Error Estimation of Collocation Solutions

By

Andrew Fraser

A Thesis Submitted to

Saint Mary's University, Halifax, Nova Scotia

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Applied Science.

Oct 24, 2023, Halifax, Nova Scotia

Approved: Dr. Paul Muir
Supervisor
Department of Mathematics
and Computer Science

Approved: Dr. David Iron
External Examiner
Department of Mathematics
and Statistics,
Dalhousie University

Approved: Dr. Walt Finden
Supervisory Committee
Department of Mathematics
and Computer Science

Approved: Dr. Jiju Poovvancheri
Supervisory Committee
Department of Mathematics
and Computer Science

Date: Oct 24, 2023.

Abstract

Error Estimation of Collocation Solutions

By Andrew Fraser

Partial differential equations (PDEs) often arise in mathematical models that explain, investigate, or predict real-world phenomena. The error-controlled numerical solution of time-dependent PDEs with one spatial dimension is an area that has seen much work, but the error-controlled numerical solution of PDEs with more than one spatial dimension has not had much focus. One of the goals of this thesis is to extend established algorithms for spatial error estimation of numerical solutions of PDEs in one dimension to PDEs in two dimensions. We focus on spatial error estimation schemes for numerical solutions obtained through the use of B-spline Gaussian collocation. This thesis also includes an investigation into the impact of error control on the computation of numerical solutions to a time-dependent, one-dimensional COVID-19 PDE model.

Oct 24, 2023.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Differential equations are a natural choice to model the change of a given phenomenon in time. In recent years many models of the spread of COVID-19 have been based on using partial differential equations (PDEs), e.g. [1], [17]. Examples of other models that are based on PDEs are the growth of a brain tumour [14], and the population of an invasive species [10].

Differential equations typically do not have a closed-form solution and are complicated enough that the only feasible approach is to calculate an approximate solution using numerical methods implemented in software. Since we get an approximate solution, it will of course have an error associated with it.

For PDEs, the error of an approximate solution can be attributed to the spatial discretization and time integration. The spatial discretization error is associated with factors such as the size and shape of the spatial domain, the location of mesh points, the distance between mesh points, and the numerical method used to perform the discretization. The temporal integration is affected by the tolerance employed by the time integrator used in order to advance the approximate solution through time. The accuracy of the time integration depends on the time step and numerical method used to perform the time integration.

1.1 Overview of Thesis

In Chapter 2, we begin by discussing the collocation method for time-dependent PDEs in one spatial dimension (1D). We then introduce error estimation and control with an example of numerical software which implements both. This is followed by a discussion of two interpolation-based error estimation schemes and why it is desirable to use an interpolant as the basis for an error estimation scheme. We then discuss the generalization of collocation so that it can be applied to PDEs with two spatial dimensions (2D PDEs).

Chapter 3 considers a COVID-19 type model involving a system of 1D time-dependent PDEs which is solved using numerical software featuring error estimation and control. The model is solved for two different scenarios and with a range of error tolerances. The goal of this chapter is to demonstrate how error control can be useful for these types of models.

Chapter 4 introduces a two-dimensional Hermite-Birkhoff interpolant to be used for estimating the error of an approximate solution to a 2D PDE that has been computed using a 2D collocation method. We also introduce generalizations of existing 1D interpolation-based error estimation schemes to two dimensions. These depend on collocation solution and derivative values. A new two-dimensional interpolant for error estimation which does not depend on collocation solution derivative values is also introduced. Chapter 4 also contains numerical results from the testing of these interpolants to assess their usefulness for error estimation.

The thesis concludes with Chapter 5 which summarizes the results and identifies potential future work.

1.2 General Problem Definition

The numerical methods we will use within this thesis require the PDEs to be provided in certain forms. The following subsections introduce the general form for the different classes of PDEs considered in this thesis.

1.2.1 One-Dimensional Non-Time-Dependent

The general from we will use for a 1D non-time-dependent problem is as follows:

The ordinary differential equation (ODE) is given by

$$u_{xx}(x) = f(x, u(x), u_x(x))), \tag{1.1}$$

where $u(x)$ is the true solution of the differential equation. The spatial domain is defined by $A \leq x \leq B$. The boundary conditions are $g_a(u(A), u_x(A)) = 0$ and $g_b(u(B), u_x(B)) = 0$.

1.2.2 One-Dimensional Time-Dependent

The general from we will use for a 1D time-dependent problem is as follows:

The PDE is given by

$$u_t(x,t) = f(x, t, u(x,t), u_x(x,t), u_{xx}(x,t))), \tag{1.2}$$

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

$$u(x, t_0) = \mu(x), \tag{1.3}$$

where $\mu(x)$ is a function that represents the beginning state of the solution across the spatial domain at time t_0. The boundary conditions are

$$g_a(t, u(A,t), u_x(A,t)) = 0 \text{ and } g_b(t, u(B,t), u_x(B,t)) = 0.$$

1.2.3 Two-Dimensional Non-Time-Dependent

The general form we will use for a 2D non-time-dependent problem is as follows:

The PDE is given by

$$u_{xx}(x,y) + u_{yy}(x,y) = f(x,y,u(x,y),u_x(x,y),u_y(x,y),u_{xy}(x,y)), \tag{1.4}$$

where $u(x,y)$ is the true solution to the PDE. The spatial domain is defined by $A \leq x \leq B$, $C \leq y \leq D$. The boundary conditions are:

$$g_a(y,u(A,y),u_y(A,y),u_x(A,y)) = 0, \quad g_b(y,u(B,y),u_y(B,y),u_x(B,y)) = 0,$$
$$g_c(x,u(x,C),u_y(x,C),u_x(x,C)) = 0, \quad g_d(x,u(x,D),u_y(x,D),u_x(x,D)) = 0.$$

Note that at each of the four corners of the spatial domain, the two boundary conditions which are relevant must be consistent. For example, when $x = A$ and $y = C$, we must have:

$$g_a(C,u(A,C),u_y(A,C),u_x(A,C)) = g_c(A,u(A,C),u_y(A,C),u_x(A,C)).$$

1.2.4 Two-Dimensional Time-Dependent

The general form we will use for a 2D time-dependent problem is as follows:

The PDE is given by

$$u_t(x,y,t) = f(x,y,t,u(x,y,t),u_x(x,y,t),u_{xx}(x,y,t),u_y(x,y,t),u_{yy}(x,y,t),u_{xy}(x,y,t)), \tag{1.5}$$

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

The initial condition is:

$$u(x,y,t_0) = \mu(x,y), \tag{1.6}$$

where $\mu(x,y)$ is a function which represents the beginning state of the solution across the spatial domain at time t_0. The boundary conditions are:

$$g_a(t,y,u(A,y,t),u_y(A,y,t),u_x(A,y,t)) = 0, \quad g_b(t,y,u(B,y,t),u_y(B,y,t),u_x(B,y,t)) = 0,$$
$$g_c(t,x,u(x,C,t),u_y(x,C,t),u_x(x,C,t)) = 0, \quad g_d(t,x,u(x,D,t),u_y(x,D,t),u_x(x,D,t)) = 0.$$

Note that at each of the four corners of the spatial domain, the two boundary conditions which are relevant must be consistent. For example, when $x = A$ and $y = C$, we must have:

$$g_a(t, C, u(A, C), u_y(A, C), u_x(A, C)) = g_c(t, A, u(A, C), u_y(A, C), u_x(A, C)).$$

Chapter 2

Background

This chapter introduces the B-spline Gaussian collocation method [18] for time-dependent partial differential equations with one and two spatial dimensions and non-time-dependent PDEs with two spatial dimensions. Error estimation and control are also introduced along with some examples of software which implement these concepts. These topics are all discussed in the context of the BACOLI software package [16]. Interpolation based error estimation methods implemented in BACOLI are also discussed.

2.1 B-spline Gaussian Collocation for 1D time-dependent PDEs

Collocation is a method of solving PDEs which gives a continuous solution approximation across the spatial domain at a set of discrete points in time along the temporal domain. The general idea of collocation is to represent the space of all potential approximate solutions as a linear combination of basis functions. We assume a B-spline basis which means that the basis functions are piecewise polynomials of a given degree, p. The method then calculates the coefficients of these basis functions which we can use with the basis functions to obtain an approximate solution to the PDE.

To begin the calculation, the number of subintervals into which the spatial domain will be divided, N, and the degree of the basis functions, p, must be set. The number of subintervals can be any positive integer while the degree must be greater than 3 and is typically less than 8. Both of

these parameters, as they are increased, can increase the accuracy of the approximate solution, but increasing these parameters will also lead to a greater computational cost.

We will consider the simple case of the spatial domain being equally divided into N subintervals by $N+1$ equidistant mesh points labelled $x_i : i = 1, 2, ..., N+1$, where $x_i = A$ and $x_{N+1} = B$. As mentioned above, we assume that the spatially dependent basis functions are B-splines [7]. The B-spline functions are piecewise polynomials defined on the spatial mesh $\{x_i\}_{i=1}^{N+1}$. To define B-spline basis functions, $b_i(x) : i = 1, 2, 3, ..., NCPTS$, with number of components $NCPTS = N(p-1)+2$, of degree p over the N subintervals, with C^1-continuity, we define a knot sequence based on the mesh points. This knot sequence will have $NCPTS + p + 1$ points and is constructed by repeating the points x_1 and x_{N+1}, $p+1$ times and repeating all other mesh points, x_2 to x_N, $p-1$ times. Further details on the construction of B-spline basis functions are available in [7].

Each of the $NCPTS$ basis functions will have a corresponding time-dependent coefficient. These coefficients will be defined as $c_i(t) : i = 1, 2, ..., NCPTS$. We can then express the approximate solution, $U(x,t)$, in the form,

$$U(x,t) = \sum_{i=1}^{NCPTS} c_i(t) b_i(x). \tag{2.1}$$

The collocation points are points within each subinterval where we require the approximate solution to satisfy the PDE. There are $p-1$ points per subinterval and they are set as the mapping of the $p-1$ Gauss-Legendre points from $[-1, 1]$ onto $[x_i, x_{i+1}]$, for each of the N subintervals. Figure 2.1 shows an example of this. There will be a total of $N(p-1) = NCPTS - 2$ collocation points, which we will label $\gamma_i : i = 1, 2, ..., NCPTS - 2$.

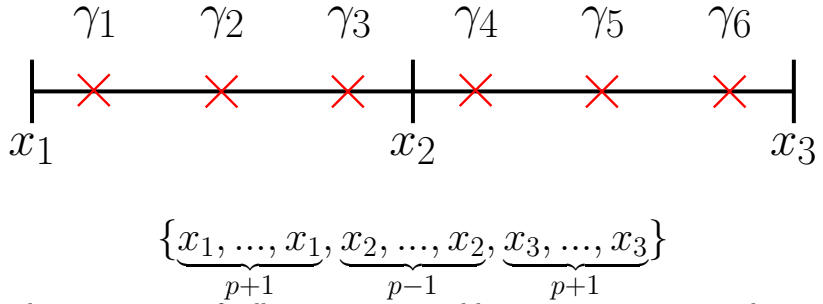

Figure 2.1: Visual representation of collocation points and knot sequence across 2 subintervals with $p = 4$.

The collocation conditions, obtained by requiring that the approximate solution satisfy the PDE

at the collocation points, have the following form:

$$f\left(\gamma_i, t, U\left(\gamma_i, t\right), U_x\left(\gamma_i, t\right), U_{xx}\left(\gamma_i, t\right)\right) - U_t\left(\gamma_i, t\right) = 0 : i = 1, 2, \ldots, NCPTS - 2. \tag{2.2}$$

In addition to the above $NCPTS - 2$ equations, we use the 2 boundary conditions to complete the system. We require the approximate solution to satisfy the boundary conditions at $x = A$ and $x = B$. The general form for these conditions is:

$$g_a(t, U(A, t), U_x(A, t)) = 0, \text{ and } g_b(t, U(B, t), U_x(B, t)) = 0. \tag{2.3}$$

Combining the boundary and collocation conditions, we obtain a $NCPTS \times NCPTS$ system of time-dependent, non-linear differential-algebraic equations (DAEs) whose solution gives the B-spline coefficients, $c_i(t)$, that define the approximate solution. A DAE solver can then be used to solve for these coefficients at a set of time points, across the temporal domain. These coefficients can then be used to represent the approximate solution at these points in time across the spatial domain. To begin the computation, an initial set of B-spline coefficients based on the provided initial conditions must be calculated and then provided to the DAE solver. That is, we project the initial conditions onto the B-spline basis at time t_0. Also, due to the locality of the B-spline basis functions, the equations that make up the DAE system can be ordered so that the Jacobian of the system has an almost-block diagonal (ABD) structure [8], which can be leveraged by the DAE solvers to speed up the computation.

When B-spline Gaussian collocation is applied to a 1D ordinary differential equation (ODE) the error of the collocation solution, $U(x)$, has the form [2] at non-mesh points:

$$u^{(j)}(x) - U^{(j)}(x) = u^{(p+1)}(x_i) P^{(j)}\left(\frac{x - x_i}{h_i}\right) h_i^{p+1-j} + O(h_i^{p+2-j}) + O(h^{2(p-1)}), \tag{2.4}$$

where $x_i \leq x \leq x_{i+1}$, $i = 1, ..., N$, $u(x)$ represents the true solution, $u^{(j)}(x)$ is the jth derivative of $u(x)$, $j = 0, 1, ..., p$, and where,

$$P(\xi) = \frac{1}{(p-1)!} \int_0^{\xi} (t-\xi) \sum_{r=1}^{p-1} (t-\rho_r) dt. \tag{2.5}$$

where ρ_r represents the mapping of the $p-1$ Gauss points onto $[0,1]$.

At mesh points the error satisfies:

$$u^{(j)}(x) - U^{(j)}(x) = O(h^{2(p-1)}) \tag{2.6}$$

where $j = 0, 1$.

In equation (2.4), assuming $p > 2$, the error will be dominated by the h^{p+1} term. As such, we expect that the error over the entire spatial domain will be $O(h^{p+1})$. Furthermore, since the h^{p+1} term is multiplied by the function P, at the zeros of P, the error will be dominated by the h^{p+2} term. These results have also been experimentally observed in the time-dependent PDE case [2].

In the BACOLI software package [16] the DAE solver used is DASSL [15]. DASSL is a DAE solver based on a family of backward differentiation formulas (BDFs) [15]. DASSL implements error control through adaptivity of the order of methods used as well as the size of the time step taken. The version of DASSL used in BACOLI has been modified to employ the COLROW [8] linear system solver to efficiently solve the ABD matrices that arise during the computation.

2.1.1 Error Estimation and Control for 1D PDEs

When calculating any approximate solution with numerical software, there will be some error associated with the approximation, meaning that there is a difference between the true solution and the approximate solution. In most practical cases where numerical software is being used to calculate an approximation, the true solution is not known, and the error of the approximate solution cannot be directly calculated. An error estimate can be calculated instead to give an idea of how good a given approximation is, which is a desirable feature of numerical software. The method by which a high-quality, reliable error estimate can be calculated depends on the method by which the approximate solution was calculated.

