi = xi - nconti
            if xi < 1 then
                if indX + 1 <= N then
                    indX(2) = indX + 1
                end
                break
            elseif xi <= (p + 1 - 2 * nconti) then
                indX = indX + 1
                break
            end
            xi = xi - (p + 1 - 2 * nconti)
            indX = indX + 1
        end
    end
endfunction

// Returns the interval(s) in Y which coefficient yi will affect.
function indY = getYint(yi)
    if yi <= (q + 1 - nconti) then
        indY = 1
    elseif yi >= ncptY - nconti
        indY = M
    else
        indY = 1
        yi = yi - (q-1)
        while yi > 0
            yi = yi - nconti
            if yi < 1 then
                if indY + 1 <= M then
                    indY(2) = indY + 1
                end
                break
            elseif yi <= (q + 1 - 2 * nconti) then
                indY = indY + 1
                break
            end
            yi = yi - (q + 1 - 2 * nconti)
            indY = indY + 1
        end
    end
endfunction

//  Calculates the indices of the collocation conditions which occur in
// the rectangle with coordinate xInd, yInd
function inds = getColInds(xInd, yInd)
    ind = 1
    sqOs = ncptY * ((p-1) * (xInd - 1) + 1) + 1 + (q-1) * (yInd - 1)
    for i = 1:p-1
        colLOs = (i-1) * ncptY + sqOs
        for j = 1:q-1
            inds(ind) = colLOs + j
            ind = ind + 1
        end
    end
endfunction

// Calculates the indices of the conditions which correspond to the X boundary.
// yInd is the interval in Y, while isHigh determines if it is the low or
// high X boundary (%F / %T)
function inds = getxBndInds(yInd, isHigh)
    ind = 1
    if ~isHigh then
        sqOs = (q-1) * (yInd - 1) + 1
    else
        sqOs = (((p-1) * N) + 1) * ncptY + (q-1) * (yInd - 1) + 1
    end
    for i = 1:(q-1)
        inds(ind) = sqOs + i
        ind = ind + 1
    end
endfunction

// Calculates the indices of the conditions which correspond to the Y boundary.
// xInd is the interval in X, while isHigh determines if it is the low or
// high Y boundary (%F / %T)
function inds = getyBndInds(xInd, isHigh)
    os = ncptY * ((xInd - 1) * (p-1) + 1)
    if ~isHigh then
        os = os - (ncptY - 1)
    end
    ind = os
    for i = 1:(p-1)
        ind = ind + ncptY
        inds(i) = ind
    end
endfunction

// Returns the index of the corner condition specified by xHigh, yHigh
// xHigh / yHigh = %T means it is the corner with the high X / Y boundary.
function ind = getCornerInd(xHigh, yHigh)
    if ~xHigh then
        if ~yHigh then
            ind = 1
        else
            ind = ncptY
        end
    else
        ind = ncpts
        if ~yHigh then
            ind = ind - ncptY + 1
        end
    end
endfunction

// Calculates and returns the indices of the collocation conditions for
// all N*M rectangles
function colInds = saveColInds()
    colInds = zeros(N, M, (p-1) * (q-1))
    for i = 1:N
        for j = 1:M
            colInds(i, j, :) = getColInds(i, j)
        end
    end
endfunction

// Calculates and returns all of the X boundary condition indices.
function xBndInds = savexBndInds()
    xBndInds = zeros(2, M, q-1)
    for i = 1:M
        xBndInds(1, i, :) = getxBndInds(i, %F)
        xBndInds(2, i, :) = getxBndInds(i, %T)
    end
endfunction

// Calculates and returns all of the Y boundary condition indices.
function yBndInds = saveyBndInds()
    yBndInds = zeros(2, N, p-1)
    for i = 1:N
        yBndInds(1, i, :) = getyBndInds(i, %F)
        yBndInds(2, i, :) = getyBndInds(i, %T)
    end
endfunction

// Calculates and returns all of the corner condition indices.
function cornInds = saveCornInds()
    cornInds(1, 1) = getCornerInd(%F, %F)
    cornInds(1, 2) = getCornerInd(%F, %T)
    cornInds(2, 1) = getCornerInd(%T, %F)
    cornInds(2, 2) = getCornerInd(%T, %T)
endfunction

// Returns which line of x a point is in along with which point in the
// line it is.
function [l, r] = getLine(ind)
    l = 1
    while ind > ncptY
        ind = ind - ncptY
        l = l + 1
    end
    r = ind
endfunction

// Returns the index of a collocation point for jacCond()
function [inter, col] = getColPt(rem, isY)
    // Set variables for x or y
    if argn(2) < 2 then
        isY = %F
    end
    if isY then
        deg = q
    else
        deg = p
    end

    // Find which interval and which col pt within the interval
    col = rem - 1
    inter = 1
    while col > (deg -1)
        inter = inter + 1
        col = col - (deg - 1)
    end
endfunction

// Quicker version of U to be used when evaluating the collocation solution at
// a collocation or boundary point in X and Y
function z = savedU(indX, indY, dx, dy)
    if argn(2) == 2 then
        dx = 1
        dy = 1
    end
    if indX == 1 then
        xVals = matrix(meshXBasisVals(1, :, dx), 1, p+1)'
        interX = 1
    elseif indX == ncptX then
        xVals = matrix(meshXBasisVals(N+1, :, dx), 1, p+1)'
        interX = N
    else
        [interX, pt] = getColPt(indX, %F)
        xVals = matrix(colXBasisVals(interX, pt, :, dx), 1, p+1)'
    end
    if indY == 1 then
        yVals = matrix(meshYBasisVals(1, :, dy), 1, q+1)'
        interY = 1
    elseif indY == ncptY then
        yVals = matrix(meshYBasisVals(M+1, :, dy), 1, q+1)'
        interY = M
    else
        [interY, pt] = getColPt(indY, %T)
        yVals = matrix(colYBasisVals(interY, pt, :, dy), 1, q+1)'
    end
    bv = xVals * yVals'
    co = coefSq((interX-1)*(p-1)+1,(interY-1)*(q-1)+1)
    z = sum(co .* bv)
endfunction
```

coreBVODE.sci

```
// Function to build system passed to fsolve
function [func] = fsol(coef)

    // Index to facilitate population of system
    index = 1

    // Two boundary conditions
    // -------------------------------------------------------------------------
    func(index) = Y(coef(1:(p+1)), meshX(1)) - Y_a
    index = index+1

    // --------------- end boundary conditions -------------------------------

    // Collocation conditions
    // ------------------------------------------------------------------------
    // Loop through the N subintervals
    for i = 1:N
        // Loop for each collocation point per subinterval
        for j = 1:(p-1)
            func(index) = PDE(coef(firstCoef(i):last(i)), colX(i,j))
            index = index + 1
        end
    end
    // ----------- end collocation conditions ---------------------------------
    func(index) = Y(coef(firstCoef(N):last(N)), meshX(N+1)) - Y_b
    index = index+1
endfunction

function info = collocate(deg, nint)
```

```
    global N
    global p
    global colX
    global meshX
    global ncpts
    global coeffs

    p = deg

    N = nint
    ncpts = N * (p-1) + nconti
    meshX = buildMesh()

    if verbose > 0 then
        disp("probNum=" + string(probNum) + " p=" + string(p) + " N=" + string(N))
    end

    colX = zeros(N, p-1)
    for i = 1:N
        colX(i,:) = getGaussPts(meshX(i), meshX(i+1), p-1);
    end
    global Y_a Y_b
    Y_a = actual(A) // Y value at A boundary
    Y_b = actual(B)
    // Calculate Jacobian

    // Generate a first guess
    fg = ones((p-1)*N+2,1);
    fg(1) = Y_a
    fg((p-1) * N + nconti) = Y_b
    // Send system to fsolve
    if verbose > 0 then
        disp("Solving for coefficients.")
        if verbose > 1 then
            tic
        end
    end

    if jacMode == 0 then
        [coeffs, v, info] = fsolve(fg, fsol, atol*0.1)
    elseif jacMode == 1 then
        [coeffs, v, info] = fsolve(fg, fsol, fjac, atol*0.1)
    end

    if verbose > 0 then
        disp("Coefficients set.")
        if verbose > 1 then
            time = toc()
            disp("Elapsed time: " + string(time) + " seconds")
        end
        disp("fsolve info: " + string(info))
        disp("fsolve max residual: " + string(max(abs(fsol(coeffs)))))
    end
endfunction

// Approximates the Jacobian for fsolve
function r = fjac(coefs)
    global coeffs t0 dy
    inds = nonZeroInds(1)
    coeffs = coefs

    for k = 1:size(inds)(1)
            ind = inds(k)
            coeffs(1) = coefs(1)+dy
            rp1 = condI(ind)
            coeffs(1) = coefs(1)-dy
            r(ind,1) = (0.5*rp1 - 0.5*condI(ind))/dy
    end
    for i = 2:ncpts
        inds = nonZeroInds(i)
        coeffs(i-1) = coefs(i-1)
        for k = 1:size(inds)(1)
            ind = inds(k)
            coeffs(i) = coefs(i)+dy
            rp1 = condI(ind)
            coeffs(i) = coefs(i)-dy
            r(ind,i) = (0.5*rp1 - 0.5*condI(ind))/dy
        end
    end
endfunction

// Returns a single entry of the residual
```

```
function r = condI(ind)
    global coeffs
    if ind == 1 then
        r = Y(coeffs(1:(p+1)), meshX(1)) - Y_a
    elseif ind == ncpts then
        r = Y(coeffs(firstCoef(N):last(N)), meshX(N+1)) - Y_b
    else
        i = ceil((ind-1)/(p-1))
        j = ind-1-(i-1)*(p-1)
        r = PDE(coeffs(firstCoef(i):last(i)), colX(i,j))
    end
endfunction
```

corePPDE.sci