In cases where the true solution to a problem is unknown, a second approximate solution may be used to calculate an error estimate for an approximate solution. The second approximate solution can be calculated with a more accurate method which is expected to yield an approximate solution that is closer to the true solution than the original approximate solution. For example, if we had an approximate solution calculated using a collocation method with an error that is $O(h^k)$, we could then calculate a second approximate solution whose error is $O(h^{k+1})$. Here h is the maximum subinterval size and k is dependent on the degree of the basis functions. When an approximate solution has an error that is $O(h^k)$ we say that the approximation is of order k. The difference between this second approximate solution and the first approximate solution is then used to calculate the error estimate.

The error can be divided into two main contributors, the spatial and temporal errors. The spatial error is dependent on the width of the subintervals and the degree of the basis functions. The DAE solver must also feature error control so that the temporal error can be controlled so that it is less than the spatial error.

A desirable feature of numerical software is error control. Error control allows the user of the numerical software to specify a tolerance, i.e., an upper bound on the estimated error, and the software will attempt to return an approximate solution whose estimated error satisfies the provided tolerance. This not only has the benefit of the user being returned an approximation which meets their desired tolerance but also can allow for the approximate solution to be calculated more efficiently.

Error control is typically achieved by using adaptive methods. The BACOLI software [16] implements spatial error control through spatial mesh refinement. Once the collocation method has been applied, the DAE systems are solved by the DAE solver using a slightly sharper tolerance than the user provided tolerance. When the DAE solver returns from each time step, the error across the spatial domain is estimated and compared with the user-supplied error tolerance. If this tolerance is not met, the software then evaluates if more subintervals should be added, so that the error tolerance will be met. Conversely, if the estimated error is significantly smaller than the error tolerance, the algorithm can also adjust the mesh to have fewer subintervals. As an approximate

solution with more subintervals will tend to take more computational time, error control can allow for a more optimal number of subintervals to be used while still returning an approximation that satisfies the tolerance. In addition to determining the optimal number of subintervals, the mesh refinement algorithm also determines the optimal placement of the mesh points across the spatial domain so that the estimated error will be evenly distributed over the subintervals of the spatial domain.

2.1.2 Interpolation Based Error Estimation for 1D PDEs

While calculating a second, higher accuracy, approximate solution can enable the calculation of an error estimate, it also incurs extra computational cost. If a full collocation method, using a B-spline basis of degree $p + 1$, is used to obtain the second approximate solution, this will cause the overall computation time to be more than double that of the original approximate solution. Interpolation based error estimation aims to avoid this extra computational cost by instead using an interpolant to the original approximate solution; this is relatively cheap to evaluate compared to the computation of a second collocation solution. The difference between the interpolant and the original approximate solution can be used to calculate an error estimate.

The BACOLI software package [16] is a PDE solver based on B-spline Gaussian collocation as introduced above. It also features two distinct interpolants that can be used for calculating error estimates. The Superconvergent Interpolant (SCI) [2], an approximate solution that is of one order higher than the original approximate solution, and the Lower Order Interpolant (LOI) [3], an approximate solution that is of one order lower than that of the original approximate solution. When using the SCI error estimation method, BACOLI is said to be operating in standard error control mode. When the LOI error estimation method is used within BACOLI, the software is said to be operating in local extrapolation mode. This error estimate provides an overestimate of the error but has the advantage that its construction is simpler than that of the SCI.

To have an order of accuracy that is one order higher than that of the collocation solution, the SCI ensures that the data error dominates the interpolation error by interpolating the mesh point

solution and derivative values and the collocation solution at a sufficient number of special points within each subinterval for which the order is one order higher than that of the collocation solution at a non-mesh point. To obtain sufficiently many points, the SCI also interpolates the nearest higher-order points from each adjacent subinterval. The LOI gives an approximate solution of one order lower by interpolating sufficiently few points so that the interpolation error dominates the data error of the collocation solution and derivative values that it interpolates.

Both the SCI and LOI are expressed in a Hermite-Birkhoff interpolant form [9], with points w_j representing points where the approximate solution is evaluated and $s_i : i = 1, 2$ representing points where both solution and derivative values are interpolated. The number of w_j points is dependent on p, as well as which of the SCI or LOI error estimation methods is being used. The general form is as follows:

$$q(x) = \sum_{i=1}^{|s|} H_i(x)\, u(s_i) + \sum_{i=1}^{|s|} \bar{H}_i(x)\, u_x(s_i) + \sum_{j=1}^{|w|} G_j(x)\, u(w_j), \tag{2.7}$$

where,

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

and where,

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

Both the SCI and LOI can return reliable error estimates allowing for the BACOLI software package [16] to feature error control, while also doing so efficiently.

2.2 B-spline Gaussian Collocation for 2D PDEs

Collocation can also be used to calculate approximate solutions for PDEs with more than one spatial dimension. In this section, we will introduce collocation for time-dependent and non-time-

dependent PDEs with two spatial dimensions. The method introduced here is based on the implementation in [13].

2.2.1 B-spline Gaussian Collocation for 2D time-dependent PDEs

As with the 1D case, we will begin dividing the spatial domain into N intervals in the x-domain and M intervals in the y-domain. This will create a grid of subrectangles across the spatial domain. However, it is useful to continue considering the subintervals of each spatial domain; these subintervals are used to define the basis functions and collocation points for each spatial dimension. We will also use degree p and q basis functions in x and y respectively.

The mesh points, defined as $x_i : i = 1, 2, 3, ..., N+1$, and $y_i : i = 1, 2, 3, ..., M+1$, are used to create a knot sequence in x and y following the same approach as in the 1D case. These knot sequences will then define $NCPT_x = N(p-1)+2$ B-spline basis functions in x and $NCPT_y = M(q-1)+2$ B-spline basis functions in y. These basis functions will be $b_i(x) : i = 1, 2, 3, ..., NCPTS_x$ and $d_i(y) : 1, 2, 3, ..., NCPT_y$. Instead of each basis function having a corresponding time-dependent coefficient, each product of basis function in x and y will have a coefficient, $c_{i,j}(t) : i = 1, 2, 3, ..., NCPT_x, j = 1, 2, 3, ..., NCPT_y$.

This leads to an approximate solution that has the form,

$$U(x, y, t) = \sum_{i=1}^{NCPT_x} \sum_{j=1}^{NCPT_y} c_{i,j}(t) b_i(x) d_j(y). \tag{2.8}$$

The collocation points are once again defined as the mapping of Gauss-Legendre points from $[-1, 1]$ onto each subinterval of each spatial domain. There will be $N(p-1) = NCPT_x - 2$ and $M(q-1) = NCPT_y - 2$ points in the x and y domains, respectively. We will define the collocation points as $\gamma_i : i = 1, 2, ..., NCPT_x - 2$ in x and $\delta_i : i = 1, 2, ..., NCPT_y - 2$ in y. By requiring the approximate solution to satisfy the PDE at these collocation points, we obtain the following collocation conditions,

$$f\left(\gamma_i, \delta_j, t, U\left(\gamma_i, \delta_j, t\right), U_x\left(\gamma_i, \delta_j, t\right), U_{xx}\left(\gamma_i, \delta_j, t\right), U_y\left(\gamma_i, \delta_j, t\right), U_{yy}\left(\gamma_i, \delta_j, t\right), U_{xy}\left(\gamma_i, \delta_j, t\right)\right)$$

$$-U_t\left(\gamma_i, \delta_j, t\right) = 0, \quad (2.9)$$

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

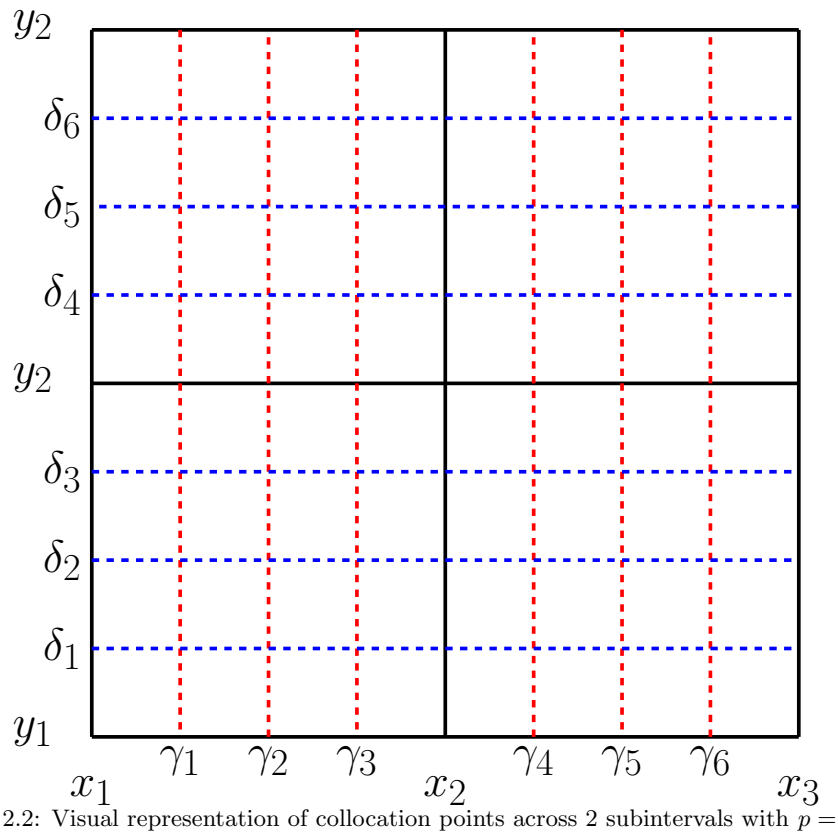

Figure 2.2: Visual representation of collocation points across 2 subintervals with $p = q = 4$.

Once again the remaining conditions are obtained by using the boundary conditions. The boundary conditions are imposed at the collocation points along each edge of the spatial boundary, and each of the four corners of the complete spatial domain for the problem. These boundary conditions will have the following form:

$$g_a(t, C, U(A, C, t), U_y(A, C, t), U_x(A, C, t)) = 0, \quad g_a(t, D, U(A, D, t), U_y(A, D, t), U_x(A, D, t)) = 0,$$

$$g_b(t, C, U(B, C, t), U_y(B, C, t), U_x(B, C, t)) = 0, \quad g_a(t, D, U(B, D, t), U_y(B, D, t), U_x(B, D, t)) = 0,$$

$$g_a(t, \delta_j, U(A, \delta_j, t), U_y(A, \delta_j, t), U_x(A, \delta_j, t)) = 0, \quad g_b(t, \delta_j, U(B, \delta_j, t), U_y(B, \delta_j, t), U_x(B, \delta_j, t)) = 0,$$

$$g_c(t,\gamma_i,U(\gamma_i,C,t),U_y(\gamma_i,C,t),U_x(\gamma_i,C,t))=0, \quad g_d(t,\gamma_i,U(\gamma_i,D,t),U_y(\gamma_i,D,t),U_x(\gamma_i,D,t))=0,$$
$$i=1,2,\ldots,NCPTX-2, \quad j=1,2,\ldots,NCPTY-2.$$

After the initial B-spline coefficients have been determined based on the initial condition for the problem, the boundary and collocation conditions are then combined to form a DAE system which can then be solved using a DAE solver to obtain the coefficients of the basis functions.

2.2.2 B-spline Gaussian Collocation for 2D non-time-dependent PDEs

The collocation method may also be used to solve non-time-dependent PDEs. The entire collocation process, in this case, is similar to a single time step of the time-dependent case. The differences arise in the collocation conditions which no longer include a $U(x,y)_t$ term, causing the resultant system of boundary and collocation conditions to no longer be a system of DAEs but instead a system of non-linear equations with the B-spline basis coefficients as the unknowns.

We must first divide the spatial domain into N intervals in the x-domain and M intervals in the y-domain. We will also use degree p and q basis functions in x and y respectively. The knots are created in the same manner as section 2.2.1 to define the B-spline basis functions, $b_i(x) : i = 1,2,3,...,NCPTS_x$, and $d_i(y) : 1,2,3,...,NCPT_y$. The basis functions will have coefficients $c_{i,j} : i = 1,2,3,...,NCPT_x, j = 1,2,3,...,NCPT_y$, which can be used to express the approximate solution in the form,

$$U(x,y) = \sum_{i=1}^{NCPT_x} \sum_{j=1}^{NCPT_y} c_{i,j} b_i(x) d_j(y). \tag{2.10}$$

We will define the collocation points as $\gamma_i : i = 1,2,...,NCPT_x-2$ in x and $\delta_i : i = 1,2,...,NCPT_y-2$ in y following the same procedure as in section 2.2.1. The collocation conditions are represented by the following equation:

$$f(\gamma_i,\delta_j,U(\gamma_i,\delta_j),U_x(\gamma_i,\delta_j),U_y(\gamma_i,\delta_j)) - U_{xx}(\gamma_i,\delta_j) - U_{yy}(\gamma_i,\delta_j) = 0, \tag{2.11}$$

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

The boundary conditions are of the form:

$$g_a(C, U(A,C), U_y(A,C), U_x(A,C)) = 0, \quad g_a(D, U(A,D), U_y(A,D), U_x(A,D)) = 0,$$
$$g_b(C, U(B,C), U_y(B,C), U_x(B,C)) = 0, \quad g_a(D, U(B,D), U_y(B,D), U_x(B,D)) = 0,$$
$$g_a(\delta_j, U(A,\delta_j), U_y(A,\delta_j), U_x(A,\delta_j)) = 0, \quad g_b(\delta_j, U(B,\delta_j), U_y(B,\delta_j), U_x(B,\delta_j)) = 0,$$
$$g_c(\gamma_i, U(\gamma_i,C), U_y(\gamma_i,C), U_x(\gamma_i,C)) = 0, \quad g_d(\gamma_i, U(\gamma_i,D), U_y(\gamma_i,D), U_x(\gamma_i,D)) = 0,$$
$$i = 1, 2, \ldots, NCPTX - 2, \quad j = 1, 2, \ldots, NCPTY - 2.$$

These conditions are then combined to create a system of non-linear equations which is solved for the unknown coefficients of the B-spline basis functions, using a non-linear system solver.

Chapter 3

Error Controlled Numerical Solution of a COVID-19 PDE Model

Recently the idea of mathematical modelling of epidemics, specifically COVID-19, has seen an extreme focus. Many of these models are compartmental models, which divide the population into distinct compartments and consider how these compartments interact with one another. The SIR model [12] is one of the first of this type and is one of the simplest, consisting of only three compartments: Susceptible (S), Infected (I), and Recovered (R). To more accurately model the spread of diseases with an incubation period, the SEIR model was developed as an extension of the SIR model. This adds the Exposed (E) category, which allows the model to account for the incubation period of a virus. Other variations of the SIR model have also been developed which account for other behaviours a virus may exhibit [1], [17].

The traditional SIR model is defined by a system of ODEs. These ODE models have the independent variable, time, and the resulting solution is the proportion of the population in each compartment through time.

These ODE models have also been extended to systems of PDEs which include a spatial and temporal domain. That is, there is an independent variable corresponding to time, and one or more corresponding to space. The simplest of these have one spatial dimension. These models still contain the same compartments as an ODE model, although the solution will represent the proportion of the population in each compartment across the spatial domain and through time. This allows for the modelling of phenomenon such as the interaction between two distant regions of high infection, or modelling the spatial spread of the disease. PDE models may also contain two spatial dimensions. This case is more natural, where we seek a solution approximation over a region and the spread of disease (i.e. the population of the compartments) throughout it over time.

3.0.1 Compartmental ODE Models

The first compartmental model, given by equations (3.1), was introduced by Kermack and McKendrick in 1927 [12]. This model is now known as a susceptible-infected-recovered (SIR) model. The population is separated into these three compartments and a system of equations is created which defines how each compartment affects the others. The system has the form,

$$\begin{aligned} \frac{dS}{dt} &= -\kappa SI, \\ \frac{dI}{dt} &= \kappa SI - lI, \\ \frac{dR}{dt} &= lI. \end{aligned} \tag{3.1}$$

The variables S, I, R from equations (3.1) represent the proportion of a total population, N, which is contained in each compartment. These variables represent the proportion of the population that is susceptible (S), infected (I), and removed (R) respectively. The constant, κ, represents the infectivity rate, and the constant, l, represents the rate of removal (sum of deaths and recoveries).

Variations of this model have also been developed which can factor in the dynamics of different diseases. The most relevant to the COVID-19 pandemic is the susceptible-exposed-infected-removed (SEIR) model, which accounts for the time period between when an individual is exposed and when they will become infectious. Equations (3.2) present an ODE SEIR model. The model has the form,

$$
\begin{aligned}
S_t(t) = &\quad \mu N - \mu S(t) - \tfrac{\beta}{N} S(t) I(t), \\
E_t(t) = &\quad \tfrac{\beta}{N} S(t) I(t) - \alpha E(t) - \mu E(t), \\
I_t(t) = &\quad \alpha E(t) - \gamma I(t) - \mu I(t), \\
R_t(t) = &\quad \gamma I(t) - \mu R(t).
\end{aligned} \tag{3.2}
$$

The variables S, E, I, R from equations (3.1) represent the proportion of a total population, N, which is contained in each compartment. These variables represent the proportion of the population that is susceptible (S), Exposed (E), infected (I), and removed (R) respectively. The death and birth rates, μ, are assumed to be equal. The infection rate is represented by the constant β. The incubation period to model the time between exposure and infection is determined by the constant α. The recovery rate is represented by the constant γ.

3.1 Numerical Solution of a Covid-19 PDE Model

In this section, we will consider a COVID-19-type PDE model, with two diffusion functions to represent two different scenarios. The BACOLI software package [16] is used to solve these problems using an equidistributed initial spatial mesh set by the BACOLI software; this defaults to a mesh with 10 subintervals. The number of collocation points used is, $kcol = 5$; this means that the degree of the B-spline basis functions will be $p = kcol + 1 = 6$. The error control mode used is standard error control which means that the SCI-based error estimation scheme will be used, and the error tolerance is only applied to the estimate of the absolute error.

3.1.1 A Covid-19 PDE Model

The Covid-19 model used in both scenarios involves the PDEs (3.3), [6]. The system has the form,

$$
\begin{aligned}
S(x,t)_t = &\quad D_S(x)S(x,t)_{xx} + \mu N - \mu S(x,t) - \tfrac{\beta}{N} S(x,t)I(x,t), \\
E(x,t)_t = &\quad D_E(x)E(x,t)_{xx} + \tfrac{\beta}{N} S(x,t)I(x,t) - \alpha E(x,t) - \mu E(x,t), \\
I(x,t)_t = &\quad D_I(x)I(x,t)_{xx} + \alpha E(x,t) - \gamma I(x,t) - \mu I(x,t), \\
R(x,t)_t = &\quad D_R(x)R(x,t)_{xx} + \gamma I(x,t) - \mu R(x,t).
\end{aligned} \tag{3.3}
$$

The spatial domain is $-5 \leq x \leq 5$, and the temporal domain is $0 \leq t \leq 50$. The value and role of parameters appearing in equations (3.3) are given in Table 3.1.

birth/death rate	μ	$\frac{0.01}{365}$
recovery rate	γ	0.06
transmission rate	β	0.9
incubation rate	α	0.125
total population	N	1

Table 3.1: Value and role of parameters for COVID-19 type PDE model.

The diffusion functions $D_s(x), D_E(x), D_I(x)$, and $D_R(x)$ will be discussed individually for each scenario. The initial conditions used are,

$$
S(x,0) = 1 - I(x,0), \quad I(x,0) = 0.2e^{-10(x+1)^2},
$$

$$
E(x,0) = R(x,0) = 0.
$$

Figure 3.1 showing the inital state of $I(x,0)$.

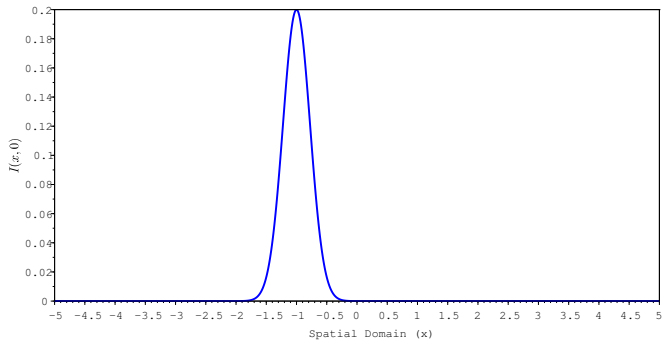

Figure 3.1: Plot of the initial condition I(x,0).

3.1.2 Scenario 1 Problem Definition

This scenario aims to represent two nearby population centres, and this is done by creating a diffusion function with two nearby local maximums; $D_S(x)$ is shown in Figure 3.2 and all four diffusion coefficients are defined in equations (3.4).

$$\begin{aligned} D_S(x) = D_E(x) = D_R(x) &= (\max Ds - \min Ds)e^{-10(\sqrt{x^2}-1)^2} + \min Ds, \\ D_I(x) &= D_E(x)/10, \end{aligned} \tag{3.4}$$

where $\max Ds = 0.05$ and $\min Ds = 0.001$. These points are chosen arbitrarily to provide a high level of diffusivity in the areas of dense population while still maintaining a baseline of diffusivity at all other areas.

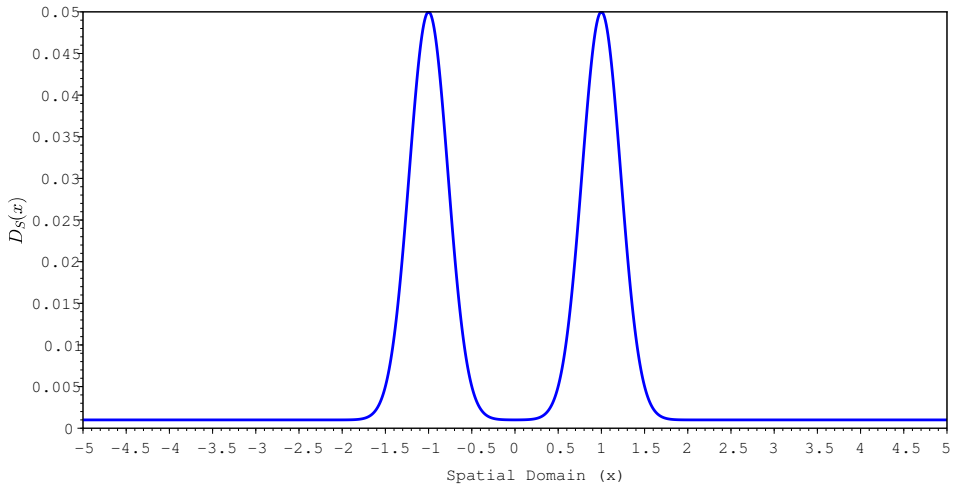

Figure 3.2: Plot of the diffusivity parameter $D_S(x)$ for Scenario 1.

As mentioned earlier, these models do not have a closed-form solution and thus we can not know the exact error of any approximate solution. However, we will use as a reference solution an approximate solution computed using a very sharp tolerance. We can then take the difference between this reference solution and a less accurate approximate solution to obtain a reasonable estimate of

the error for the lower tolerance (i.e., less accurate) approximate solution. The I component of the sharp tolerance approximate solution for Scenario 1 can be seen in Figure 3.3.

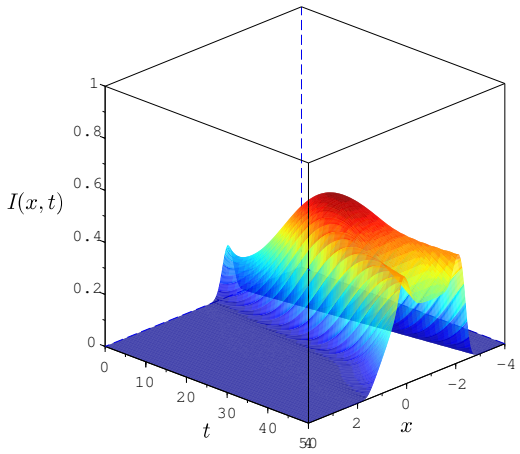

Figure 3.3: $I(x,t)$ solution component of the sharp tolerance tolerance solution to Scenario 1.

3.1.3 Scenario 2 Problem Definition

This scenario aims to represent a lone population centre, and employs a diffusion function with one local maximum. For this scenario, $D_S(x)$ is shown in Figure 3.4 and all four diffusion coefficients are defined in equations (3.5).

$$\begin{aligned} D_S(x) = D_E(x) = D_R(x) &= (\max Ds - \min Ds)e^{-10(x^2)} + \min Ds, \\ D_I(x) &= D_E(x)/10, \end{aligned} \tag{3.5}$$

where $\max Ds = 0.05$ and $\min Ds = 0.001$.

The I component of the sharp tolerance approximate solution for Scenario 2 can be seen in Figure 3.3.

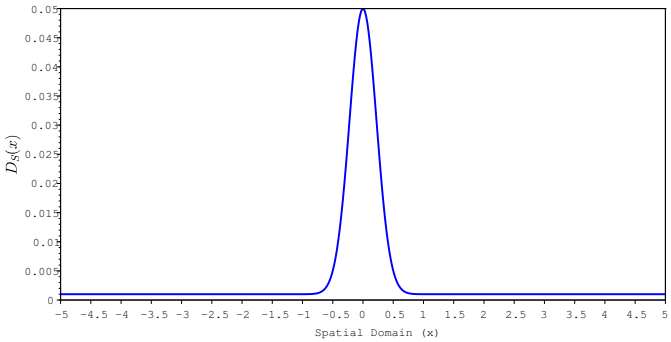

Figure 3.4: Plot of the diffusivity parameter $D_S(x)$ for Scenario 2.

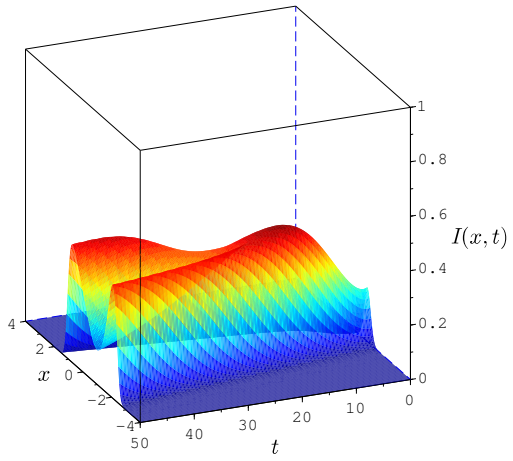

Figure 3.5: $I(x,t)$ solution component of the sharp tolerance solution to Scenario 2.

3.2 Numerical Results

The approximate solutions to these models were calculated using the error control BACOLI software [16] with a range of error tolerances, $10^{-i}, i = 2, 3, 4, ..., 10$.

This section contains two main types of plots. The first shows the absolute maximum of the difference between the coarser tolerance $I(x,t)$ solution approximations and the $I(x,t)$ reference solution approximation computed using the sharp tolerance of 10^{-10}, for t from 0 to 50. The second type of plot shows the two approximate solutions obtained using the most coarse and the most sharp tolerances (i.e., 10^{-2} and 10^{-10}) at a specific point in time.

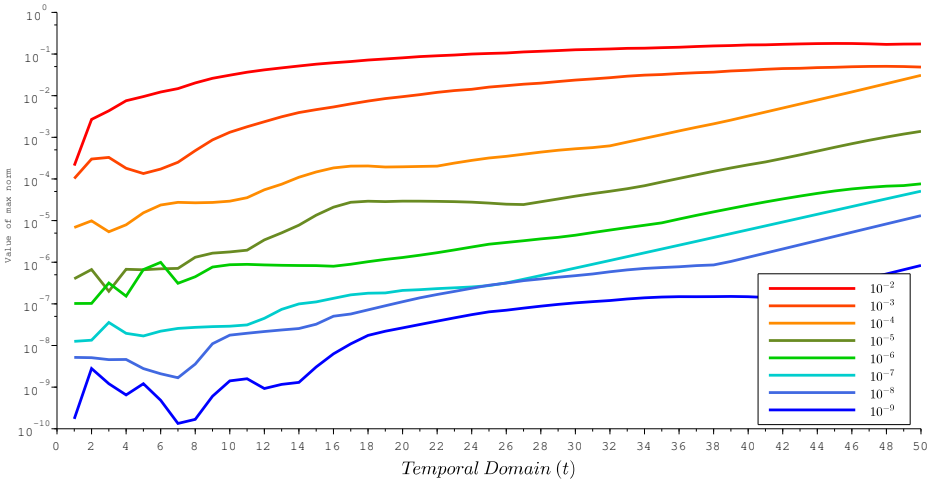

Figure 3.6: Absolute maximum difference between the lower tolerance $I(x,t)$ solutions and the $I(x,t)$ solution obtained using the sharpest tolerance, for Scenario 1.

3.2.1 Scenario 1 Results

In Figure 3.6 we plot the maximum difference, across the spatial domain, between the coarser tolerance $I(x,t)$ solution approximations and the reference sharp tolerance solution approximation. From this figure, we can see there is a clear trend of the coarser tolerance solution approximations converging to the sharp tolerance solution as the tolerance decreases, which suggests that the error control makes an impact on the accuracy of an approximate solution to this model. We also see that as the time advances the error increases in all of the solutions, eventually to the point where they no longer satisfy the provided absolute error tolerance. This is related to the exponential growth of some of the solution components over the time domain.