```
global t0 // The initial time for time integration
global errScheme // The currently 'active' error scheme. 1 = New LOI, 2 = LOI, 3 =
    SCI
global tc // The time which global coeffs is a collocation solution for
global colPtBasisVals // (i, j, k, l) Stores the value of the (k-1)th
    // derivative of the lth non-zero b-spline basis function at the jth
    // collocation point within the ith interval

// Functions for reducing calls to B-spline functions
function y = bsplvSave(pt, nder)
    vec = bsplvd(pt, nder, knotsX, p, 2)
    y = zeros(p+1, 3)
    y(:,1) = vec(1:p+1)
    if nder >= 2 then
        y(:,2) = vec(p+2:2*p+2)
    end
    if nder >= 3 then
        y(:,3) = vec(2*p+3:3*p+3)
    end
endfunction

function y = fastColPtU(i, j)
    y = fastColPtEval(i, j, 1)
endfunction

function y = fastColPtUx(i, j)
    y = fastColPtEval(i, j, 2)
endfunction

function y = fastColPtUxx(i, j)
    y = fastColPtEval(i, j, 3)
endfunction

function y = fastColPtEval(i, j, nder)
    coInd = firstCoef(i)
    y = sum(coeffs(coInd:coInd+p) .* matrix(colPtBasisVals(i, j, nder, :), p+1))
endfunction

function saveColPtBasisVals(nder)
    global colPtBasisVals
    colPtBasisVals = zeros(N, p-1, 3, p+1)
    for i = 1:N
        for j = 1:(p-1)
            colPtBasisVals(i, j, :, :) = bsplvSave(colX(i, j), nder)'
        end
    end
endfunction
// -------------

// The residual function for setting the initial temporal derivative of the
    coefficients
function r = fsolD(coefDer)
    r = res(t0, coeffs, coefDer)
endfunction

// Function to build residual system passed to fsolve when setting initial
    coefficients
function r = fsol(coef)
    global coeffs
    coeffs = coef
    r(1) = uinit(A) - U(A)
    ind = 2
    for i = 1:N
        for j = 1:(p-1)
```

```
            pt = colX(i, j)
            r(ind) = uinit(pt) - U(pt)
            ind = ind + 1
        end
    end
    r(ncpts) = uinit(B) - U(B)
endfunction

// Takes in the derivative of the coefficients w.r.t time and returns the temporal
    derivative of the collocation
// solution at the evaluation points
function ptDer = getPtDer(coDer)
    global coeffs
    temp = coeffs
    coeffs = coDer
    saveColPtBasisVals(1)
    for i = 1:N
        for j = 1:(p-1)
            ptDer((i-1) * (p-1) + j) = fastColPtU(i, j)
        end
    end
    coeffs = temp
endfunction

// Generates a first guess for the initial coefficients
function coeffs = iniCoeffs()
    coeffs = ones((p-1)*N+2,1);
    coeffs(1) = uinit(A)
    coeffs(ncpts) = uinit(B)
    for i = 1:N
        for j = 1:(p-1)
            coeffs((i-1)*(p-1)+j+1) = uinit(colX(i, j))
        end
    end
endfunction

// Sets the initial coefficient in the global coeffs
function initCoeffs()
    fg = iniCoeffs()
    global coeffs
    if jacMode == 0 then
        [coeffs,n,inf] = fsolve(fg, fsol, 1.d-14)
    elseif jacMode == 1 then
        [coeffs,n,inf] = fsolve(fg, fsol, fjac, 1.d-14)
    end
    if verbose > 1 then
        disp("fsolve info: " + string(inf))
        disp("fsolve residual: " + string(max(abs(fsol(coeffs)))))
    end
endfunction

// Approximates the Jacobian for fsolve when setting the initial coefficients
function r = fjac(coefs)
    y = coefs
    mockY = y
    ydot = zeros(ncpts,1)
    for i = 1:ncpts
        inds = nonZeroInds(i)
        mockY(i) = y(i)
        for k = 1:size(inds)(1)
            ind = inds(k)
            mockY(i) = y(i)+dy
            rp1 = resI(t0, mockY, ydot, ind)
            mockY(i) = y(i)-dy
            r(ind,i) = (0.5*rp1 - 0.5*resI(t0,mockY,ydot, ind))/dy
            mockY(i) = y(i)
        end
    end
endfunction

// Returns a single entry of the residual
function r = resI(t,y,ydot,ind)
    global coeffs
    coeffs = y
    if ind == 1 then
        r = bndxa(t, U(A), Ux(A))
    elseif ind == ncpts then
        r = bndxb(t, U(B), Ux(B))
    else
        ptDer = getPtDer(ydot)
        i = ceil((ind-1)/(p-1))
        j = ind-1-(i-1)*(p-1)
        r = f(t, colX(i, j), fastColPtU(i, j), fastColPtUx(i, j), fastColPtUxx(i, j))
             - ptDer(ind-1)
    end
endfunction

// Approximates the Jacobian used for setting the initial spatial derivative of the
    coefficients
function r = fjacD(ydot)
    global coeffs t0 dy
    inds = nonZeroInds(1)
    y = coeffs
    t = t0
    mockYdot = ydot

    for k = 1:size(inds)(1)
            ind = inds(k)
            mockYdot(1) = ydot(1)+dy
            rdp1 = resI(t, y, mockYdot, ind)
            mockYdot(1) = ydot(1)-dy
            r(ind,1) = (0.5*rdp1 - 0.5*resI(t, y, mockYdot, ind))/dy
    end
    for i = 2:ncpts
        inds = nonZeroInds(i)
        mockYdot(i-1) = ydot(i-1)
        for k = 1:size(inds)(1)
            ind = inds(k)
            mockYdot(i) = ydot(i)+dy
            rdp1 = resI(t, y, mockYdot, ind)
            mockYdot(i) = ydot(i)-dy
            r(ind,i) = (0.5*rdp1 - 0.5*resI(t, y, mockYdot, ind))/dy
        end
    end
endfunction

// Sets the initial temporal derivative of coefficients
function y0p = initCoeffsDer()
    fg = ones((p-1)*N+2, 1)
    if jacMode == 0 then
        [y0p,b,inf] = fsolve(fg, fsolD, atol * 0.1)
    elseif jacMode == 1 then
        [y0p,b,inf] = fsolve(fg, fsolD, fjacD, atol * 0.1)
    end

    if verbose > 0 then
        disp("fsolve info: " + string(inf))
        disp("fsolve max residual: " + string(max(abs(fsolD(y0p)))))
    end
endfunction

// Residual function for time integration
function [r, ires] = res(t, u, up)
    global coeffs
    ptDer = getPtDer(up)
    coeffs = u
    saveColPtBasisVals(3)
    r(1) = bndxa(t, U(A), Ux(A))
    ind = 2
    for i = 1:N
        for j = 1:(p-1)
            r(ind) = f(t, colX(i, j), fastColPtU(i, j), fastColPtUx(i, j),
                fastColPtUxx(i, j)) - ptDer(ind-1)
            ind = ind + 1
        end
    end
    r(ncpts) = bndxb(t, U(B), Ux(B))
    ires = 0
endfunction

function coefs = collocate(deg, nint, ti, tOut)
    // Calculate the collocation points
    global N
    global p
    global colX
    global meshX
    global ncpts
    global numPts
    global u0 allPts t0 tc
    t0 = ti
    tc = t0
    p = deg
    N = nint
    ncpts = N * (p-1) + nconti
    setBnds()
    meshX = buildMesh()
    if verbose > 0 then
        disp("probNum=" + string(probNum) + " p=" + string(p) + " N=" + string(N))
        if verbose > 1 then
            tic()
        end
    end

    colX = zeros(N, p-1)
    for i = 1:N
        colX(i,:) = getGaussPts(meshX(i), meshX(i+1), p-1);
    end

    // Set the coefficients
    if verbose > 0 then
        disp("Calculating initial coefficients.")
        if verbose > 1 then
            tic
        end
    end
    initCoeffs()
    if verbose > 0 then
        disp("Coefficients set.")
        if verbose > 1 then
            time = toc()
            disp("Elapsed time: " + string(time) +" seconds")
            tic
        end
        disp("Calculating initial derivative of coefficients.")
    end
    coDer = initCoeffsDer()
    if verbose > 0 then
        disp("Derivative of coefficients set.")
        if verbose > 1 then
            time = toc()
            disp("Elapsed time: " + string(time) + " seconds")
            tic
        end
        disp("Beginning time integration.")
    end

    // Set DASSL options
    info = list([], 1, [], [], [], 0, -ones(ncpts,1), 0, 0, 0, 0, [], [], 0)
    ng = 0
    deff('[rts] = gr1(t, y)', 'rts = [1]')

   // Call DASSL
    if jacMode == 0 then
        [r, nn] = daskr([coeffs, coDer], t0, tOut, atol * 0.1, res, ng, gr1, info)
    elseif jacMode == 1 then
        [r, nn] = daskr([coeffs, coDer], t0, tOut, atol * 0.1, res, qjacRes, ng, gr1,
             info)
    end

    if verbose > 0 then
        disp("Time integration complete.")
        if verbose > 1 then
            time = toc()
            disp("Elapsed time: " + string(time) + " seconds")
        end
    end

    global coeffs
    for i = 1:max(size(tOut))
        coefs(:, i) = r(2:ncpts+1, i)
    end
    coeffs = r(2:ncpts+1, size(r)(2))
    tc = r(1, size(r)(2))
endfunction