Figure 3.7 shows the solution approximation calculated with a tolerance of 10^{-2} and the reference solution (computed with a tolerance of 10^{-10}) at $t = 50$. From this plot, we see that the approximate solutions show different spatial behaviour. Also from Figure 3.7, we can see that these two solution approximations have a maximum difference of about 10^{-1}, or about 10% of the total population. Furthermore, we see that the coarse tolerance solution is negative on certain regions of the spatial domain; this result is unacceptable as $I(x,t)$ represents a proportion of the population and should always be zero or greater. This suggests that error control can have a significant impact on the

accuracy of an approximate solution of a COVID-19 PDE model.

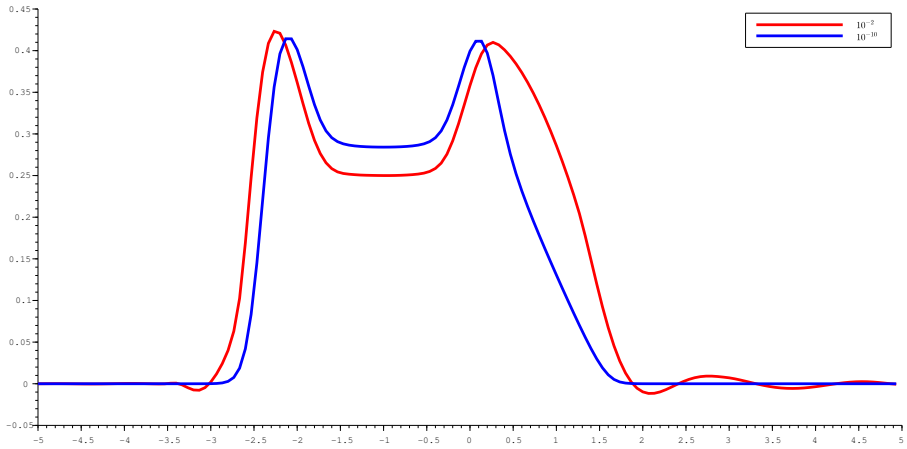

Figure 3.7: Comparison of coarse and sharp tolerance solutions to COVID-19 model at $t = 50$ for Scenario 1.

3.2.2 Scenario 2 Results

Figure 3.8 is similar to Figure 3.6 from Scenario 1. We see a decrease in error as the tolerance is sharpened, suggesting that error control is having an impact. However, the increase in the error through time is more noticeable. Similar to Scenario 1, in Figure 3.8, we also see the errors increasing beyond the tolerance as time progresses. However, the solution solved with a tolerance of 10^{-3} does begin to regain some accuracy near the end of the temporal domain.

In Figure 3.9 we see a plot that is similar to the one shown in Figure 3.7, with the coarse tolerance solution approximation being noticeably different from the reference solution approximation. In Figure 3.9, we see larger oscillations near the flat portions of the coarse tolerance solution approximation. We also note that the coarse tolerance solution is negative on certain regions of the spatial domain.

3.2.3 Results Discussion

In both scenarios, as the tolerance supplied to BACOLI is sharpened, the resultant approximate solutions converge to the reference solution. Also when using the coarsest tolerances, we see a

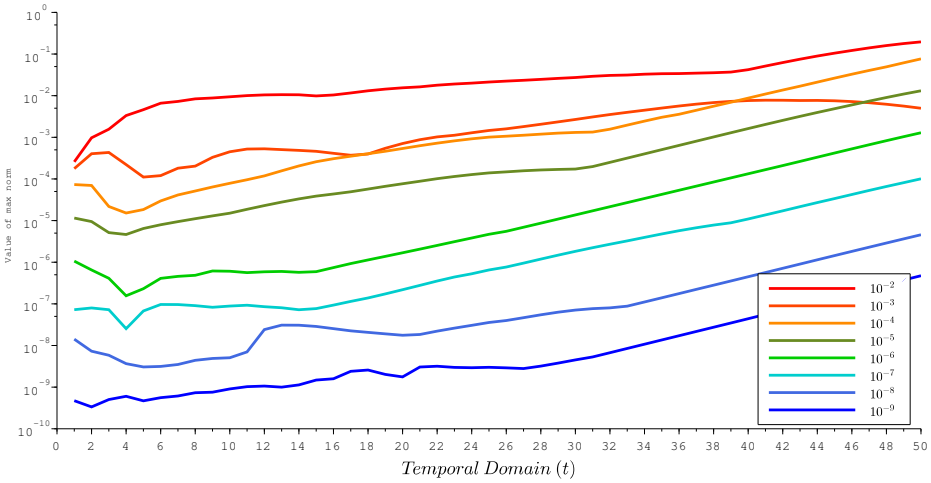

Figure 3.8: Absolute maximum difference of lower tolerance $I(x,t)$ solutions to the $I(x,t)$ solution with atol $= 10^{-10}$ for scenario 2.

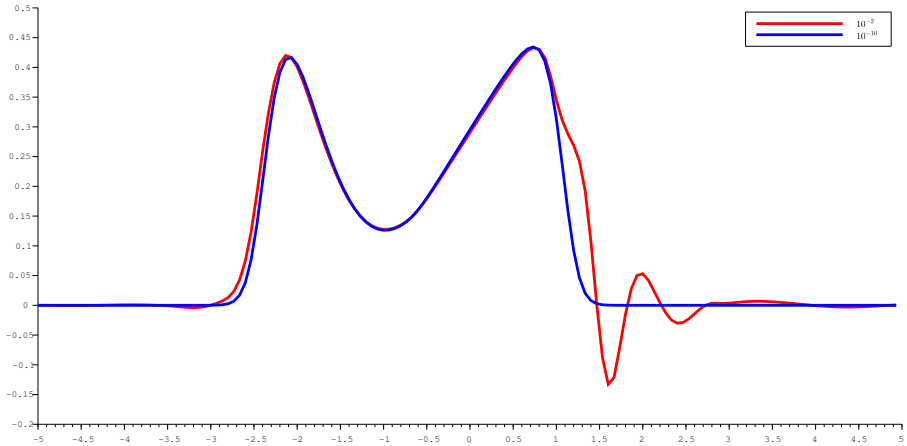

Figure 3.9: Comparison of coarse and sharp tolerance solutions to COVID-19 model at $t = 50$ for Scenario 2.

different spatial behaviour compared to the reference approximate solutions, as seen in Figures 3.7 and 3.9. If we were to consider a scenario where a group in charge of implementing COVID restrictions is performing modelling to help inform their decisions, and they were to use a PDE solver without error control, then the approximate solution they compute could easily have a large error, although the software would provide them with no indication of this. However, if they were to use a PDE solver with error control they would be able to specify a sharp tolerance and have reasonable confidence that they will compute an accurate solution approximation.

Chapter 4

2D Interpolants for Error Estimation of 2D Collocation Solutions

In this chapter, we introduce 2D generalizations of two previously developed 1D interpolants and one new 2D interpolant that can be used to calculate error estimates for 2D Gaussian collocation solutions. We will also assess the performance of these interpolants to investigate their rates of convergence and the quality of the error estimates for 2D collocation solutions that can be obtained using these interpolants.

4.1 2D Interpolants for 2D Gaussian Collocation Solutions

4.1.1 2D Hermite-Birkhoff Interpolants

Here we introduce a general form for a 2D Hermite-Birkhoff interpolant. These interpolants depend on solution and derivative values at several points associated with each grid of the spatial domain. We will label the points at which both solution and derivative values are interpolated as

s_1 and s_2 for the x domain, and t_1 and t_2 for the y domain. The points where only solution values are interpolated are labelled as w_i in the x domain and v_i in the y domain. The number of w_i, v_i, points will depend on the degree of interpolant being constructed. With these values defined, one can then represent the interpolant as follows (note that the superscripts (x), and (y), are used to differentiate between two versions of the function G with different parameters corresponding to each spatial dimension):

$$\begin{aligned} q(x,y) = \sum_{i=1}^{2}\sum_{i=j}^{2} \bar{H}_i(s_i)\bar{F}_j(t_j)U_{xy}(s_i,t_j) + \sum_{j=1}^{2}\sum_{i=1}^{2} \bar{H}_i(s_i)F_j(t_j)U_x(s_i,t_j) + \\ \sum_{i=1}^{2}\sum_{j=1}^{|v|} + \bar{H}_i(s_i)G_j^{(y)}(v_j)U_x(s_i,v_j) + \sum_{i=1}^{2}\sum_{j=1}^{2} H_i(s_i)\bar{F}_j(t_j)U_y(s_i,t_j) + \\ \sum_{j=1}^{2}\sum_{i=1}^{|w|} \bar{F}_j(t_j)G_i^{(x)}(w_i)U_y(w_i,t_j) + \sum_{i=1}^{2}\sum_{j=1}^{2} H_i(s_i)F_j(t_j)U(s_i,t_j) + \\ \sum_{i=1}^{2}\sum_{j=1}^{|v|} H_i(s_i)G_j^{(y)}(v_j)U(s_i,v_j) + \sum_{j=1}^{2}\sum_{i=1}^{|w|} F_j(t_j)G_i^{(x)}(w_i)U(w_i,t_j) + \\ \sum_{i=1}^{|w|}\sum_{j=1}^{|v|} G_i^{(x)}(w_i)G_j^{(y)}(v_j)U(w_i,v_j). \end{aligned} \tag{4.1}$$

where

$$H_i(x) = (1-(x-s_i)\lambda_i)\hat{H}_i(x),\ \bar{H}_i(x) = (x-s_i)\hat{H}_i(x),$$
$$\hat{H}_i(x) = \tfrac{\psi_i^2(x)\phi(x)}{\psi_i^2(s_i)\phi(s_i)},\ G_i^{(x)}(x) = \tfrac{\psi^2(x)\phi_i(x)}{\psi^2(s_i)\phi_j(s_i)},$$
$$F_j(y) = (1-(y-t_j)\Lambda_j)\hat{F}_j(y),\ \bar{F}_j(y) = (y-t_j)\hat{F}_j(y),$$
$$\hat{F}_j(y) = \tfrac{\Psi_i^2(y)\Phi(y)}{\Psi_i^2(t_i)\Phi(t_i)},\ G_j^{(y)}(y) = \tfrac{\Psi^2(y)\Phi_j(y)}{\Psi^2(t_j)\Phi_j(t_j)},$$

and

$$\psi_i(x) = \prod_{\substack{k=1\\k\neq i}}^{|s|}(x-s_k),\quad \psi(x) = \prod_{k=1}^{|s|}(x-s_k),$$
$$\phi_j(x) = \prod_{\substack{k=1\\k\neq j}}^{|w|}(x-w_k),\quad \phi(x) = \prod_{k=1}^{|w|}(x-w_k),$$
$$\lambda_i = \sum_{j=1}^{|w|}\tfrac{1}{s_i-w_j} + 2\sum_{\substack{j=1\\j\neq i}}^{2}\tfrac{1}{s_i-s_j},$$

$$\Psi_i(y) = \prod_{\substack{k=1\\k\neq i}}^{|t|}(y-t_k),\quad \Psi(y) = \prod_{k=1}^{|t|}(y-t_k),$$

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

4.1.2 The SCI in 2 Dimensions

The generalization of the 2D SCI that we will consider is constructed by interpolating the solution values and spatial derivatives for each dimension in the same manner as the 1D case. However, in order to have sufficiently many interpolation points, we also interpolate the cross derivative at the four corners of the sub-rectangle upon which the interpolant is constructed. If we consider a collocation solution of degree p in x and q in y and the sub-rectangle bounded by $[x_\alpha \leq x \leq x_\omega]$ and $[y_\alpha \leq y \leq y_\omega]$, we have $s_1 = x_\alpha$, $s_2 = x_\omega$, $t_1 = y_\alpha$, and $t_2 = y_\omega$. We will also have $p-1$ points, w_i, and $q-1$ points, v_j, of which $p-3$ and $q-3$ are set relatively across each spatial domain according to Table 4.1. The nearest interpolation points within each adjacent sub-rectangle are also included in order to obtain the remaining interpolation points for each dimension.

Figure 4.1 shows how these points, w_i and v_j, are set when $p = q = 6$ and the sub-rectangle for which we are constructing the interpolant is not on the edge of the spatial domain of the PDE. In such a case where the point closest to an edge would be outside the spatial domain of the problem we instead use the two nearest points from the adjacent sub-rectangle. Equation 4.1 can then be used to obtain the 2D SCI on a given sub-rectangle.

Degree of Collocation Solution			
4	5	6	7
0.5	0.3110177634953864	0.2113248654051871	0.1526267046965671
	0.6889822365046136	0.5	0.3747185964571342
		0.7886751345948129	0.6252814035428658
			0.8473732953034329

Table 4.1: Relative position of interpolation points corresponding to solution values within the sub-rectangle that are used in the construction of the 2D SCI.

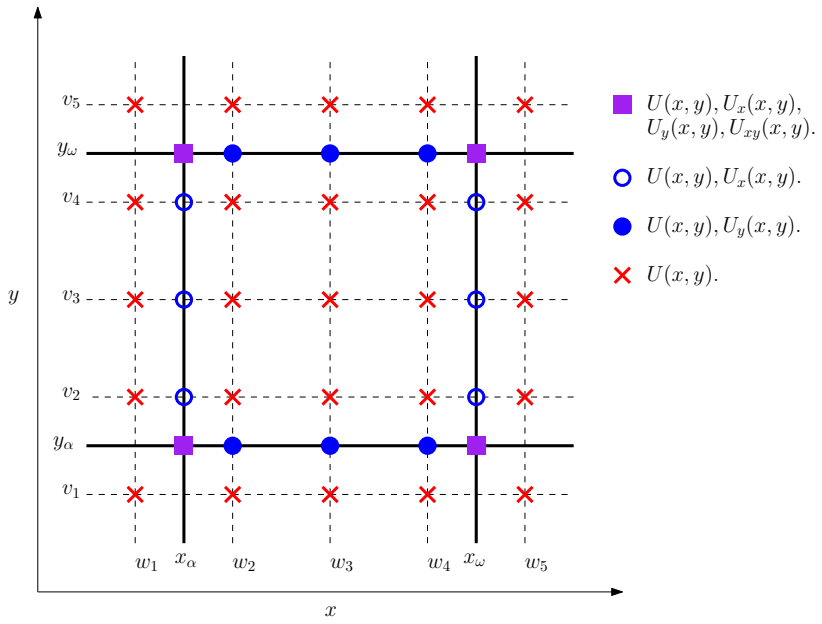

Figure 4.1: Visual representation of points where 2D SCI interpolates solution and derivative values when $p = q = 6$.

Degree of Collocation Solution			
4	5	6	7
	0.5	0.3110177634953864	0.2113248654051871
		0.6889822365046136	0.5
			0.7886751345948129

Table 4.2: Relative position of interpolation points corresponding to solution values within the sub-rectangle that are used in the construction of the 2D LOI.

4.1.3 The LOI in 2 Dimensions

The generalization of the 1D LOI we will consider is constructed by treating each spatial dimension independently to determine the points where solution and derivative values will be interpolated. If we consider a collocation solution of degree p in x and q in y, and the sub-rectangle bounded by $[x_\alpha \leq x \leq x_\omega]$ and $[y_\alpha \leq y \leq y_\omega]$, we have $s_1 = x_\alpha$, $s_2 = x_\omega$, $t_1 = y_\alpha$, and $t_2 = y_\omega$, where solution and derivative values are interpolated. We will also have $p-4$ points, w_i, and $q-4$ points, v_j, which are set relatively across the spatial domain, according to Table 4.2, where only solution values are interpolated.

The LOI does not have any edge cases to consider as it only utilizes solution information from within the sub-rectangle on which the interpolant is being constructed. Equation 4.1 can then be used to obtain the LOI on a given rectangle.

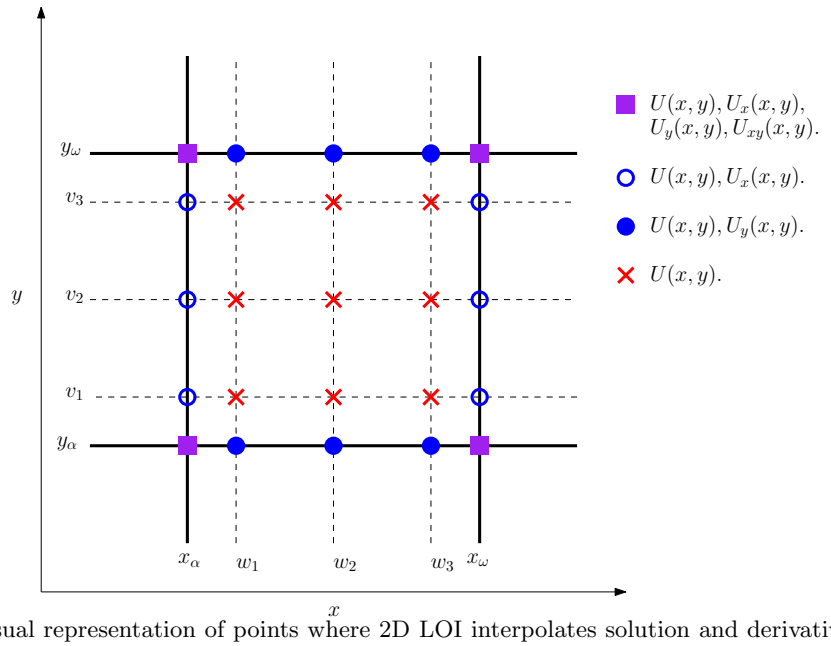

Figure 4.2: Visual representation of points where 2D LOI interpolates solution and derivative values when $p = q = 7$.

4.1.4 The LOI2

The LOI2 is a 2D interpolant that we introduce in this subsection. The LOI2 does not interpolate any spatial derivatives and we will show later in this chapter why this may be an advantage.

We first consider a 1D version of the LOI2. For a 1D collocation solution of degree p, the corresponding 1D version of the LOI2 is constructed using p points; these are the 2 mesh points bounding the subinterval and $p - 2$ internal points. The internal points are chosen so that the L^2-norm, over the spatial domain, between the interpolation error and the known collocation error, as seen in Equation (2.4), is minimized. The points for collocation solutions of degrees 4 through 7 are given in Table 4.3.

Degree of Collocation Solution			
4	5	6	7
0	0	0	0
0.302331973224813	0.179424182143275	0.143465814421734	0.107235231524833
0.697668026790731	0.5	0.37051884018424	0.283676545203967
1	0.820575567392312	0.629481160240031	0.5
	1	0.856534185886017	0.716323434111384
		1	0.892764777407028
			1

Table 4.3: Interpolation points used to construct the LOI2 for collocation solutions of degrees 4 through 7.

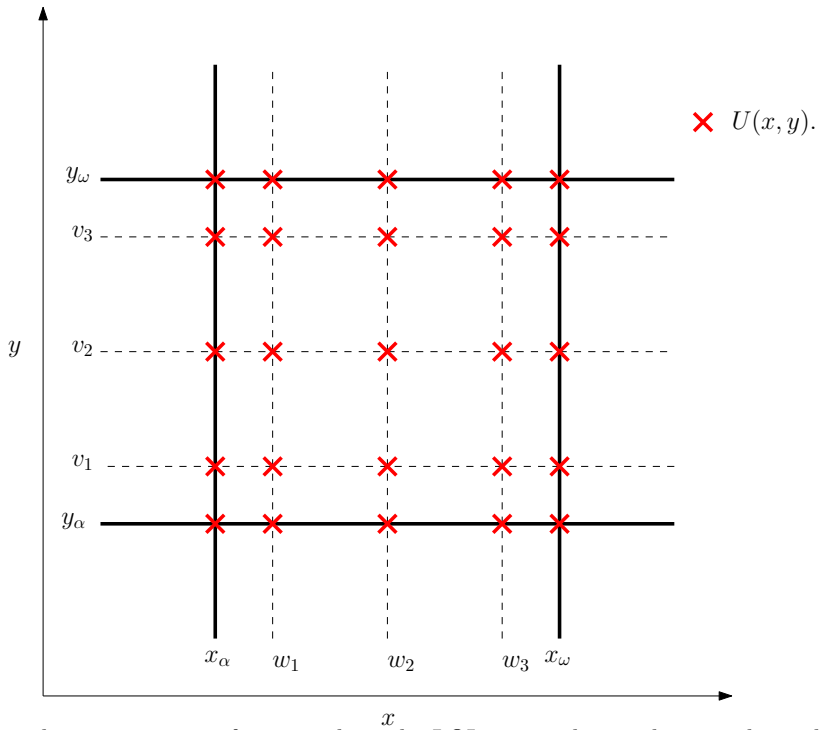

Figure 4.3: Visual representation of points where the LOI2 interpolates solution values when $p = q = 5$.

As this 1D version of the LOI2 does not make use of any spatial derivative values it can be constructed as a Lagrange interpolant. With interpolation points $w_i : i = 1, 2, ..., p$, the 1D version of the LOI2 can be constructed as follows:

$$q(x) = \sum_{i=1}^{p} U(w_i) l_i(x). \tag{4.2}$$

where

$$l_i(x) = \prod_{\substack{k=1 \\ k \neq i}}^{|p|} \frac{(x - w_k)}{(w_i - w_k)}.$$

The 2D LOI2 can be represented using the 2D Hermite Birkhoff interpolant introduced in 4.1.1, choosing the internal points in each dimension appropriately according to the degree of the collocation solution in that dimension (see Table 4.3). However, because the 2D LOI2 does not use any spatial derivative values, we can instead use a simpler 2D Lagrange interpolant form. Given a collocation solution, $U(x, y)$, of degree p in x, and degree q in y, and a sub-rectangle based on the subintervals, $x \in [x_\alpha, x_\omega]$ and $y \in [y_\alpha, y_\omega]$, and the interpolation points, $w_i : i = 1, 2, \ldots, p$,

$v_i : i = 1, 2, \ldots, q$, the 2D LOI2 can be constructed as follows:

$$q\left(x, y\right) = \sum_{i=1}^{p} \sum_{j=1}^{q} l_i\left(x\right) \hat{l}_j\left(y\right) U\left(w_i, v_j\right), \tag{4.3}$$

where

$$l_i(x) = \prod_{\substack{k=1 \\ k \neq i}}^{|p|} \frac{(x - w_k)}{(w_i - w_k)}, \;\; \hat{l}_j(x) = \prod_{\substack{k=1 \\ k \neq j}}^{|q|} \frac{(y - v_k)}{(v_j - v_k)}.$$

4.2 Testing Software

Testing scripts have been written in Scilab version 6.1.0 to implement these interpolants, investigate their convergence rates, and assess the quality of error estimates they can provide. The scripts implement B-spline Gaussian collocation as described in Chapter 2. For the time-dependent case, the Scilab *daskr* ([4], [5]) function is used to solve the DAEs that arise from the collocation process. For the non-time-dependent case, the Scilab nonlinear solver function *fsolve* is used to solve the nonlinear equations that arise from the collocation process. The Mingw compiler module for Scilab is also required as the B-spline software is implemented in Fortran. The testing scripts are available in the appendix.

4.3 Collocation Solution Error and Convergence Results

Collocation solutions to the following test problems were calculated with a range of basis function degrees and number of subintervals dividing the x and y spatial domains. We choose the degree and number of subintervals in each dimension to be the same.

As the Scilab scripts used to calculate these approximate solutions utilize the error control software *daskr* or *fsolve*, an error tolerance must be provided. We consider pure absolute error control. The absolute tolerance, $atol$, is set such that: $atol = 10^{-n}$ where n is the largest integer such that *daskr* or *fsolve* will successfully return with a solution. The values of n used for the tests are provided in Tables 4.4 and 4.10.

The errors of the collocation solutions are calculated as the difference between the collocation solution and the known exact solution. The differences are calculated on a grid of 7 by 7 points on each sub-rectangle of the spatial domain. The grid is constructed as an even spacing of 7 points including the 2 mesh points bounding each subinterval. The maximum of these differences over all the sub-rectangles is defined to be the maximum error of the collocation solution. This selection of a 7 by 7 grid was tested against resolutions of points up to 40 by 40 and it was found that a 7 by 7 grid is sufficient to obtain reliable results.

The convergence rates, labelled "Rate" in the following tables is calculated as follows. The errors of collocation solutions for the same degree but different number of intervals is compared. If $ERROR_2$ and $ERROR_4$ represent the error for a collocation solution of degree p with 2 and 4 intervals, the rate is calculated as $-log_2(ERROR_4/ERROR_2)$.

4.3.1 2D Non-Time-Dependant PDEs

Test Problems

2D, Non-Time-Dependant Test Problem 1:

The PDE is given by:

$$u_{xx}(x,y) + u_{yy}(x,y) = xe^y,$$

with spatial domain:

$$x \in [0,1],\ y \in [0,1],$$

and boundary conditions taken from the true solution:

$$u(x,y) = xe^y.$$

2D, Non-Time-Dependant Test Problem 2:

The PDE is given by:

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

with spatial domain:

$$x \in [0,1],\ y \in [0,1],$$

and boundary conditions taken from the true solution:

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

2D, Non-Time-Dependant Test Problem 3:

The PDE is given by:

$$u_{xx}(x,y) + u_{yy}(x,y) = \cos(x+y) + \cos(x-y),$$

with spatial domain:

$$x \in [0,\pi],\ y \in [0,\pi],$$

and boundary conditions taken from the true solution:

$$u(x,y) = \cos(x)\cos(y).$$

nint	Degree (p)			
	4	5	6	7
	n	n	n	n
Problem 1				
2	14	14	13	14
4	14	14	14	13
8	14	14	14	13
16	14	14	14	13
Problem 2				
2	14	14	13	14
4	14	14	14	13
8	14	14	14	13
16	14	14	14	13
Problem 3				
2	14	14	14	13
4	16	14	13	13
8	14	14	14	14
16	14	14	14	13

Table 4.4: Value of n for each 2D non-time-dependent PDE, p, and *nint* value used to define tolerance, 10^{-n}, provided to *fsolve*.

Convergence Results

nint	Degree (p)							
	4		5		6		7	
	GE	$Rate$	GE	$Rate$	GE	$Rate$	GE	$Rate$
Problem 1								
2	3.96×10^{-6}		8.53×10^{-8}		1.33×10^{-9}		2.15×10^{-11}	
4	1.51×10^{-7}	4.715	1.59×10^{-9}	5.745	1.23×10^{-11}	6.761	9.90×10^{-14}	7.763
8	5.17×10^{-9}	4.864	2.77×10^{-11}	5.842	1.06×10^{-13}	6.859	1.38×10^{-14}	2.847
16	1.69×10^{-10}	4.933	4.62×10^{-13}	5.907	9.57×10^{-14}	0.143	4.75×10^{-14}	-1.787
Problem 2								
2	3.91×10^{-6}		8.25×10^{-8}		9.85×10^{-10}		1.80×10^{-11}	
4	2.00×10^{-7}	4.293	1.89×10^{-9}	5.45	1.28×10^{-11}	6.261	7.99×10^{-14}	7.816
8	8.41×10^{-9}	4.569	3.71×10^{-11}	5.668	1.50×10^{-13}	6.418	1.89×10^{-14}	2.082
16	3.05×10^{-10}	4.784	7.54×10^{-13}	5.622	1.31×10^{-13}	0.201	7.06×10^{-14}	-1.903
Problem 3								
2	6.67×10^{-4}		4.68×10^{-5}		2.38×10^{-6}		1.08×10^{-7}	
4	2.00×10^{-5}	5.056	1.19×10^{-6}	5.301	1.73×10^{-8}	7.103	7.25×10^{-10}	7.216
8	6.40×10^{-7}	4.968	2.12×10^{-8}	5.811	1.37×10^{-10}	6.983	3.20×10^{-12}	7.824
16	2.01×10^{-8}	4.996	3.41×10^{-10}	5.954	1.04×10^{-12}	7.035	1.53×10^{-14}	7.707

Table 4.5: Global error (GE) and convergence rate ($Rate$) of collocation solution for 2D, non time-dependent problems, $p = 4, 5, 6, 7$, and $nint = 2, 4, 8, 16$.

Table 4.5 shows the errors and corresponding observed convergence rate for the calculated collocation solutions. GE is the global maximum of the difference between the collocation solution

	Degree (p)							
	4		5		6		7	
nint	*Error*	*Rate*	*Error*	*Rate*	*Error*	*Rate*	*Error*	*Rate*
Problem 1								
2	3.48×10^{-7}		2.50×10^{-8}		7.23×10^{-11}		4.13×10^{-12}	
4	6.73×10^{-9}	5.691	5.01×10^{-10}	5.639	3.56×10^{-13}	7.667	2.04×10^{-14}	7.659
8	1.18×10^{-10}	5.828	8.85×10^{-12}	5.823	1.62×10^{-14}	4.456	1.33×10^{-14}	0.617
16	1.97×10^{-12}	5.913	1.47×10^{-13}	5.912	9.53×10^{-14}	-2.555	4.71×10^{-14}	-1.821
Problem 2								
2	3.48×10^{-7}		2.50×10^{-8}		7.23×10^{-11}		4.13×10^{-12}	
4	6.73×10^{-9}	5.691	5.01×10^{-10}	5.639	3.56×10^{-13}	7.667	2.04×10^{-14}	7.659
8	1.18×10^{-10}	5.828	8.85×10^{-12}	5.823	2.13×10^{-14}	4.061	1.87×10^{-14}	0.131
16	1.97×10^{-12}	5.913	1.47×10^{-13}	5.912	1.30×10^{-13}	-2.61	6.99×10^{-14}	-1.907
Problem 3								
2	2.20×10^{-17}		4.51×10^{-17}		2.78×10^{-17}		1.96×10^{-17}	
4	2.16×10^{-6}	-36.51	1.47×10^{-7}	-31.6	1.11×10^{-9}	-25.25	5.98×10^{-11}	-21.54
8	4.37×10^{-8}	5.627	3.21×10^{-9}	5.517	5.70×10^{-12}	7.604	3.24×10^{-13}	7.527
16	7.24×10^{-10}	5.916	5.41×10^{-11}	5.89	2.40×10^{-14}	7.893	1.12×10^{-14}	4.853

Table 4.6: Maximum error (*Error*) and convergence rate (*Rate*) of collocation solution values at mesh points for 2D, non time-dependent problems, $p = 4, 5, 6, 7$, and $nint = 2, 4, 8, 16$.