// Efficient approximation of Jacobian requested by DASKR.
function r = qjacRes(t, y, ydot, cj)
    inds = nonZeroInds(1)
    mockY = y
    mockYdot = ydot

    for k = 1:size(inds)(1)
        ind = inds(k)
        mockY(1) = y(1)+dy
        mockYdot(1) = ydot(1)+dy
        rp1 = resI(t, mockY, ydot, ind)
        rdp1 = resI(t, y, mockYdot, ind)
        mockY(1) = y(1)-dy
        mockYdot(1) = ydot(1)-dy
        r(ind,1) = (0.5*rp1 - 0.5*resI(t,mockY,ydot, ind))/dy
        r(ind,1) = r(ind,1) + cj*(0.5*rdp1 - 0.5*resI(t, y, mockYdot, ind))/dy
    end
    for i = 2:ncpts
        inds = nonZeroInds(i)
        mockY(i-1) = y(i-1)
        mockYdot(i-1) = ydot(i-1)
        for k = 1:size(inds)(1)
            ind = inds(k)
            mockY(i) = y(i)+dy
            mockYdot(i) = ydot(i)+dy
            rp1 = resI(t, mockY, ydot, ind)
            rdp1 = resI(t, y, mockYdot, ind)
            mockY(i) = y(i)-dy
            mockYdot(i) = ydot(i)-dy
            r(ind,i) = (0.5*rp1 - 0.5*resI(t,mockY,ydot, ind))/dy
            r(ind,i) = r(ind,i) + cj*(0.5*rdp1 - 0.5*resI(t, y, mockYdot, ind))/dy
        end
    end
endfunction
```

coreEPDE.sci

```
// Calculates a collocation solution of degree p with N intervals
// in x, and degree q with M subintervals in y. Stores the calculated
// coefficients in global coeffs
//function info = collocate(degX, intX, degY, intY, useFast, verbose)
function info = collocate(degs, ints)
    degX = degs(1)
    if max(size(degs)) == 2 then
        degY = degs(2)
    else
        degY = degs(1)
    end
    intX = ints(1)
    if max(size(ints)) == 2 then
        intY = ints(2)
    else
        intY = ints(1)
    end

    // Set global variables
    global p N q M ncpts AC AD BC BD ncptX ncptY
    global colInds xBndInds yBndInds cornInds
    p = degX
    N = intX
    q = degY
    M = intY
    ncptY = (M * (q-1) + 2)
    ncptX = (N * (p-1) + 2)
    ncpts =  ncptX * ncptY
    if verbose > 0 then
        disp("p=" + string(p) + " N=" + string(N) + " q=" + string(q) + ..
    " M=" + string(M))
    end
    setBounds()

    if verbose > 0 then
        disp("Setting mesh, knot, and collocation points.")
    end
    // Set the mesh
    prepareMesh()

    // Set the knot sequence
    prepareKnots()

    // Set the collocation points
    prepareColP()

    if verbose > 0 then
        disp("Points set.")
        disp("Saving reusable b-spline evaluations.")
    end

    // Save basis values at mesh and col points
    setSavedBasisVals()
    saveBoundaryEvals()

    if verbose > 0 then
        disp("Saved.")
    end

    // Generate a random first guess
    fg = zeros(ncpts,1)

    // Send collocation to fsolve and save in global coeffs
    global coeffs
    coeffs(ncpts) = 0
    colInds = saveColInds()
    xBndInds = savexBndInds()
    yBndInds = saveyBndInds()
    cornInds = saveCornInds()

    if verbose > 0 then
        disp("Solving for coefficients.")
        if verbose > 1 then
            tic
        end
    end

    if jacMode == 0 then
        [coeffs, b, info] = fsolve(fg, colConds, atol*0.1)
    elseif jacMode == 1 then
        [coeffs, b, info] = fsolve(fg, colConds, fjac, atol*0.1)
    end

    if verbose > 0 then
        disp("Coefficients set.")
        if verbose > 1 then
            time = toc()
            disp("Elapsed time: " + string(time) + " seconds")
        end
        mprintf("fsolve info: %d\n", info)
    end
endfunction

// Calculates the residuals for given coefs for all of the collocation
// conditions.
function c = colConds(coefs)
    // Set coefs to global coeffs
    global coeffs
    coeffs = coefs
    c(ncpts) = 0

    // Set x = A first
    c(1) = savedU(1, 1, 1, 1) - bndA(C)
    // Loop through the collocation points on x = A
    for i = 1:M
        for j = 1:(q-1)
            c(i*q-i-q+j+2) = savedU(1, i*q-i-q+j+2, 1, 1) - bndAvals(i, j)
        end
    end
    c(ncptY) = savedU(1, ncptY, 1, 1) - bndA(D)


    // Loop through the collocation points in X
    for i = 1:N
        for j = 1:(p-1)

            // Set the bottom boundary condition
            c(ncptY*(i*p-i-p+j+1)+1) = ...
            savedU(i*p-i-p+j+2, 1, 1, 1) - bndCvals(i, j)

            // Loop up through the collocation points in Y
            for ii = 1:M
                for jj = 1:(q-1)
                    c(ncptY*(i*p-i-p+j+1)+ii*q-ii-q+jj+2) =...
                    savedPDE(i, j, ii, jj)
                end
            end
            // Set the top boundary condition
            c(ncptY*(i*p-i-p+j+2)) = savedU(i*p-i-p+j+2, ncptY, 1, 1) - bndDvals(i, j
                )
        end
    end

    // Set x = B
    c((ncptX-1)*ncptY+1) = savedU(ncptX, 1, 1, 1) - bndB(C)

    // Loop through the collocation points on x = B
    for i = 1:M
        for j = 1:(q-1)
            c((ncptX-1)*ncptY+i*q-i-q+j+2) = savedU(ncptX, i*q-i-q+j+2, 1, 1) -
                bndBvals(i, j)
        end
    end
    c(ncpts) = savedU(ncptX, ncptY, 1, 1) - bndB(D)
endfunction

function r = fjac(coefs)
    global coeffs t0 dy
    inds = nonZeroInds(1)
    coeffs = coefs

    for k = 1:size(inds)(1)
            ind = inds(k)
            coeffs(1) = coefs(1)+dy
            rp1 = condI(ind)
            coeffs(1) = coefs(1)-dy
            r(ind,1) = (0.5*rp1 - 0.5*condI(ind))/dy
    end
    for i = 2:ncpts
        inds = nonZeroInds(i)
        coeffs(i-1) = coefs(i-1)
        for k = 1:size(inds)(1)
            ind = inds(k)
            coeffs(i) = coefs(i)+dy
            rp1 = condI(ind)
            coeffs(i) = coefs(i)-dy
            r(ind,i) = (0.5*rp1 - 0.5*condI(ind))/dy
        end
    end
endfunction

function c = condI(ind)

    [line, rem] = getLine(ind)
    if line == 1 then // x = A
        c = savedU(line, rem)
    elseif line == ncptX then // x = B
        c = savedU(line, rem)
    elseif rem == 1 then // y = C
        c = savedU(line, rem)
    elseif rem == ncptY then // y = D
        c = savedU(line, rem)
    else
        cx = modulo(line-1,p-1)
        cy = modulo(rem-1,q-1)
        if cx == 0 then
        cx = p-1
        end
        if cy == 0 then
        cy = q-1
        end
        c = savedPDE(int((line-2)/(p-1))+1, cx,int((rem-2)/(q-1))+1, cy)
    end
endfunction
```

core2DPPDE.sci

```
global calcJacMode
calcJacMode = %F;
global coefTimes // Stores the times of the coefficients in allCoeffs
global t0 // The starting time given to DASKR
global tc
global allCoeffs


// Calculates a collocation solution of degree p with N intervals
// in x, and degree q with M subintervals in y. Stores the calculated
// coefficients in global coeffs
function collocate(deg, nints, ti, tOut)
    // Set global variables
    global p N q M ncpts AC AD BC BD ncptX ncptY t0 tc coeffs
    global colInds xBndInds yBndInds cornInds initY initYdot
```

```
p = deg
N = nints
q = deg
M = nints
ncptY = (M * (q-1) + 2)
ncptX = (N * (p-1) + 2)
ncpts =  ncptX * ncptY
if verbose > 0 then
    disp("p=" + string(p) + " N=" + string(N) + " q=" + string(q) + ..
" M=" + string(M))
end
setBounds()

if verbose > 0 then
    disp("Setting mesh, knot, and collocation points.")
end
// Set the mesh
prepareMesh()

// Set the knot sequence
prepareKnots()

// Set the collocation points
prepareColP()
if verbose > 0 then
    disp("Points set.")
    disp("Saving reusable b-spline evaluations.")
end

// Save basis values at mesh and col points
setSavedBasisVals()

if verbose > 0 then
    disp("Saved.")
end

t0 = ti
colInds = saveColInds()
xBndInds = savexBndInds()
yBndInds = saveyBndInds()
cornInds = saveCornInds()
if verbose > 0 then
    disp("Calculating initial coefficients.")
    if verbose > 1 then
        tic
    end
end
initCoeffs()
if verbose > 0 then
    disp("Coefficients set.")
    if verbose > 1 then
        time = toc()
        disp("Elapsed time: " + string(time) +" seconds")
        tic
    end
    disp("Calculating initial derivative of coefficients.")
end
coDer = initCoeffsDer()
if verbose > 0 then
    disp("Derivative of coefficients set.")
    if verbose > 1 then
        time = toc()
        disp("Elapsed time: " + string(time) + " seconds")
        tic
    end
    disp("Beginning time integration.")
end

// INFO list which controls running of DASKR
info = list([], 0, [], [], [], 0, -ones(1,ncpts), 0, 0, 0, 0, [], [], 0)

// Define the surfaces, which are required to call but not needed.
ng = 0
deff('[rts] = gr1(t, y)', 'rts = [1]')

// Call DASSL
if jacMode == 0 then
    [r, nn] = daskr([coeffs, coDer], t0, tOut, atol*0.1, res,ng, gr1, info)
elseif jacMode == 1 then
    [r, nn] = daskr([coeffs, coDer], t0, tOut, atol*0.1, res,qjacRes,ng, gr1,
        info)
end
```

```
    if verbose > 0 then
        disp("Time integration complete.")
        if verbose > 1 then
            time = toc()
            disp("Elapsed time: " + string(time) + " seconds")
        end
    end

    // Save results from DASKR
    global coeffs coefTimes allCoeffs
    coeffs = r(2:ncpts+1, size(r)(2))
    tc = r(1, size(r)(2))
endfunction


// Efficient approximation of Jacobian requested by DASKR.
function r = qjacRes(t, y, ydot, cj)
    inds = nonZeroInds(1)
    mockY = y
    mockYdot = ydot

    for k = 1:size(inds)(1)
        ind = inds(k)
        mockY(1) = y(1)+dy
        mockYdot(1) = ydot(1)+dy
        rp1 = resI(t, mockY, ydot, ind)
        rdp1 = resI(t, y, mockYdot, ind)
        mockY(1) = y(1)-dy
        mockYdot(1) = ydot(1)-dy
        r(ind,1) = (0.5*rp1 - 0.5*resI(t,mockY,ydot, ind))/dy
        r(ind,1) = r(ind,1) + cj*(0.5*rdp1 - 0.5*resI(t, y, mockYdot, ind))/dy
    end
    for i = 2:ncpts
        inds = nonZeroInds(i)
        mockY(i-1) = y(i-1)
        mockYdot(i-1) = ydot(i-1)
        for k = 1:size(inds)(1)
            ind = inds(k)
            mockY(i) = y(i)+dy
            mockYdot(i) = ydot(i)+dy
            rp1 = resI(t, mockY, ydot, ind)
            rdp1 = resI(t, y, mockYdot, ind)
            mockY(i) = y(i)-dy
            mockYdot(i) = ydot(i)-dy
            r(ind,i) = (0.5*rp1 - 0.5*resI(t,mockY,ydot, ind))/dy
            r(ind,i) = r(ind,i) + cj*(0.5*rdp1 - 0.5*resI(t, y, mockYdot, ind))/dy
        end
    end
endfunction

// Approximates the Jacobian used for setting the initial spatial derivative of the
    coefficients
function r = fjacD(ydot)
    global coeffs t0 dy
    inds = nonZeroInds(1)
    y = coeffs
    t = t0
    mockYdot = ydot

    for k = 1:size(inds)(1)
            ind = inds(k)
            mockYdot(1) = ydot(1)+dy
            rdp1 = resI(t, y, mockYdot, ind)
            mockYdot(1) = ydot(1)-dy
            r(ind,1) = (0.5*rdp1 - 0.5*resI(t, y, mockYdot, ind))/dy
    end
    for i = 2:ncpts
        inds = nonZeroInds(i)
        mockYdot(i-1) = ydot(i-1)
        for k = 1:size(inds)(1)
            ind = inds(k)
            mockYdot(i) = ydot(i)+dy
            rdp1 = resI(t, y, mockYdot, ind)
            mockYdot(i) = ydot(i)-dy
            r(ind,i) = (0.5*rdp1 - 0.5*resI(t, y, mockYdot, ind))/dy
        end
    end
endfunction

// Approximates the Jacobian used for setting the initial coefficients
function r = fjac(y)
```

```
    global coeffs t0 dy
    inds = nonZeroInds(1)
    coeffs = y

    for k = 1:size(inds)(1)
            ind = inds(k)
            coeffs(1) = y(1)+dy
            [xind,yind] = toCoord(ind)
            rp1 = savedU(xind,yind)
            coeffs(1) = y(1)-dy
            r(ind,1) = (0.5*rp1 - 0.5*savedU(xind,yind))/dy
    end
    for i = 2:ncpts
        inds = nonZeroInds(i)
        coeffs(i-1) = y(i-1)
        for k = 1:size(inds)(1)
            ind = inds(k)
            coeffs(i) = y(i)+dy
            [xind,yind] = toCoord(ind)
            rp1 = savedU(xind,yind)
            coeffs(i) = y(i)-dy
            r(ind,i) = (0.5*rp1 - 0.5*savedU(xind,yind))/dy
        end
    end
endfunction

// Efficient version of F which only makes the neccesarry evaluations.
// t is the time for which the PDE is being evaluated at.
// x/y is the index (not value) of the point to evaluate at.
//     x = 1 -> x = A, x = 2 -> x = colX(1, 1), ..., x = ncptX -> x = B
function r = resF(t, x, y)
    used = usedFEvals()
    evals = zeros(5, 1)
    if used(1) then
        evals(1) = savedU(x, y, 1, 1)
    end
    if used(2) then
        evals(2) = savedU(x, y, 2, 1)
    end
    if used(3) then
        evals(3) = savedU(x, y, 1, 2)
    end
    if used(4) then
        evals(4) = savedU(x, y, 3, 1)
    end
    if used(5) then
        evals(5) = savedU(x, y, 1, 3)
    end
    x = getPtFromInd(x)
    y = = getPtFromInd(y, %T)
    r = f(t, x, y, evals(1), evals(2), evals(3), evals(4), evals(5))
endfunction

// Efficient version of bnd. condition
// t is the time to evaluate the boundary condition at
// y is the index of the point in y to evaluate at.
function r = resBndxa(y, t)
    used = usedBndEvals('E')
    evals = zeros(3, 1)
    if used(1) then
        evals(1) = savedU(1, y, 1, 1)
    end
    if used(2) then
        evals(2) = savedU(1, y, 2, 1)
    end
    if used(3) then
        evals(3) = savedU(1, y, 1, 2)
    end
    pt = getPtFromInd(y, %T)
    r = bndxa(pt, t, evals(1), evals(2), evals(3))
endfunction

// Efficient version of bnd. condition
// t is the time to evaluate the boundary condition at
// y is the index of the point in y to evaluate at.
function r = resBndxb(y, t)
    used = usedBndEvals('E')
    evals = zeros(3, 1)
    if used(1) then
        evals(1) = savedU(ncptX, y, 1, 1)
    end
    if used(2) then
        evals(2) = savedU(ncptX, y, 2, 1)
    end
    if used(3) then
        evals(3) = savedU(ncptX, y, 1, 2)
    end
    pt = getPtFromInd(y, %T)
    r = bndxb(pt, t, evals(1), evals(2), evals(3))
endfunction

// Efficient version of bnd. condition
// t is the time to evaluate the boundary condition at
// y is the index of the point in y to evaluate at.
function r = resBndyc(x, t)
    used = usedBndEvals('E')
    evals = zeros(3, 1)
    if used(1) then
        evals(1) = savedU(x, 1, 1, 1)
    end
    if used(2) then
        evals(2) = savedU(x, 1, 2, 1)
    end
    if used(3) then
        evals(3) = savedU(x, 1, 1, 2)
    end
    pt = getPtFromInd(x)
    r = bndyc(pt, t, evals(1), evals(2), evals(3))
endfunction

// Efficient version of bnd. condition
// t is the time to evaluate the boundary condition at
// y is the index of the point in y to evaluate at.
function r = resBndyd(x, t)
    used = usedBndEvals('E')
    evals = zeros(3, 1)
    if used(1) then
        evals(1) = savedU(x, ncptY, 1, 1)
    end
    if used(2) then
        evals(2) = savedU(x, ncptY, 2, 1)
    end
    if used(3) then
        evals(3) = savedU(x, ncptY, 1, 2)
    end
    pt = getPtFromInd(x)
    r = bndyd(pt, t, evals(1), evals(2), evals(3))
endfunction

// Returns the value of a point from its index.
// ind is the index of the point.
// isY is a boolean to specify if it is a point in x(T) or Y(F)
function pt = getPtFromInd(ind, isY)
    if argn(2) < 2 then
        isY = %F
    end
    select isY
    case %F then
        if ind == 1 then
            pt = A
        elseif ind == ncptX then
            pt = B
        else
            [inter, colp] = getColPt(ind, isY)
            pt = colX(inter, colp)
        end
    case %T then
        if ind == 1 then
            pt = C
        elseif ind == ncptX then
            pt = D
        else
            [inter, colp] = getColPt(ind, isY)
            pt = colX(inter, colp)
        end
    end
endfunction

// Sets global coeffs to an interpolation of the initial conditions
// using fsolve.
function initCoeffs()
```

```
    iniCoeffs()
    global coeffs
    coeffs = ones(ncpts, 1)
    info = 4
    if jacMode == 0 then
        [coeffs, n, info] = fsolve(coeffs, fsol, 1.d-14)
    elseif jacMode == 1 then
        [coeffs, n, info] = fsolve(coeffs, fsol, fjac, 1.d-14)
    end
    if verbose > 1 then
        disp("fsolve info: " + string(info))
        disp("fsolve residual: " + string(max(abs(fsol(coeffs)))))
    end
endfunction

// Residual function for setting the initial condition so that they are
// interpolated. Parameter coefs gets set as the global coeffs and
// residual is calculated by the difference from the value of the
// collocation solution to the inital condition.
function c = fsol(coefs)
    // Set coefs to global coeffs
    global coeffs
    coeffs = coefs

    // Set x = A first
    c(ncpts) = 0
    c(1) = uinit(A, C) - savedU(1, 1)
    // Loop through the collocation points on x = A
    for i = 1:M
        for j = 1:(q-1)
            c(i*q-i-q+j+2) = uinit(A, colY(i,j)) - savedU(1, i*q-i-q+j+2)
        end
    end
    c(ncptY) = uinit(A, D) - savedU(1, ncptY)

    // Loop through the collocation points in X
    for i = 1:N
        for j = 1:(p-1)

            // Save the reused collocation point
            colP = colX(i, j)

            // Set the bottom boundary condition
            xInd = (i-1) * (p-1) + j + 1
            c(ncptY*(i*p-i-p+j+1)+1) = uinit(colP, C) - savedU(xInd, 1)

            // Loop up through the collocation points in Y
            for ii = 1:M
                for jj = 1:(q-1)
                    c(ncptY*(i*p-i-p+j+1)+ii*q-ii-q+jj+2) =...
                    uinit(colP, colY(ii, jj)) - savedU(xInd, (ii - 1) * (q-1) + jj +
                        1)
                end
            end
            // Set the top boundary condition
            c(ncptY*(i*p-i-p+j+2)) = uinit(colP, D) - savedU(xInd, ncptY)
        end
    end