	Degree (p)							
	4		5		6		7	
nint	*Error*	*Rate*	*Error*	*Rate*	*Error*	*Rate*	*Error*	*Rate*
Problem 1								
2	2.84×10^{-6}		2.98×10^{-7}		1.65×10^{-9}		1.28×10^{-10}	
4	1.04×10^{-7}	4.774	1.23×10^{-8}	4.599	1.60×10^{-11}	6.693	1.31×10^{-12}	6.616
8	3.62×10^{-9}	4.84	4.36×10^{-10}	4.819	6.04×10^{-14}	8.047	1.81×10^{-13}	2.855
16	1.20×10^{-10}	4.919	1.45×10^{-11}	4.906	6.52×10^{-13}	-3.433	3.13×10^{-13}	-0.791
Problem 2								
2	5.83×10^{-5}		1.41×10^{-6}		2.71×10^{-8}		4.66×10^{-10}	
4	4.02×10^{-6}	3.861	4.94×10^{-8}	4.84	4.70×10^{-10}	5.848	3.98×10^{-12}	6.869
8	2.63×10^{-7}	3.93	1.63×10^{-9}	4.919	7.75×10^{-12}	5.923	1.71×10^{-13}	4.546
16	1.69×10^{-8}	3.965	5.25×10^{-11}	4.956	7.30×10^{-13}	3.408	3.38×10^{-13}	-0.987
Problem 3								
2	1.95×10^{-3}		1.45×10^{-4}		8.95×10^{-6}		4.76×10^{-7}	
4	1.52×10^{-4}	3.687	5.75×10^{-6}	4.654	1.74×10^{-7}	5.68	4.70×10^{-9}	6.661
8	9.80×10^{-6}	3.953	1.91×10^{-7}	4.915	2.85×10^{-9}	5.937	3.89×10^{-11}	6.919
16	6.18×10^{-7}	3.988	6.04×10^{-9}	4.979	4.50×10^{-11}	5.985	3.69×10^{-13}	6.717

Table 4.7: Maximum error (*Error*) and convergence rate (*Rate*) of x spatial derivative of collocation solution values at mesh points for 2D, non time-dependent problems, $p = 4, 5, 6, 7$, and $nint = 2, 4, 8, 16$.

nint	Degree (p)							
	4		5		6		7	
	Error	*Rate*	*Error*	*Rate*	*Error*	*Rate*	*Error*	*Rate*
Problem 1								
2	5.83×10^{-5}		1.41×10^{-6}		2.71×10^{-8}		4.66×10^{-10}	
4	4.02×10^{-6}	3.861	4.94×10^{-8}	4.84	4.70×10^{-10}	5.848	4.05×10^{-12}	6.843
8	2.63×10^{-7}	3.93	1.63×10^{-9}	4.919	7.75×10^{-12}	5.923	1.60×10^{-13}	4.664
16	1.69×10^{-8}	3.965	5.24×10^{-11}	4.96	6.15×10^{-13}	3.656	2.84×10^{-13}	-0.827
Problem 2								
2	5.83×10^{-5}		1.41×10^{-6}		2.71×10^{-8}		4.66×10^{-10}	
4	4.02×10^{-6}	3.861	4.94×10^{-8}	4.84	4.70×10^{-10}	5.848	4.05×10^{-12}	6.843
8	2.63×10^{-7}	3.93	1.63×10^{-9}	4.919	7.75×10^{-12}	5.923	1.60×10^{-13}	4.664
16	1.69×10^{-8}	3.965	5.24×10^{-11}	4.96	7.62×10^{-13}	3.346	3.52×10^{-13}	-1.139
Problem 3								
2	1.95×10^{-3}		1.45×10^{-4}		8.95×10^{-6}		4.76×10^{-7}	
4	1.52×10^{-4}	3.687	5.75×10^{-6}	4.654	1.74×10^{-7}	5.68	4.70×10^{-9}	6.661
8	9.80×10^{-6}	3.953	1.91×10^{-7}	4.915	2.85×10^{-9}	5.937	3.89×10^{-11}	6.919
16	6.18×10^{-7}	3.988	6.04×10^{-9}	4.979	4.50×10^{-11}	5.985	3.41×10^{-13}	6.832

Table 4.8: Maximum error (*Error*) and convergence rate (*Rate*) of y spatial derivative of collocation solution values at mesh points for 2D, non time-dependent problems, $p = 4, 5, 6, 7$, and $nint = 2, 4, 8, 16$.

nint	Degree (p)							
	4		5		6		7	
	Error	*Rate*	*Error*	*Rate*	*Error*	*Rate*	*Error*	*Rate*
Problem 1								
2	8.15×10^{-4}		3.54×10^{-5}		1.03×10^{-6}		2.54×10^{-8}	
4	1.13×10^{-4}	2.849	2.49×10^{-6}	3.828	3.59×10^{-8}	4.845	4.42×10^{-10}	5.847
8	1.49×10^{-5}	2.925	1.66×10^{-7}	3.913	1.18×10^{-9}	4.924	3.55×10^{-11}	3.636
16	1.91×10^{-6}	2.963	1.07×10^{-8}	3.958	7.86×10^{-11}	3.912	1.72×10^{-10}	-2.275
Problem 2								
2	1.79×10^{-3}		7.37×10^{-5}		2.11×10^{-6}		5.16×10^{-8}	
4	2.32×10^{-4}	2.941	5.04×10^{-6}	3.87	7.23×10^{-8}	4.869	8.85×10^{-10}	5.865
8	3.00×10^{-5}	2.954	3.32×10^{-7}	3.925	2.38×10^{-9}	4.925	3.20×10^{-11}	4.791
16	3.83×10^{-6}	2.971	2.14×10^{-8}	3.958	9.64×10^{-11}	4.625	8.55×10^{-11}	-1.42
Problem 3								
2	1.44×10^{-2}		2.20×10^{-3}		1.83×10^{-4}		1.63×10^{-5}	
4	1.36×10^{-3}	3.41	1.84×10^{-4}	3.585	4.25×10^{-6}	5.427	3.27×10^{-7}	5.637
8	1.77×10^{-4}	2.944	1.23×10^{-5}	3.898	1.39×10^{-7}	4.937	5.43×10^{-9}	5.913
16	2.23×10^{-5}	2.986	7.84×10^{-7}	3.975	4.38×10^{-9}	4.985	1.03×10^{-10}	5.725

Table 4.9: Maximum error (*Error*) and convergence rate (*Rate*) of xy spatial derivative of collocation solution values at mesh points for 2D, non time-dependent problems, $p = 4, 5, 6, 7$, and $nint = 2, 4, 8, 16$.

and the known true solution. For degrees 4 through 6, the experimental convergence rate is close to the expected rate of $p+1$, with the larger values of *nint* corresponding to better agreement with the expected convergence rates. For degree 7, the errors begin to reach the practical minimum of about 10^{-14}, where the accuracy limitations of *fsolve* interfere with the accuracy of the collocation solution; however, there are a few entries that approximately agree with the expected rate. These accuracy limitations arise because *fsolve* is solving for the B-spline coefficients to within its specified tolerance, however we are analysing the solution and its derivative values which will not have the same errors as the coefficients.

Table 4.6 shows the maximum error of the collocation solution at each point where both the x and y values are mesh points. Tables 4.7, 4.8, and 4.9 provide results that assesses the error of the x spatial derivatives, the y spatial derivatives, and the x, y cross derivatives, of the collocation solution.

In Table 4.6, when $p = 4$, the experimentally observed convergence rates agree with the expected rate of $2(p-1)$. However, when $p = 5$, the convergence rate is the same as the rate for $p = 4$, which means that this case does not agree with the expected convergence rate. For $p = 6$, the magnitude of the errors becomes small enough that the accuracy of the results from *fsolve* becomes an issue. However, from the few results which are not affected by the accuracy of *fsolve*, the error seems to converge at a rate close to 8, which is well below the expected rate of 10. The results for $p = 7$ are also affected by the accuracy of the results from *fsolve* with the few reasonable results showing a convergence rate of about 8, far below the expected rate of 12.

In Table 4.7 we expect to see a convergence rate of $2(p-1)$ for the derivative values. For $p = 4$, the results for the first test problem show a convergence rate of 5, which is below the expected rate of 6. Test problems 2 and 3 show a convergence rate of 4, which is also below the expected rate of 6. Also, the error of the derivative is larger than the corresponding error of the collocation solution (see Table 4.5). For $p = 5$, the results are consistent across the three test problems; however, the convergence rate is 5, which is far below the expected rate of 8. The errors of the derivatives are larger than the errors of the collocation solutions, as seen in Table 4.5.

For $p = 6$ and $p = 7$ in Table 4.7, the errors begin to be impacted by the accuracy limitations of *fsolve*, and this impacts the convergence rates. There are however a few entries not impacted by this issue. When $p = 6$ the observed convergence rate is about 6, and when $p = 7$, the observed convergence rate is about 7. These are both substantially below their expected convergence rates of 10 and 12. These errors are also larger than the corresponding errors of the collocation solutions as seen in Table 4.5.

In Table 4.8 we expect to see similar results to those presented in Table 4.7. However, since the test problems are not all symmetric in x and y, there are some differences. When $p = 6$ and $p = 7$, there is much less impact from the accuracy limitations of *fsolve* compared to the results for the x derivative. All of the values of p give derivatives whose errors converge at a rate of p, which is well below their expected rates of $2(p - 1)$. Also, the errors of the derivatives are larger than the corresponding collocation solution errors given in Table 4.5.

Examining Tables 4.7 and 4.8 together, we can see that the x and y derivative values at the mesh points have a convergence rate of p, and the errors of these derivatives are larger than the corresponding collocation solution errors.

In Table 4.9 the errors of the cross derivatives converge at a rate of $p - 1$. There is minimal interference from the accuracy limitations of *fsolve* because the errors are even larger than the errors of the first derivatives which we noted earlier were larger than the corresponding collocation solutions errors.

4.3.2 2D Time-Dependant PDEs

Test Problems

2D time-dependent PDE Test Problem 1:

The PDE is given by:

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

$$\epsilon = \tfrac{1}{10},$$

the spatial domain is:

$$x \in [0, 2],\ y \in [0, 2],$$

and the initial and boundary conditions are taken from the true solution:

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

The time integration is from $t = 0$ to $t = 1$.

2D time-dependent PDE Test Problem 2:

The PDE given by:

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

$$\epsilon = \tfrac{1}{10},$$

the spatial domain is:

$$x \in [0, 1],\ y \in [0, 1],$$

and the initial and boundary conditions are taken from the true solution:

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

The time integration is from $t = 0$ to $t = 1$.

Convergence Results

Table 4.11 shows the global error and convergence rate of the collocation solutions for the 2D time-dependent PDE test problems. The convergence rate for the global error is expected to be of order $p + 1$. When $p = 4$, the experimental convergence rate closely agrees with the expected rate. For $p = 5$ the experimental convergence rates for *nint* from 2 to 8 do agree with their expected rates, although not as closely as for $p = 4$.

For $p = 6$ in Table 4.11, the accuracy limitations of *daskr* significantly impact the observed convergence rates. These accuracy limitations arise because *daskr* is solving for the B-spline coefficients to within its specified tolerance, however we are analysing the solution and derivative values

nint	Degree (p)			
	4	5	6	7
	n	n	n	n
Problem 1				
2	10	10	10	10
4	11	11	11	11
8	11	11	11	10
16	11	11	9	4
Problem 2				
2	11	10	9	5
4	14	9	8	8
8	11	9	8	5
16	13	8	3	1

Table 4.10: Value of n for each 2D time-dependent PDE, p, and *nint* value used to define tolerance, 10^{-n}, provided to *daskr*.

nint	Degree (p)							
	4		5		6		7	
	GE	$Rate$	GE	$Rate$	GE	$Rate$	GE	$Rate$
Problem 1								
2	3.73×10^{-4}		2.86×10^{-5}		1.52×10^{-6}		6.55×10^{-8}	
4	1.24×10^{-5}	4.909	7.60×10^{-7}	5.233	1.09×10^{-8}	7.12	1.21×10^{-9}	5.76
8	3.93×10^{-7}	4.983	1.38×10^{-8}	5.778	8.03×10^{-10}	3.763	3.30×10^{-9}	-1.451
16	1.23×10^{-8}	4.995	9.62×10^{-10}	3.847	4.08×10^{-9}	-2.342	1.63×10^{-4}	-15.59
Problem 2								
2	2.04×10^{-3}		3.19×10^{-4}		3.91×10^{-5}		4.19×10^{-4}	
4	9.82×10^{-5}	4.38	6.49×10^{-6}	5.62	6.72×10^{-7}	5.865	3.88×10^{-8}	13.4
8	3.36×10^{-6}	4.868	1.48×10^{-7}	5.457	1.22×10^{-8}	5.78	4.29×10^{-6}	-6.788
16	1.06×10^{-7}	4.984	1.72×10^{-8}	3.105	1.70×10^{-3}	-17.08	$0.00 \times 10^{+0}$	Inf

Table 4.11: Global error (GE) and convergence rate ($Rate$) of collocation solution for 2D, time-dependent problems, $p = 4, 5, 6, 7$, and $nint = 2, 4, 8, 16$.

which will not have the same errors as the coefficients. For $p = 6$, the measured convergence rate of the error for test problem 1, for *nint* from 2 to 4, does agree with the expected convergence rate. However, for larger values of *nint* there is no clear convergence rate. For test problem 2 and $p = 6$, the errors converge at an order of about p although this could be caused by the collocation solution only being calculated to a tolerance of 10^{-8}, as indicated in Table 4.10.

For $p = 7$ in Table 4.11, no clear convergence rates are observed. For test problem 1 we quickly run into issues associated with the accuracy of the results from *daskr*. For test problem 2 the errors are limited by the tolerance with which the collocation solutions were calculated, so we would not expect to see any obvious convergence rate.