    // Set x = B
    c((ncptX-1)*ncptY+1) = uinit(B, C) - savedU(ncptX, 1)

    // Loop through the collocation points on x = B
    for i = 1:M
        for j = 1:(q-1)
            c((ncptX-1)*ncptY+i*q-i-q+j+2) =...
            uinit(B, colY(i, j)) - savedU(ncptX, i*q-i-q+j+2)
        end
    end
    c(ncpts) = uinit(B, D) - savedU(ncptX, ncptY)
endfunction

// Creates a first guess of the coefficients for fsolve before solving for the
// initial conditions.
function iniCoeffs()
    //ind = 1

    // Set coefs to global coeffs
    global coeffs

    // Set x = A first
    coeffs(ncpts) = 0
    coeffs(1) = uinit(A, C)

    // Loop through the collocation points on x = A
    for i = 1:M
        for j = 1:(q-1)
            coeffs(i*q-i-q+j+2) = uinit(A, colY(i, j))
        end
    end
    coeffs(ncptY) = uinit(A, D)

    // Loop through the collocation points in X
    for i = 1:N
        for j = 1:(p-1)

            // Save the reused collocation point
            colP = colX(i, j)

            // Set the bottom boundary condition
            coeffs(ncptY*(i*p-i-p+j+1)+1) = uinit(colP, C)

            // Loop up through the collocation points in Y
            for ii = 1:M
                for jj = 1:(q-1)
                    coeffs(ncptY*(i*p-i-p+j+1)+ii*q-ii-q+jj+2) =...
                    uinit(colP, colY(ii, jj))
                end
            end

            // Set the top boundary condition
            coeffs(ncptY*(i*p-i-p+j+2)) = uinit(colP, D)
        end
    end

    // Set x = B
    coeffs((ncptX-1)*ncptY+1) = uinit(B, C)

    // Loop through the collocation points on x = B
    for i = 1:M
        for j = 1:(q-1)
            coeffs((ncptX-1)*ncptY+i*q-i-q+j+2) = uinit(B, colY(i, j))
        end
    end
    coeffs(ncpts) = uinit(B, D)
endfunction

// Residual function used for setting the derivative of coefficients initial
// guess given to DASKR.
function r = fsolD(coDer)
    [r, ires] = res(t0, coeffs, coDer)
endfunction

// Returns the initial guess of derivative of the coefficients w.r.t time
// Uses fsolve to minimize the residual at the inital time with the
// initial coefficients.
function y0p = initCoeffsDer()
    fg = zeros(ncpts, 1)
    if jacMode == 0 then
        [y0p, n, inf] = fsolve(fg, fsolD, 1.d-14)
    elseif jacMode == 1 then
        [y0p, n, inf] = fsolve(fg, fsolD, fjacD, 1.d-14)
    end
    if verbose > 1 then
        disp("fsolve info: " + string(inf))
        disp("fsolve max residual: " + string(max(abs(fsolD(y0p)))))
    end
endfunction

// Residual function which is used by DASKR.
// t is the current time to calculate the residual at
// u is the coefficients passed in by DASKR
// up is the derivative w.r.t of the coefficients
function [c, ires] = res(t, u, up)
    global coeffs

    ptDer = getPtDer(up)
    coeffs = u

    // Set coefs to global coeffs
    global coeffs

    // Loop through the boundary conditions on x = A
    for i = 1:ncptY
        c(i) = resBndxa(i, t)
    end

    // Loop through the vertical lines
    for i = 2:(ncptX - 1)
        c((i-1)*ncptY+1) = resBndyc(i, t)
        for j = 2:(ncptY - 1)
            c((i-1)*ncptY+j) = resF(t, i, j) - ptDer((i-1)*ncptY+j)
        end
        c(i*ncptY) = resBndyd(i, t)
    end

    // Loop through boundary conditions on x = B
    for i = 1:ncptY
        c((ncptX-1)*ncptY+i) = resBndxb(i, t)
    end
    ires = 0
endfunction

// Returns the indth residual as would appear from the res() function
// given input t, u, up
function c = resI(t, u, up, ind)
    global coeffs
    coeffs = u
    [line, rem] = getLine(ind)
    if line == 1 then // x = A
        c = resBndxa(rem, t)
    elseif line == ncptX then // x = B
        c = resBndxb(rem, t)
    elseif rem == 1 then // y = C
        c = resBndyc(line, t)
    elseif rem == ncptY then // y = D
        c = resBndyd(line, t)
    else
        coeffs = up
        ptDer = savedU(line, rem, 1, 1)
        coeffs = u
        c = resF(t, line, rem) - ptDer
    end
endfunction

// Returns the derivative w.r.t of the condition points from a given
// derivative w.r.t of the b-spline coefficients.
function ptDer = getPtDer(coDer)
    global coeffs
    temp = coeffs
    coeffs = coDer

    ptDer(ncpts) = 0
    for i = 1:ncptY
        ptDer(i) = savedU(1, i, 1, 1)
    end

    for xInd = 2:ncptX - 1
        ptDer((xInd-1)*ncptY+1) = savedU(xInd, 1, 1, 1)
        for yInd = 2:ncptY - 1
            ptDer((xInd-1)*ncptY+yInd) = savedU(xInd, yInd, 1, 1)
        end
        ptDer(xInd*ncptY) = savedU(xInd, ncptY, 1, 1)
    end

    for i = 1:ncptY
        ptDer((ncptX-1)*ncptY+i) = savedU(ncptX, i, 1, 1)
    end

    coeffs = temp
endfunction

// Evaluates the known solution at the time of the collocation solution.
function z = truu(x, y)
    z = correct(x, y, tc)
endfunction
```

probsBVODE.sci

```
// Specifies the bounds on x for solving the DE
function setBounds()
    global A B
    select probNum
    case 1 then
```

```
        A = 0
        B = 2
    case 2 then
        A = 0
        B = 2
    end
endfunction

function y = actual(x) // The actual answer
    select probNum
    case 1 then
        y = (1/6)*(x^3)*(%e^x)-(5/3)*x*(%e^x)+2*(%e^x)-x-2
    case 2 then
        y = x * (%e^x - %e)
    end
endfunction

function y = truu(x)
    y = actual(x)
endfunction


// The right hand side of PDE goes here
function y = PDE(coeff, x)
    select probNum
    case 1 then
        y = Yxx(coeff, x) - 2 * Yx(coeff, x) + Y(coeff, x) - x*(%e^x) + x
    case 2 then
        y = Yxx(coeff, x) - %e^x * (x+2)
    end
endfunction
```

probsPPDE.sci

```
// Problem parameters
global eps
eps = 1/16;

function setBnds()
    global A B
    select probNum
    case 1 then
        A = 0
        B = 1
    case 2 then
        A = 0
        B = 1
    end
endfunction

function r = bndxa(t, u, u_x)
    select probNum
    case 1 then
        r = u - 0.5 + 0.5 * tanh((-0.5*t-0.25)/(4*eps))
    case 2 then
        r = u
    end
endfunction

function r = bndxb(t, u, u_x)
    select probNum
    case 1 then
        r = 0.5 * tanh((0.75-0.5*t)/(4*eps))-0.5 + u
    case 2 then
        r = u
    end
endfunction

function y = uinit(x)
    select probNum
    case 1 then
        y = 0.5 - 0.5 * tanh((x-0.25)/(4*eps))
    case 2 then
        y = 2 * sin(2 * %pi * x)
    end
endfunction

function y = f(t, x, u, u_x, u_xx)
    select probNum
    case 1 then
        y = eps * u_xx - u * u_x
    case 2 then
        y = eps * u_xx
    end
endfunction

function y = correct(x, t)
    select probNum
    case 1 then
        y = 0.5 - 0.5 * tanh( (x-0.5*t-0.25) / (4*eps) )
    case 2 then
        y = 2 * sin(2* %pi * x) * %e^-(t*(%pi^2)/4)
    end
endfunction

function y = trux(x)
    select probNum
    case 1 then
        y = -1.25 * sech(1.25*tc - 2.5*x + 0.625)^2
    case 2 then
        y = 4 * %pi * cos(2 * %pi * x) * %e^-(tc*(%pi^2)/4)
    end
endfunction

function y = truu(x)
    y = correct(x, tc)
endfunction
```

probsEPDE.sci

```
// The PDE to be satisfied at collocation points
// Use this function to evaluate at any point (x, y)
function p = f(x, y)
    select probNum
    case 1 then
        p = U(x,y,3,1)+U(x,y,1,3) - ..
        x * %e^y
    case 2 then
        p = U(x,y,3,1)+U(x,y,1,3) - ..
        (x^2 + y^2) * %e^(x*y)
    end
endfunction

// Evaluates the PDE at a collocation point. xi and yi are the interval
// in x and y respectovely, while xj and yj are the index of the collocation
// point within that interval.
function p = savedPDE(xi, xj, yi, yj)
    xInd = (xi-1)*(p-1)+xj+1
    yInd = (yi-1)*(q-1)+yj+1
    x = colX(xi, xj)
    y = colY(yi, yj)
    select probNum
    case 1 then
        p = savedU(xInd,yInd,3,1)+savedU(xInd,yInd,1,3) - ..
        x * %e^y
    case 2 then
        p = savedU(xInd,yInd,3,1)+savedU(xInd,yInd,1,3) - ..
        (x^2 + y^2) * %e^(x*y)
    end
endfunction

// Boundary condition along x = A
function z = bndA(y)
    select probNum
    case 1 then
        z = 0
    case 2 then
        z = 1
    end
endfunction

// Boundary condition along x = B
function z = bndB(y)
    select probNum
    case 1 then
        z = 2 * %e^y
    case 2 then
        z = %e^(2 * y)
    end
endfunction
```

```
// Boundary condition along y = C
function z = bndC(x)
    select probNum
    case 1 then
        z = x
    case 2 then
        z = 1
    end
endfunction

// Boundary condition along y = D
function z = bndD(x)
    select probNum
    case 1 then
        z = %e * x
    case 2 then
        z = %e^x
    end
endfunction

// Specifies the bounds on x and y for solving the PDE
function setBounds()
    global A B C D
    select probNum
    case 1 then
        A = 0
        B = 2
        C = 0
        D = 1
    case 2 then
        A = 0
        B = 2
        C = 0
        D = 1
    end
endfunction

// End functions defining the PDE being solved
// ------------------------------------------------------------------------
// Information about the actual solution to the PDE, used for testing

// The true solution
function z = truu(x, y)
    select probNum
    case 1 then
        z = x * %e^y
    case 2 then
        z = %e^(x * y)
    end
endfunction


// End actual solution information
```

probs2DPPDE.sci

```
global eps // Problem parameter
eps = 0.1;

// The PDE which is to be satsisfied at the collocation points
function r = f(t, x, y, u, u_x, u_y, u_xx, u_yy)
    select probNum
    case 1 then
        // Heat diffusion
        r = eps * (u_xx + u_yy)
    case 2 then
        // Burgers equation
        r = eps * (u_xx + u_yy) - u * (u_x + u_y)
    end
endfunction

// The PDE which is used when calculating the jacobian
function p = jacPDE(xi, yi)
    select probNum
    case 1 then
        // Heat diffusion
        p = savedU(xi, yi, 3, 1) + savedU(xi, yi, 1, 3)
    case 2 then
        // Burgers equation
        p = savedU(xi, yi, 3, 1) + savedU(xi, yi, 1, 3) - savedU(xi,yi,1,1)*(savedU(
            xi, yi, 2, 1) + savedU(xi, yi, 1, 2))
    end
endfunction

// Gives the initial conditions of a problem
function z = uinit(x, y)
    select probNum
    case 1 then
       // z = sin(x*%pi/2) * sin(y*%pi/2)
        z = correct(x, y, t0)
    case 2 then
        z = 1 / (1 + %e^((x+y)/(2 * eps)))
    end
endfunction

// Specifies which evaluations must be made to evaluate the boundary
// conditions. The indices of used correspond to U, Ux, and Uy.
function used = usedBndEvals(letter)
    select probNum
    case 1 then
        used = [%T, %F, %F]
    case 2 then
        used = [%T, %F, %F]
    end
endfunction

// Specifies which evaluations must be made to evaluate the PDE.
// The indices of used correspond to U, Ux, Uy, Uxx, Uyy
function used = usedFEvals()
    select probNum
    case 1 then
        used = [%F, %F, %F, %T, %T]
    case 2 then
        used = [%T, %T, %T, %T, %T]
    end
endfunction

// Boundary condition along x = A
function r = bndxa(y, t, u, u_x, u_y)
    select probNum
    case 1 then
        r = u
    case 2 then
        r = u - correct(A, y, t)
    end
endfunction

// Boundary condition along x = B
function r = bndxb(y, t, u, u_x, u_y)
    select probNum
    case 1 then
        r = u
    case 2 then
        r = u - correct(B, y, t)
    end
endfunction

// Boundary condition along y = C
function r = bndyc(x, t, u, u_x, u_y)
    select probNum
    case 1 then
        r = u
    case 2 then
        r = u - correct(x, C, t)
    end
endfunction

// Boundary condition along y = D
function r = bndyd(x, y, u, u_x, u_y)
    select probNum
    case 1 then
        r = u
    case 2 then
        r = u - correct(x, D, t)
    end
endfunction

// Specifies the bounds on x and y for solving the PDE
function setBounds()
    global A B C D
    select probNum
    case 1 then
        A = 0
        B = 2
        C = 0
        D = 2
    case 2 then
        A = 0
        B = 1
        C = 0
        D = 1
    end
endfunction

// The known true solution to the PDE
function z = correct(x, y, t)
    if calcJacMode then
        z = 0
        return
    end
    select probNum
    case 1 then
        z = sin((%pi/2) * x) * sin((%pi/2) * y) * %e^-(t*eps*((%pi/(B-A))^2 + (%pi/(D
          -C))^2))
    case 2 then
        z = 1 / (1 + %e^((x+y-t)/(2 * eps)))
    end
endfunction
```

err.sci

```
// Weight function for HBI
function y = gammaHBI(j, s, w)
    y = 0
    for i = 1:max(size(w))
        y = y + (1/(s(j) - w(i)))
    end

    for i = 1:max(size(s))
        if i ~= j then
            y = y + 2 * (1/(s(j) - s(i)))
        end
    end
endfunction

// Weight function for HBI
function y = eta(x, j, s)
    y = 1
    for r = 1:max(size(s))
        if r ~= j then
            y = y * (x - s(r))
        end
    end
endfunction

// Weight function for HBI
function y = eta2(x, j, s)
    y = eta(x, j, s)^2
endfunction

// Weight function for HBI
function y = phi(x, j, w)
    y = 1
    for r = 1:max(size(w))
        if r ~= j then
            y = y * (x - w(r))
        end
    end
endfunction

// Weight function for HBI
function y = G(x, j, s, w)
    y = (phi(x, j, w) * eta2(x, -1, s))/(phi(w(j), j, w) * eta2(w(j), -1, s))
endfunction

// Weight function for HBI
function y = Hsup(x, j, s, w)
    y = (eta2(x, j, s) * phi(x, -1, w))/(eta2(s(j), j, s) * phi(s(j), -1, w))
endfunction

// Weight function for HBI
function y = Hbar(x, j, s, w)
    y = (x - s(j)) * Hsup(x, j, s, w)
endfunction
```

```
// Weight function for HBI
function y = H(x, j, s, w)
    y = (1 - (x - s(j)) * gammaHBI(j, s, w)) *  Hsup(x, j, s, w)
endfunction

// Returns Gaussian quadrature weights for k points
function wts = getQuadWts(k)
    select k
    case 5 then
        wts(1) = 0.2369268850561891
        wts(2) = 0.4786286704993665
        wts(3) = 0.5688888888888889
        wts(4) = 0.4786286704993665
        wts(5) = 0.2369268850561891
    case 6 then
        wts(1) = 0.1713244923791704
        wts(2) = 0.3607615730481386
        wts(3) = 0.4679139345726910
        wts(4) = 0.4679139345726910
        wts(5) = 0.3607615730481386
        wts(6) = 0.1713244923791704
    case 7 then
        wts(1) = 0.1294849661688697
        wts(2) = 0.2797053914892766
        wts(3) = 0.3818300505051189
        wts(4) = 0.4179591836734694
        wts(5) = 0.3818300505051189
        wts(6) = 0.2797053914892766
        wts(7) = 0.1294849661688697
    case 8 then
        wts(1) = 0.1012285362903763
        wts(2) = 0.2223810344533745
        wts(3) = 0.3137066458778873
        wts(4) = 0.3626837833783620
        wts(5) = 0.3626837833783620
        wts(6) = 0.3137066458778873
        wts(7) = 0.2223810344533745
        wts(8) = 0.1012285362903763
    case 9 then
        wts(1) = 0.0812743883615744
        wts(2) = 0.1806481606948574
        wts(3) = 0.2606106964029354
        wts(4) = 0.3123470770400029
        wts(5) = 0.3302393550012598
        wts(6) = 0.3123470770400029
        wts(7) = 0.2606106964029354
        wts(8) = 0.1806481606948574
        wts(9) = 0.0812743883615744
    end
endfunction

// Returns numQp gaussian quadrature points and their weights
function [pts, wts] = getQuadPtsWts(meshL, meshH, numQp)
    pts = getGaussPts(meshL, meshH, numQp)
    wts = getQuadWts(numQp)
endfunction

// Function which returns the appropriate superconvergent points to use for
// the SCI on interval ind, on meshP having nint intervals, and for a
// collocation solution of degree deg.
function w = getsciW(ind, meshP, deg, nint)

    // Get the superconvergent points internal to the subinterval
    internal = scNonMeshP(meshP(ind:ind+1), deg, 1)

    // External points depend on location of the interval
    if ind == 1 then // Need to take both external from the next interval
        if deg == 4 then
            // Only one scp so need to use next mesh point
            ext(1) = scNonMeshP(meshP(2:3), 4, 1)
            ext(2) = meshP(3)
        else
            // Take the first two scp
            ext = scNonMeshP(meshP(2:3), deg, 1)(1:2)
        end
    elseif ind == nint then // Need to take both from last interval
        if deg == 4 then
            // Need to take meshpoint also
            ext(1) = scNonMeshP(meshP(nint-1:nint), 4, 1)
```

```
                ext(2) = meshP(nint-1)
            else
                // Take the last two scp
                ext = scNonMeshP(meshP(nint-1:nint), deg, 1)(deg-4:deg-3)
            end
        else
            // Take last scp from last interval and first from next interval
            ext(1) = scNonMeshP(meshP(ind-1:ind),deg,1)(deg-3)
            ext(2) = scNonMeshP(meshP(ind+1:ind+2),deg,1)(1)
        end

        // Combine the internal and external points
        w = gsort([internal;ext], 'g', 'i')
endfunction

// Calculates and returns all the non-mesh super convergent points for
// meshP and deg. The values to calculate these points are taken from
// the BACOLI code
function p = scNonMeshP(meshP, deg, nint)
    // Index for populating the points
    ind = 1

    // Loop over the subintervals
    for i = 1:nint
        // Calculate h for this subinterval
        h = meshP(i+1) - meshP(i)