	Degree (p)							
	4		5		6		7	
nint	*Error*	*Rate*	*Error*	*Rate*	*Error*	*Rate*	*Error*	*Rate*
Problem 1								
2	9.91×10^{-6}		4.11×10^{-7}		2.28×10^{-8}		3.46×10^{-10}	
4	3.22×10^{-7}	4.943	6.50×10^{-9}	5.984	9.36×10^{-11}	7.929	8.98×10^{-10}	-1.374
8	4.99×10^{-9}	6.012	4.77×10^{-10}	3.768	7.88×10^{-10}	-3.073	3.30×10^{-9}	-1.88
16	3.30×10^{-10}	3.92	7.54×10^{-10}	-0.661	4.08×10^{-9}	-2.371	1.63×10^{-4}	-15.59
Problem 2								
2	1.44×10^{-4}		6.95×10^{-5}		3.40×10^{-6}		2.38×10^{-4}	
4	1.83×10^{-5}	2.971	7.84×10^{-7}	6.471	1.98×10^{-8}	7.422	6.41×10^{-9}	15.18
8	4.09×10^{-7}	5.486	1.85×10^{-8}	5.401	6.91×10^{-9}	1.522	4.13×10^{-6}	-9.329
16	6.79×10^{-9}	5.911	1.49×10^{-8}	0.316	1.61×10^{-3}	-17.83	$0.00 \times 10^{+0}$	Inf

Table 4.12: Maximum error (*Error*) and convergence rate (*Rate*) of collocation solution values at mesh points for 2D, time-dependent problems, for $p = 4, 5, 6, 7$, and $nint = 2, 4, 8, 16$.

Table 4.12 shows the error of the solution value from the collocation solutions at the mesh points. The expected convergence rate is of order $2(p - 1)$. Most of the results in this table are impacted by the accuracy limitations of *daskr* as the values of the measured errors quickly approach 10^{-10}. However, for $p = 4$, the experimental convergence rate agrees with the expected rate of 6. For $p = 5$ and test problem 1, we see an error close to 10^{-10} for $nint = 4$; however, for the collocation solutions with $nint = 4$ and $nint = 5$ we see an experimental convergence rate of about 6, below the expected rate of 8. Test problem 2 has similar results for $p = 5$, although the error only reaches a minimum value of about 10^{-8}.

For $p = 6$ and $p = 7$ in Table 4.12, no clear convergence rates are evident as the majority of the

errors are impacted by the accuracy limitations of *daskr*. However, if the values of the errors are compared to the global error recorded in Table 4.11, we can see that the solution values at these points are more accurate. There are, however, a few exceptions when p and $nint$ are large (e.g., test problem 1, $p = 7$, and $nint = 8$). The collocation solutions with $p = 4$ and $p = 5$ have a consistently smaller error at the mesh points than the corresponding global error from the entire spatial domain as observed in Table 4.11.

	Degree (p)							
	4		5		6		7	
nint	*Error*	*Rate*	*Error*	*Rate*	*Error*	*Rate*	*Error*	*Rate*
Problem 1								
2	5.06×10^{-5}		3.32×10^{-6}		3.66×10^{-8}		1.51×10^{-9}	
4	1.88×10^{-6}	4.747	5.38×10^{-9}	9.268	1.38×10^{-10}	8.053	1.41×10^{-9}	0.095
8	2.76×10^{-8}	6.092	7.64×10^{-10}	2.816	1.24×10^{-9}	-3.168	5.19×10^{-9}	-1.88
16	1.06×10^{-9}	4.7	1.18×10^{-9}	-0.633	6.40×10^{-9}	-2.371	2.56×10^{-4}	-15.59
Problem 2								
2	6.94×10^{-3}		2.14×10^{-3}		3.94×10^{-4}		2.15×10^{-3}	
4	6.58×10^{-4}	3.399	1.12×10^{-4}	4.251	2.92×10^{-6}	7.078	1.37×10^{-6}	10.61
8	6.96×10^{-5}	3.239	2.55×10^{-6}	5.461	1.85×10^{-7}	3.983	1.10×10^{-4}	-6.326
16	4.82×10^{-6}	3.853	1.58×10^{-7}	4.014	1.76×10^{-2}	-16.54	$0.00 \times 10^{+0}$	Inf

Table 4.13: Maximum error (*Error*) and convergence rate (*Rate*) of x spatial derivative of collocation solution values at mesh points for 2D, time-dependent problems, $p = 4, 5, 6, 7$, and $nint = 2, 4, 8, 16$.

	Degree (p)							
	4		5		6		7	
nint	*Error*	*Rate*	*Error*	*Rate*	*Error*	*Rate*	*Error*	*Rate*
Problem 1								
2	5.06×10^{-5}		3.32×10^{-6}		3.66×10^{-8}		1.51×10^{-9}	
4	1.88×10^{-6}	4.747	5.38×10^{-9}	9.268	1.38×10^{-10}	8.053	1.41×10^{-9}	0.095
8	2.76×10^{-8}	6.092	7.64×10^{-10}	2.816	1.24×10^{-9}	-3.168	5.19×10^{-9}	-1.88
16	1.06×10^{-9}	4.7	1.18×10^{-9}	-0.633	6.40×10^{-9}	-2.371	2.56×10^{-4}	-15.59
Problem 2								
2	6.94×10^{-3}		2.14×10^{-3}		3.94×10^{-4}		2.15×10^{-3}	
4	6.58×10^{-4}	3.399	1.12×10^{-4}	4.251	2.92×10^{-6}	7.078	1.37×10^{-6}	10.61
8	6.96×10^{-5}	3.239	2.55×10^{-6}	5.461	1.85×10^{-7}	3.983	1.10×10^{-4}	-6.326
16	4.82×10^{-6}	3.853	1.58×10^{-7}	4.01	1.76×10^{-2}	-16.54	$0.00 \times 10^{+0}$	Inf

Table 4.14: Maximum error (*Error*) and convergence rate (*Rate*) of y spatial derivative of collocation solution values at mesh points for 2D, time-dependent problems, $p = 4, 5, 6, 7$, and $nint = 2, 4, 8, 16$.

In Tables 4.13 and 4.14, the expected convergence rate of the spatial derivative is of order $2(p-1)$. As these test problems are symmetric in the x and y dimensions, the two tables have the same results. For test problem 1, most of the errors are sufficiently small that they are greatly impacted by the

accuracy limitations of *daskr*, although for $p = 4$, we see a convergence rate between 5 and 6.

For test problem 2, in Tables 4.13 and 4.14, there are fewer results impacted by the accuracy limitations of *daskr*. When $p = 4$, the errors converge at a rate of 4, while for $p = 5$ the error converges at a rate between 4 and 5. For $p = 6$ and $p = 7$, no clear convergence is rate observed although this is likely due to the solutions being calculated with a relatively coarse tolerance, as seen in Table 4.10.

Comparing the magnitude of the spatial derivative errors in Tables 4.13 and 4.14 to the global errors from Table 4.11, we see that these solution derivative values are more accurate than the solution values of an arbitrary point. Comparing the spatial derivative errors at the mesh points to the solution values at those points, we see that the errors of the solution values are consistently one order of magnitude smaller than the error of the spatial derivatives (e.g., for problem 1, $p = 4$, $nint = 2$, the spatial derivatives have an error that is 5.06×10^{-5} while the solution values have an error that is 9.91×10^{-6}).

	Degree (p)							
	4		5		6		7	
nint	*Error*	*Rate*	*Error*	*Rate*	*Error*	*Rate*	*Error*	*Rate*
Problem 1								
2	1.83×10^{-4}		9.41×10^{-6}		1.71×10^{-7}		3.88×10^{-9}	
4	6.72×10^{-6}	4.772	8.81×10^{-10}	13.38	2.01×10^{-10}	9.731	2.22×10^{-9}	0.808
8	9.91×10^{-8}	6.082	1.22×10^{-9}	-0.475	1.94×10^{-9}	-3.271	8.15×10^{-9}	-1.88
16	2.53×10^{-9}	5.295	1.86×10^{-9}	-0.604	1.01×10^{-8}	-2.371	4.02×10^{-4}	-15.59
Problem 2								
2	5.78×10^{-2}		1.72×10^{-2}		7.07×10^{-3}		4.70×10^{-3}	
4	9.93×10^{-3}	2.542	3.56×10^{-3}	2.271	1.59×10^{-4}	5.477	9.27×10^{-5}	5.664
8	1.75×10^{-3}	2.501	1.60×10^{-4}	4.476	1.77×10^{-5}	3.163	9.05×10^{-3}	-6.61
16	4.54×10^{-4}	1.952	1.23×10^{-5}	3.7	3.51×10^{-1}	-14.27	$0.00 \times 10^{+0}$	Inf

Table 4.15: Maximum error (*Error*) and convergence rate (*Rate*) of xy spatial derivative of collocation solution values at mesh points for 2D, time-dependent problems, $p = 4, 5, 6, 7$, and $nint = 2, 4, 8, 16$.

In Table 4.15 we expect a convergence rate of order $2(p - 1)$ for the xy cross derivative at the mesh points; however this rate is not observed in these experiments. While the values of the errors in this table are relatively large, they are likely still impacted by the accuracy limitations of *daskr* as the solutions from which these derivatives are calculated are affected by the accuracy limitations of *daskr*. Furthermore, we see the trend of the magnitude of the error increasing as the order of the

derivative increases, similar to the x and y spatial derivatives compared to the solution values at the mesh points.

Comparing the errors in Table 4.15 to the corresponding errors in Table 4.11, we see that the errors of the xy cross derivatives are larger than the global error for most combinations of p and $nint$.

4.3.3 Collocation Convergence Results Discussion

In the time-dependent and non-time-dependent PDE cases, we see that the global error of the collocation solution agrees with its theoretically expected convergence rate, suggesting that the test collocation software is performing correctly.

However, when we consider the error of the solution values at mesh points, there are some cases where the expected convergence rate is met, although, in the majority of cases it is not. However, across the entire spatial domain, the error at these points is smaller than the global error, indicating that there is higher accuracy at these points.

The errors of the x and y spatial derivatives at the mesh points also do not meet their expected convergence rates. For the non-time-dependent case, the magnitude of the errors is the same or larger than the global solution value errors across the spatial domain. However, for the time-dependent case, the errors are smaller than the global error of the solution values.

The xy cross derivatives also do not meet the expected convergence rate. Looking at the magnitude of these errors, we see that for both the time-dependent and non-time-dependent cases the errors of the xy cross derivative are larger than the global error of the solution values.

4.4 Interpolant Error and Convergence Results

In this section, we examine the errors and convergence rates for the interpolants described earlier in this chapter. These results are calculated using the same method as was used for the collocation solution in section 4.3. The expected order of convergence for the LOI and LOI2 is p while the expected order of convergence for the SCI is $p + 2$. These expected convergence rates for the SCI

and LOI are based on assuming that the 2D versions presented in this thesis converge at the same rate as their original 1D versions.

4.4.1 2D Non-Time-Dependant PDEs

	Degree (p)							
	4		5		6		7	
nint	GE	$Rate$	GE	$Rate$	GE	$Rate$	GE	$Rate$
Problem 1								
2	4.95×10^{-6}		9.04×10^{-8}		1.98×10^{-9}		2.25×10^{-11}	
4	1.91×10^{-7}	4.694	1.65×10^{-9}	5.777	1.83×10^{-11}	6.754	9.90×10^{-14}	7.83
8	6.66×10^{-9}	4.842	2.78×10^{-11}	5.89	1.59×10^{-13}	6.849	1.38×10^{-14}	2.847
16	2.20×10^{-10}	4.922	4.65×10^{-13}	5.903	9.55×10^{-14}	0.736	4.71×10^{-14}	-1.774
Problem 2								
2	6.51×10^{-6}		1.12×10^{-7}		1.98×10^{-9}		2.25×10^{-11}	
4	2.99×10^{-7}	4.445	2.59×10^{-9}	5.436	1.97×10^{-11}	6.652	1.08×10^{-13}	7.7
8	1.14×10^{-8}	4.714	4.93×10^{-11}	5.717	2.33×10^{-13}	6.399	1.93×10^{-14}	2.488
16	3.94×10^{-10}	4.855	8.47×10^{-13}	5.863	1.31×10^{-13}	0.837	7.06×10^{-14}	-1.87
Problem 3								
2	1.08×10^{-3}		6.35×10^{-5}		3.85×10^{-6}		1.14×10^{-7}	
4	2.15×10^{-5}	5.654	1.26×10^{-6}	5.651	1.91×10^{-8}	7.653	7.25×10^{-10}	7.301
8	6.46×10^{-7}	5.059	2.12×10^{-8}	5.902	1.30×10^{-10}	7.197	3.20×10^{-12}	7.824
16	2.03×10^{-8}	4.993	3.41×10^{-10}	5.954	9.88×10^{-13}	7.041	1.61×10^{-14}	7.635

Table 4.16: Global error (GE) and convergence rate ($Rate$) of SCI for 2D, non time-dependent problems, $p = 4, 5, 6, 7$, and $nint = 2, 4, 8, 16$.

Table 4.16 presents the errors and convergence rates of the SCI. The magnitudes of the errors are very similar to that of the collocation solution with the errors of the SCI being marginally larger. As such, the errors of the SCI converge at a rate of $p + 1$, which is below the expected rate of $p + 2$. However, this lower rate of convergence of the SCI is expected as the solution and derivative values of the collocation solution which are interpolated by the SCI do not satisfy the expected convergence rates, as seen in section 4.3.