        // Split on deg, which determines the number of scNonMeshP
        if deg == 4 then // 1 SC point
            p(ind) = meshP(i) + 0.5 * h
            ind = ind+1
        elseif deg == 5 then // 2 sc points ------------------------------------
            p(ind) = meshP(i) + 0.3110177634953864 * h
            ind = ind+1
            p(ind) = meshP(i) + 0.6889822365046136 * h
            ind = ind+1
        elseif deg == 6 then // 3 sc points -----------------------------------
            p(ind) = meshP(i) + 0.2113248654051871 * h
            ind = ind+1
            p(ind) = meshP(i) + 0.5 * h
            ind = ind+1
            p(ind) = meshP(i) + 0.7886751345948129 * h
            ind = ind+1
        elseif deg == 7 then // 4 sc points ------------------------------------
            p(ind) = meshP(i) + 0.1526267046965671 * h
            ind = ind+1
            p(ind) = meshP(i) + 0.3747185964571342 * h
            ind = ind+1
            p(ind) = meshP(i) + 0.6252814035428658 * h
            ind = ind+1
            p(ind) = meshP(i) + 0.8473732953034329 * h
            ind = ind+1
        elseif deg == 8 then // 5 sc points ------------------------------------
            p(ind) = meshP(i) + 0.1152723378341063 * h
            ind = ind+1
            p(ind) = meshP(i) + 0.2895425974880943 * h
            ind = ind+1
            p(ind) = meshP(i) + 0.5 * h
            ind = ind+1
            p(ind) = meshP(i) + 0.7104574025119057 * h
            ind = ind+1
            p(ind) = meshP(i) + 0.8847276621658937 * h
            ind = ind+1
        else
            p(ind) = -1;
        end
    end
endfunction

// Returns the points between pL and pH to be interpolated by the L2 for
// a collocation solution of degree deg.
function pt = getL2pts(pL, pH, deg)
    diam = pH - pL
        select deg
            case 4 then
                pt(1) = pL
                pt(2) = pL + diam * 0.302331973224813
                pt(3) = pL + diam * 0.697668026790731
                pt(4) = pH
            case 5 then
                pt(1) = pL
                pt(2) = pL + diam * 0.179424182143275
                pt(3) = pL + diam * 0.5
                pt(4) = pL + diam * 0.820575567392312
                pt(5) = pH
            case 6 then
                pt(1) = pL
                pt(2) = pL + diam * 0.143465814421734
                pt(3) = pL + diam * 0.37051884018424
                pt(4) = pL + diam * 0.629481160240031
                pt(5) = pL + diam * 0.856534185886017
                pt(6) = pH
            case 7 then
                pt(1) = pL
                pt(2) = pL + diam * 0.107235231524833
                pt(3) = pL + diam * 0.283676545203967
                pt(4) = pL + diam * 0.5
                pt(5) = pL + diam * 0.716323434111384
                pt(6) = pL + diam * 0.892764777407028
                pt(7) = pH
        end
endfunction
```

err1D.sci

```
function val = evalGlobalErr(f, numQuadpts, hScale)

    // Get the evaluation points and their weights
    [pts, wts] = getQuadPtsWts(meshX(1), meshX(N+1), numQuadpts)

    // Calculate the sum
    val = 0
    for i = 1:numQuadpts
        temp = U(pts(i))
        val = val + wts(i) * ((temp - f(pts(i)))/(atol+rtol*abs(temp)))^2
    end

    // Scale the quadrature
    val = val * (B-A)/2
    val = sqrt(val)

    // Scale by max h
    if hScale then
        // Find the maximal h
        maxH = 0
        for i = 1:N
            temp = meshX(i+1) - meshX(i)
            if temp > maxH then
                maxH = temp
            end
        end
        val = val * maxH
    end
endfunction

function val = evalRectErr(f, numQuadpts, hScale, nind, pScale)

    if argn(2) < 5 then
        pScale = %T
    end

    // Get the evaluation points and their weights
    [pts, wts] = getQuadPtsWts(meshX(nind), meshX(nind+1), numQuadpts)

    // Calculate the sum
    val = 0
    for i = 1:numQuadpts
        temp = U(pts(i))
        val = val + wts(i) * (abs(temp - f(pts(i)))/(atol+rtol*abs(temp)))^2
    end

    // Scale the quadrature
    diam = meshX(nind+1) - meshX(nind)
    val = val * diam/2
    val = sqrt(val)

    // Scale by h
    pow = p+1
    if hScale then
        val = val * diam
        pow = p+1
    end
    if pScale then
        val = val^(1/pow)
    end
endfunction

// Function called by sci(x) and loi(x) which evaluates a H-B interpolant
// at x, with s as the points with a known solution and derivative, and
// w as the points with just a solution value
function y = evalHBI(x, s, w)
    sum1 = 0
    sum2 = 0
    sum3 = 0
    for j = 1:size(s)(2)
        sum1 = sum1 + (H(x, j, s, w) * U(s(j)))
        sum2 = sum2 + (Hbar(x, j, s, w) * Ux(s(j)))
    end
    for j = 1:size(w)(1)
        sum3 = sum3 + (G(x, j, s, w) * U(w(j)))
    end
    y = sum1 + sum2 + sum3
endfunction

// Computes w_j for Barycentric Lagrange interpolation. Where
// wx are the x values of the points being interpolated, and j
// is the index of w which is being calculated.
function w = baryW(wx, j)
    w = 1
    for k = 1:size(wx)(1)
        if k ~= j then
            w = w * (wx(j) - wx(k))
        end
    end
    w = 1/w
endfunction

// Calculates the value at x of a Lagarange interpolant to the
// collocation solution U at points wx.
function y = baryLagrange(x, wx)
    y = 0
    l = 1
    for j = 1:size(wx)(1)
        y = y + (baryW(wx, j)/(x - wx(j)))*U(wx(j))
        l = l * (x - wx(j))
    end
    y = l * y
    // Check for Nan
    if isnan(y) then
        ind = find(wx == x)
        y = U(wx(ind))
    end
endfunction


// Returns the necessary non-mesh super convergent points outside of a given
// interval for the SCI
function x = eScNonMeshP(ind)
    if ind == 1 then // Use two from next interval
        nextScnmp = scNonMeshP(meshX(2:3), p, 1)
        x(1) = nextScnmp(1)
        if p > 4 then
            x(2) = nextScnmp(2)
        else
            x(2) = meshX(3)
        end
    elseif ind == N then
        lastScnmp = scNonMeshP(meshX(N-1:N), p, 1)
        x(1) = lastScnmp(p-3)
        if p > 4 then
            x(2) = lastScnmp(p-4)
        else
            x(2) = meshX(N - 1)
        end
    else
        x(1) = scNonMeshP(meshX(ind-1:ind), p, 1)(p-3)
        x(2) = scNonMeshP(meshX(ind+1:ind+2), p, 1)(1)
    end
endfunction


// Function to evaluate the SCI of the current collocation solution at
// some x value.
```

```
function y = sci(x)
    // Find which subinterval x is within
    ind = getInd(x)

    // Get the internal and external non-mesh super convergent points
    meshP = meshX(ind:ind+1)
    inmscp = scNonMeshP(meshP, p, 1)
    enmscp = eScNonMeshP(ind)
    nmscp = cat(1, enmscp, inmscp)

    // Evaluate the H-B interpolant for SCI
    y = evalHBI(x, meshP, gsort(nmscp, 'r','i'))
endfunction

// Evalutates the LOI of saved Col.Sol. in coeffs at x
function y = loi(x)
    // Find which subinterval x is within
    ind = getInd(x)

    // Get the mesh points of that subinterval
    meshP = meshX(ind:ind+1)

    // Get the internal points to interpolate at
    intP = scNonMeshP(meshP, p - 1, 1)
    if intP == -1 then
        intP = []
    end
    y = evalHBI(x, meshP, intP)
endfunction

function y = l2(x)
    ind = getInd(x)
    intP = getL2pts(meshX(ind), meshX(ind+1), p)
    y = baryLagrange(x, intP)
endfunction

// Returns the error estimate from the LOI for rectangle xi, yi
function e = rLoiErr(xi)
    e = evalRectErr(loi, p+1, %T, xi)
endfunction

// Returns the error estimate from the LOI for rectangle xi, yi
function e = rLoiErrNS(xi)
    e = evalRectErr(loi, p+1, %F, xi)
endfunction

// Returns the global error estimate from the LOI
function e = gLoiErr()
    e = evalGlobalErr(loi, p+1, %T)
endfunction

// Returns the error estimate from the L2 for rectangle xi, yi
function e = rL2Err(xi)
    e = evalRectErr(l2, p+1, %T, xi)
endfunction

// Returns the error estimate from the L2 for rectangle xi, yi
function e = rL2ErrNS(xi)
    e = evalRectErr(l2, p+1, %F, xi)
endfunction

// Returns the global error estimate from the L2
function e = gL2Err()
    e = evalGlobalErr(l2, p+1, %T)
endfunction

// Returns the error estimate from the SCI for rectangle xi, yi
function e = rSciErr(xi)
    e = evalRectErr(sci, p+2, %F, xi)
endfunction

// Retruns the global error estimate from the SCI
function e = gSciErr()
    e = evalGlobalErr(sci, p+2, %F)
endfunction

// Returns the actual error for rectangle of coordinate xi, yi
function e = rActErr(xi)
    e = evalRectErr(truu, p+2, %F, xi)
endfunction

// Returns the actual global error for the collocation solution
function e = gActErr()
```

```
    e = evalGlobalErr(truu, p+1, %F)
endfunction
```

err2D.sci

```
// Evalauates a Hermite-Birkhoff interpolant at (x, y) based off of
// sx/y as the points in x/y where solution value and derivative with
// respect to x/y is interpolated. wx/y are points in x/y where just
// solution value is interpolated. Usx/y are the derivatives with
// respect to x/y which are evaluated, while Usw has all of the
// solution values being interpolated.
function z = evalHBI(x, y, sx, sy, wx, wy, Usx, Usy, Usw)

    // Save H evals
    Hx(1) = H(x, 1, sx, wx)
    Hx(2) = H(x, 2, sx, wx)
    Hy(1) = H(y, 1, sy, wy)
    Hy(2) = H(y, 2, sy, wy)

    // Save Hbar evals
    Hbarx(1) = Hbar(x, 1, sx, wx)
    Hbarx(2) = Hbar(x, 2, sx, wx)
    Hbary(1) = Hbar(y, 1, sy, wy)
    Hbary(2) = Hbar(y, 2, sy, wy)

    // Save G evals
    numWx = max(size(wx))
    numWy = max(size(wy))
    for i = 1:numWx
        Gx(i) = G(x, i, sx, wx)
    end
    for i = 1:numWy
        Gy(i) = G(y, i, sy, wy)
    end

    // Initialize the sum
    z = 0

    // Add value from Hx terms
    for i = 1:2
        // Set ind
        if i == 2 then
            ind = numWx+2
        else
            ind = 1
        end

        // HxGy terms
        for j = 1:numWy
            z = z + Hx(i) * Gy(j) * Usw(ind, j+1)
        end

        // HxHy terms
        z = z + Hx(i) * Hy(1) * Usw(ind, 1)
        z = z + Hx(i) * Hy(2) * Usw(ind, numWy+2)

        // HxHbary terms
        z = z + Hx(i) * Hbary(1) * Usy(1, ind)
        z = z + Hx(i) * Hbary(2) * Usy(2, ind)
    end

    // Add value from Hbarx terms
    for i = 1:2
        // Set ind
        if i == 2 then
            ind = numWx+2
        else
            ind = 1
        end

        // HbarxGy terms
        for j = 1:numWy
            z = z + Hbarx(i) * Gy(j) * Usx(i, j+1)
        end

        // HbarHy terms
        z = z + Hbarx(i) * Hy(1) * Usx(i, 1)
        z = z + Hbarx(i) * Hy(2) * Usx(i, numWy+2)
    end
    // Add value from Gx terms
    for i = 1:numWx
        // GxHy terms
        z = z + Gx(i) * Hy(1) * Usw(i+1, 1)
        z = z + Gx(i) * Hy(2) * Usw(i+1, numWy+2)

        // GxHbary terms
        z = z + Gx(i) * Hbary(1) * Usy(1, i+1)
        z = z + Gx(i) * Hbary(2) * Usy(2, i+1)

        // GxGy terms
        for j = 1:numWy
            z = z + Gx(i) * Gy(j) * Usw(i+1, j+1)
        end
    end
endfunction

// Returns the evaluation of a tensor product Lagrange interpolant
// at (x, y). wx / wy are the coordinates in X / Y that are interpolated
// and Uvals is the value to interpolate at those points.
function z = evalLI(x, y, wx, wy, Uvals)
    z = 0
    for i = 1:max(size(wx))
        for j = 1:max(size(wy))
            z = z + G(x, i, [], wx) * G(y, j, [], wy) * Uvals(i, j)
        end
    end
endfunction

// Calculates a scaled error based off of global atol and rtol with a L2 norm.
// f is the function used to calculate the error (Setting this as truu gives
// the actual error). numP is the number of points to use for the Gaussian
// quadrature. If hScale == %T then the error will be scale by the interval
// size in X and Y from the first interval of each.
function e = evalGlobalErr(func, numP, hScale)
    xL = meshX(1)
    xH = meshX(N+1)
    yL = meshY(1)
    yH = meshY(M+1)
    ptsX = getGaussPts(xL, xH, numP)
    ptsY = getGaussPts(yL, yH, numP)
    weights = getQuadWts(numP)
    e = 0
    for ix = 1:numP
        for iy = 1:numP
            temp = U(ptsX(ix), ptsY(iy), 1, 1)
            e = e + weights(ix) * weights(iy) * ((temp - func(ptsX(ix), ptsY(iy)))/
                ..
            (atol + rtol*temp))^2
        end
    end
    e = e * (xH-xL)/2 * (yH-yL)/2
    e = sqrt(e)
    if hScale then
        e = e * (meshX(2) - meshX(1)) * (meshY(2) - meshY(1))
    end
endfunction

// Calculates a scaled error for rectangle of coordinate xi, yi based off
// of global atol and rtol with a L2 norm. f is the function used to calculate
// the error (Setting this as truu gives the actual error). numP is the number
// of points to use for the Gaussian quadrature. If hScale == %T then the error
// will be scale by the interval size in X and Y from the first
// interval of each.
function e = evalRectErr(func, numP, hScale, ix, iy)
    xL = meshX(ix)
    xH = meshX(ix+1)
    yL = meshY(iy)
    yH = meshY(iy+1)
    ptsX = getGaussPts(xL, xH, numP)
    ptsY = getGaussPts(yL, yH, numP)
    weights = getQuadWts(numP)
    e = 0
    for i = 1:numP
        for j = 1:numP
            temp = U(ptsX(i), ptsY(j), 1, 1)
            e = e + weights(i) * weights(j) * ((temp - func(ptsX(i), ptsY(j)))/ ..
            (atol + rtol*temp))^2
        end
    end
    e = e * ((xH-xL)/2) * ((yH-yL)/2)
    e = sqrt(e)
    if hScale then
        e = e^(1/p)// * (xH-xL) * (yH-yL)
    else
        e = e^(1/(p+1))
    end
endfunction

// Saves the function value sthat will be interpolated by a Lagrange
// interpolant. wx / wy are the points in X / Y to interpolate at.
function vals = saveSolVals(wx, wy)
    for i = 1:size(wx)(1)
        for j = 1:size(wy)(1)
            vals(i, j) = U(wx(i), wy(j), 1, 1)
        end
    end
endfunction

// Returns function evaluations at the points for the SCI/LOI which can be reused
// sx/y is the mesh points in x/y for the interval being evaluated while
// wx/y are the points in x/y where just solution values are interpolated.
function [Usx, Usy, Usw] = saveFunctionEvals(sx, sy, wx, wy)

    // Get the numbr of w points
    numWx = size(wx)(1)
    numWy = size(wy)(1)

    // X derivatives
    for i = 1:2
        Usx(i, 1) = U(sx(i), sy(1), 2, 1)
        for j = 1:numWy
            Usx(i, j+1) = U(sx(i), wy(j), 2, 1)
        end
        Usx(i, numWy+2) = U(sx(i), sy(2), 2, 1)
    end

    // Y derivatives
    for i = 1:2
        Usy(i, 1) = U(sx(1), sy(i), 1, 2)
        for j = 1:numWx
            Usy(i, j+1) = U(wx(j), sy(i), 1, 2)
        end
        Usy(i, numWx+2) = U(sx(2), sy(i), 1, 2)
    end

    // Solution values
    for i = 1:numWx+2
        if i == 1 then
            xp = sx(1)
        elseif i == numWx+2 then
            xp = sx(2)
        else
            xp = wx(i-1)
        end
        for j = 1:numWy+2
            if j == 1 then
                yp = sy(1)
            elseif j == numWy+2
                yp = sy(2)
            else
                yp = wy(j-1)
            end
            Usw(i, j) = U(xp, yp, 1, 1)
        end
    end
endfunction

// Returns the value of the SCI at (x, y)
function z = sci(x, y)
    xInd = getInd(x, meshX, N)
    yInd = getInd(y, meshY, M)
    sx = [meshX(xInd);meshX(xInd+1)]
    sy = [meshY(yInd);meshY(yInd+1)]
    wx = getsciW(xInd, meshX, p, N)
    wy = getsciW(yInd, meshY, q, M)
    [Usx, Usy, Usw] = saveFunctionEvals(sx, sy, wx, wy)
    z =  evalHBI(x, y, sx, sy, wx, wy, Usx, Usy, Usw)
endfunction

// Returns the value of the L2 at (x, y)
function z = l2(x, y)
```

```
    xInd = getInd(x, meshX, N)
    yInd = getInd(y, meshY, M)
    wx = getL2pts(meshX(xInd), meshX(xInd+1), p)
    wy = getL2pts(meshY(yInd), meshY(yInd+1), q)
    z = evalLI(x, y, wx, wy, saveSolVals(wx, wy))
endfunction

// Returns the value of the loi at (x, y)
function z = loi(x, y)
    xInd = getInd(x, meshX, N)
    yInd = getInd(y, meshY, M)
    sx = [meshX(xInd);meshX(xInd+1)]
    sy = [meshY(yInd);meshY(yInd+1)]
    wx = scNonMeshP(meshX(xInd:xInd+1)',p-1,1)
    wy = scNonMeshP(meshY(yInd:yInd+1)',q-1,1)
    [Usx, Usy, Usw] = saveFunctionEvals(sx, sy, wx, wy)
    z =  evalHBI(x, y, sx, sy, wx, wy, Usx, Usy, Usw)
endfunction

// Returns the error estimate from the LOI for rectangle xi, yi
function e = rLoiErr(xi, yi)
    e = evalRectErr(loi, p+1, %T, xi, yi)
endfunction

function e = rLoiErrNS(xi, yi)
    e = evalRectErr(loi, p+1, %F, xi, yi)
endfunction

// Returns the global error estimate from the LOI
function e = gLoiErr()
    e = evalGlobalErr(loi, p+1, %T)
endfunction

// Returns the error estimate from the L2 for rectangle xi, yi
function e = rL2Err(xi, yi)
    e = evalRectErr(l2, p+1, %T, xi, yi)
endfunction

function e = rL2ErrNS(xi, yi)
    e = evalRectErr(l2, p+1, %F, xi, yi)
endfunction

// Returns the global error estimate from the L2
function e = gL2Err()
    e = evalGlobalErr(l2, p+1, %T)
endfunction

// Returns the error estimate from the SCI for rectangle xi, yi
function e = rSciErr(xi, yi)
    e = evalRectErr(sci, p+2, %F, xi, yi)
endfunction

// Retruns the global error estimate from the SCI
function e = gSciErr()
    e = evalGlobalErr(sci, p+2, %F)
endfunction

// Returns the actual error for rectangle of coordinate xi, yi
function e = rActErr(xi, yi)
    e = evalRectErr(truu, p+1, %F, xi, yi)
endfunction

// Returns the actual global error for the collocation solution
function e = gActErr()
    e = evalGlobalErr(truu, p+1, %F)
endfunction
```