Table 4.17 presents the errors and convergence rates of the LOI. We observe that the LOI converges at its expected rate of p. In Table 4.18, the errors and convergence rates of the LOI2 are presented. The LOI2, like the LOI, converges at its expected rate of p. However, the errors of the LOI tend to be about one order of magnitude larger than those of the LOI2; this could be caused by the fact that the LOI interpolates several less accurate derivative values.

nint	Degree (p)							
	4		5		6		7	
	GE	$Rate$	GE	$Rate$	GE	$Rate$	GE	$Rate$
Problem 1								
2	3.45×10^{-4}		4.54×10^{-6}		1.03×10^{-7}		1.51×10^{-9}	
4	2.44×10^{-5}	3.822	1.61×10^{-7}	4.821	1.82×10^{-9}	5.822	1.33×10^{-11}	6.826
8	1.62×10^{-6}	3.91	5.34×10^{-9}	4.91	3.02×10^{-11}	5.91	1.11×10^{-13}	6.913
16	1.05×10^{-7}	3.955	1.72×10^{-10}	4.955	4.88×10^{-13}	5.95	4.71×10^{-14}	1.232
Problem 2								
2	3.45×10^{-4}		4.54×10^{-6}		1.03×10^{-7}		1.51×10^{-9}	
4	2.63×10^{-5}	3.717	2.02×10^{-7}	4.491	1.82×10^{-9}	5.822	1.35×10^{-11}	6.81
8	2.37×10^{-6}	3.472	8.31×10^{-9}	4.602	3.87×10^{-11}	5.554	1.58×10^{-13}	6.418
16	1.79×10^{-7}	3.725	3.04×10^{-10}	4.773	7.77×10^{-13}	5.637	7.02×10^{-14}	1.168
Problem 3								
2	1.57×10^{-2}		7.86×10^{-4}		4.54×10^{-5}		2.51×10^{-6}	
4	1.67×10^{-3}	3.229	2.22×10^{-5}	5.148	1.23×10^{-6}	5.201	1.79×10^{-8}	7.133
8	1.19×10^{-4}	3.814	6.79×10^{-7}	5.03	2.18×10^{-8}	5.821	1.37×10^{-10}	7.022
16	7.66×10^{-6}	3.954	2.07×10^{-8}	5.037	3.52×10^{-10}	5.956	1.05×10^{-12}	7.031

Table 4.17: Global error (GE) and convergence rate ($Rate$) of LOI for 2D, non time-dependent problems, $p = 4, 5, 6, 7$, and $nint = 2, 4, 8, 16$.

nint	Degree (p)							
	4		5		6		7	
	GE	$Rate$	GE	$Rate$	GE	$Rate$	GE	$Rate$
Problem 1								
2	5.63×10^{-5}		1.53×10^{-6}		2.45×10^{-8}		4.23×10^{-10}	
4	3.94×10^{-6}	3.836	5.42×10^{-8}	4.822	4.34×10^{-10}	5.822	3.74×10^{-12}	6.822
8	2.61×10^{-7}	3.917	1.80×10^{-9}	4.91	7.21×10^{-12}	5.91	3.20×10^{-14}	6.868
16	1.68×10^{-8}	3.958	5.81×10^{-11}	4.955	1.21×10^{-13}	5.896	4.75×10^{-14}	-0.572
Problem 2								
2	7.03×10^{-5}		1.53×10^{-6}		2.45×10^{-8}		4.29×10^{-10}	
4	6.21×10^{-6}	3.5	6.51×10^{-8}	4.558	4.34×10^{-10}	5.822	5.33×10^{-12}	6.331
8	4.63×10^{-7}	3.746	2.80×10^{-9}	4.539	9.23×10^{-12}	5.554	5.33×10^{-14}	6.644
16	3.16×10^{-8}	3.872	1.02×10^{-10}	4.771	1.81×10^{-13}	5.671	7.13×10^{-14}	-0.42
Problem 3								
2	2.89×10^{-3}		2.68×10^{-4}		1.08×10^{-5}		8.22×10^{-7}	
4	2.80×10^{-4}	3.366	7.53×10^{-6}	5.155	2.94×10^{-7}	5.193	5.43×10^{-9}	7.242
8	1.92×10^{-5}	3.865	2.30×10^{-7}	5.032	5.21×10^{-9}	5.819	4.10×10^{-11}	7.047
16	1.23×10^{-6}	3.967	7.00×10^{-9}	5.04	8.39×10^{-11}	5.956	3.17×10^{-13}	7.018

Table 4.18: Global error (GE) and convergence rate ($Rate$) of LOI2 for 2D, non time-dependent problems, $p = 4, 5, 6, 7$, and $nint = 2, 4, 8, 16$.

4.4.2 2D Time-Dependant PDEs

	Degree (p)							
	4		5		6		7	
nint	GE	$Rate$	GE	$Rate$	GE	$Rate$	GE	$Rate$
Problem 1								
2	4.06×10^{-4}		2.82×10^{-5}		1.44×10^{-6}		8.10×10^{-8}	
4	1.49×10^{-5}	4.769	7.60×10^{-7}	5.212	1.41×10^{-8}	6.673	1.21×10^{-9}	6.067
8	5.00×10^{-7}	4.897	1.38×10^{-8}	5.778	8.03×10^{-10}	4.133	3.30×10^{-9}	-1.451
16	1.59×10^{-8}	4.973	9.62×10^{-10}	3.847	4.08×10^{-9}	-2.343	1.63×10^{-4}	-15.59
Problem 2								
2	2.33×10^{-3}		3.78×10^{-4}		6.50×10^{-5}		4.19×10^{-4}	
4	9.71×10^{-5}	4.586	6.42×10^{-6}	5.88	9.03×10^{-7}	6.17	5.62×10^{-8}	12.86
8	3.50×10^{-6}	4.795	1.48×10^{-7}	5.442	1.20×10^{-8}	6.236	4.29×10^{-6}	-6.255
16	1.17×10^{-7}	4.902	1.72×10^{-8}	3.105	1.70×10^{-3}	-17.11	$0.00 \times 10^{+0}$	Inf

Table 4.19: Global error (GE) and convergence rate ($Rate$) of SCI for 2D, time-dependent problems, $p = 4, 5, 6, 7$, and $nint = 2, 4, 8, 16$.

Table 4.19 presents the errors and convergence rates of the SCI. The results for larger values of p and $nint$ become affected by the accuracy limitations of the collocation solution and derivative values that are interpolated and it is difficult to assess the convergence rates. However, from the smaller values of p and $nint$, we can see that the convergence rate is $p + 1$, this is one order lower than is expected based on corresponding results for the 1D case. As with the non-time-dependent case, the lower convergence rate of the SCI follows from the fact that the derivative values which are interpolated by the SCI do not converge at the expected rate, as discussed in section 4.3.

	Degree (p)							
	4		5		6		7	
nint	GE	$Rate$	GE	$Rate$	GE	$Rate$	GE	$Rate$
Problem 1								
2	9.48×10^{-3}		4.82×10^{-4}		2.77×10^{-5}		1.53×10^{-6}	
4	1.02×10^{-3}	3.214	1.35×10^{-5}	5.155	7.54×10^{-7}	5.201	1.13×10^{-8}	7.074
8	7.26×10^{-5}	3.816	4.15×10^{-7}	5.028	1.41×10^{-8}	5.741	3.30×10^{-9}	1.778
16	4.68×10^{-6}	3.955	1.30×10^{-8}	4.993	4.21×10^{-9}	1.742	1.63×10^{-4}	-15.59
Problem 2								
2	1.05×10^{-2}		1.82×10^{-3}		2.69×10^{-4}		4.33×10^{-4}	
4	1.28×10^{-3}	3.034	8.94×10^{-5}	4.352	5.37×10^{-6}	5.646	6.92×10^{-7}	9.29
8	9.20×10^{-5}	3.802	3.14×10^{-6}	4.829	1.55×10^{-7}	5.119	4.29×10^{-6}	-2.632
16	6.24×10^{-6}	3.882	1.13×10^{-7}	4.793	1.70×10^{-3}	-13.42	$0.00 \times 10^{+0}$	Inf

Table 4.20: Global error (GE) and convergence rate ($Rate$) of LOI for 2D, time-dependent problems, $p = 4, 5, 6, 7$, and $nint = 2, 4, 8, 16$.

Tables 4.20 and 4.21 present the errors and convergence rates of the LOI and LOI2, respectively.

The results for larger values of p and $nint$ are affected by the accuracy limitations of the collocation solution and derivative values that are interpolated. However, from looking at smaller values of p and $nint$, we observe that both interpolants converge at their expected rate of p. Comparing the magnitude of error, we see that the LOI tends to have an error which is one order larger than the LOI2.

	Degree (p)							
	4		5		6		7	
nint	*GE*	*Rate*	*GE*	*Rate*	*GE*	*Rate*	*GE*	*Rate*
Problem 1								
2	1.70×10^{-3}		1.65×10^{-4}		6.56×10^{-6}		5.06×10^{-7}	
4	1.70×10^{-4}	3.324	4.60×10^{-6}	5.169	1.80×10^{-7}	5.191	2.90×10^{-9}	7.448
8	1.17×10^{-5}	3.858	1.41×10^{-7}	5.029	3.94×10^{-9}	5.51	3.30×10^{-9}	-0.191
16	7.51×10^{-7}	3.965	4.67×10^{-9}	4.915	4.09×10^{-9}	-0.055	1.63×10^{-4}	-15.59
Problem 2								
2	3.50×10^{-3}		6.62×10^{-4}		8.19×10^{-5}		4.21×10^{-4}	
4	2.74×10^{-4}	3.671	3.08×10^{-5}	4.428	1.07×10^{-6}	6.266	1.55×10^{-7}	11.41
8	1.63×10^{-5}	4.072	1.07×10^{-6}	4.852	4.19×10^{-8}	4.667	4.29×10^{-6}	-4.795
16	1.02×10^{-6}	4.001	4.64×10^{-8}	4.522	1.70×10^{-3}	-15.31	$0.00 \times 10^{+0}$	Inf

Table 4.21: Global error (GE) and convergence rate ($Rate$) of LOI2 for 2D, time-dependent problems, $p = 4, 5, 6, 7$, and $nint = 2, 4, 8, 16$.

4.4.3 Interpolant Error and Convergence Rate Discussion

For all of the interpolants, there is no significant difference between the results for the non-time-dependent and time-dependent problem cases; therefore, the following discussion applies to both.

We observe that the SCI converges at a rate of only $p+1$, which is the same as that of the global error of the collocation solution. This is below the expected rate of $p + 2$ that is observed for the 1D case. As the SCI is constructed such that the data error dominates the interpolation error, it is no surprise that the expected convergence rate is not met. In section 4.3 we observed that the derivative values of the collocation solution that are interpolated by the SCI do not converge at their expected rates.

The LOI converges at its expected rate of p despite also interpolating the same derivative values as the SCI. This is because the LOI is constructed so that the interpolation error dominates the data

error. Thus, an increase in the error of some of the data values does not make a significant impact on the convergence rate as long as that error is still dominated by the interpolation error. The LOI2 also converges at its expected rate of p; there is no concern about the impact of the interpolated points not being sufficiently accurate as it does not interpolate the solution derivative values.

4.5 Error Estimation Results

In this section, we will examine the three error estimate schemes obtained from the SCI, the LOI, and the LOI2. The error estimation results are obtained by examining the accuracy of the error estimate on each sub-rectangle. An error estimate is calculated for each sub-rectangle by subtracting the interpolant from the collocation solution. This difference is then divided by the true error, and we take log base 10 of the result. This process will represent an error estimate that is one order of magnitude smaller than the true error as -1. An error estimate that is the same as the true error will be 0. An error estimate which is one order larger than the true error will be $+1$.

To plot this data, we group the error estimates from each sub-rectangle based on the log ratio as described above. The interval between the smallest and largest log ratios is divided into subintervals of width 0.1. Then, by counting how many of the error estimates for the sub-rectangles fall within each discrete subinterval, we can see how consistent the error estimates are. Within this section, we will use the term "consistent" to describe an interpolant that provides error estimates of similar accuracy for all of the sub-rectangles. In the graphs, a consistent error estimate will be tall and narrow, while an inconsistent error estimate will be short and wide. We can also assess the accuracy of the error estimates by examining where along the x-axis the error estimates are. If the error estimate plots are close to 0, that means that the error estimate correlates well with the true error.

In Chapter 2 we discussed local extrapolation where we saw that an approximate solution with an error of order $p-1$ is used to estimate the error for an approximate solution that has an error of order p. The LOI and LOI2 yield this type of error estimate. This type of error estimate tends to overestimate the true error. We will try to correct this overestimation in the LOI and LOI2 by also applying a scaling factor of h^2 (the area of the sub-rectangle) to the results; these results are

labelled as LOI (S) and LOI2 (S).

4.5.1 2D, Non-Time-Dependant Case

Figures 4.7 through 4.17 present the error estimation results from each interpolant for each of the three non-time-dependent test problems, degrees 5 through 7, and with 8 subintervals in each spatial dimension. The first three figures, 4.4 to 4.6, consider problem 1, degree 4, and 4, 8, and 16 subintervals in each dimension. In these first three figures, we see that the number of subintervals makes no meaningful difference to the consistency or accuracy of the error estimates. Problem 1 results in the least amount of variance in the accuracy of the error estimates while the error estimates for Problem 2 have the most variance. The variance in the quality of the error estimates also increases with the degree of the collocation solution, with the LOI and LOI2 being more impacted by this than the SCI.

The SCI based error estimate performs well in these test cases and provides very consistent error estimates except for certain cases of higher degree. This is unexpected as we saw in the previous sections that the collocation solution and derivative values at mesh points which are interpolated by the SCI do not meet their expected convergence rates.

The LOI based error estimate, as expected, tends to overestimate the error for most test cases; however the scaling, mentioned earlier, brings the error estimate to within the right order of magnitude in some cases (see Figures 4.7 and 4.8). However as the degree increases the scaled LOI, begins to underestimate the error and the original LOI gives a better error estimate. Results for Problem 3 show significant issues for the LOI with all of the degrees resulting in error estimates of variable quality. For Problem 2 we see that the LOI delivers more consistent error estimates. For Problem 1 we see that the LOI error estimates are very consistent.

The LOI2 based error estimate yields results that are very similar to those of the LOI, and the scaling is very useful for lower degrees but results in underestimations of the true error for higher degrees. The LOI2 does however provide error estimates with more variance than those associated with the LOI. Also, the LOI2 tends to give a smaller error estimate than the error estimate associated

with the LOI in most cases.

Problem 1, degree 4, nint varies

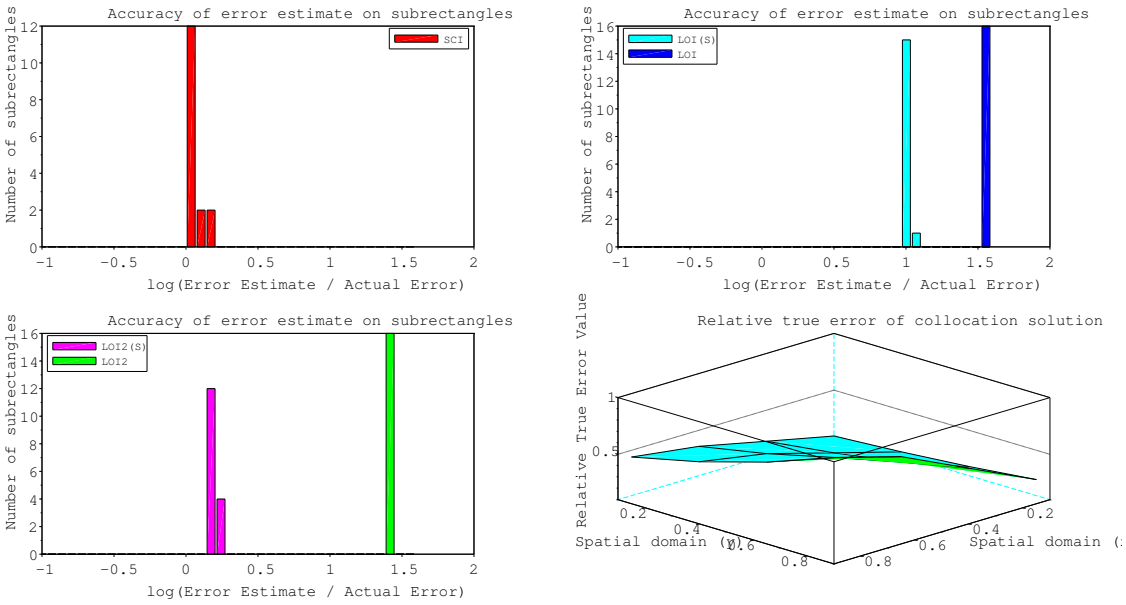

Figure 4.4: Error estimates for 2D non-time-dependent PDE Problem 1, degree 4, nint 4.

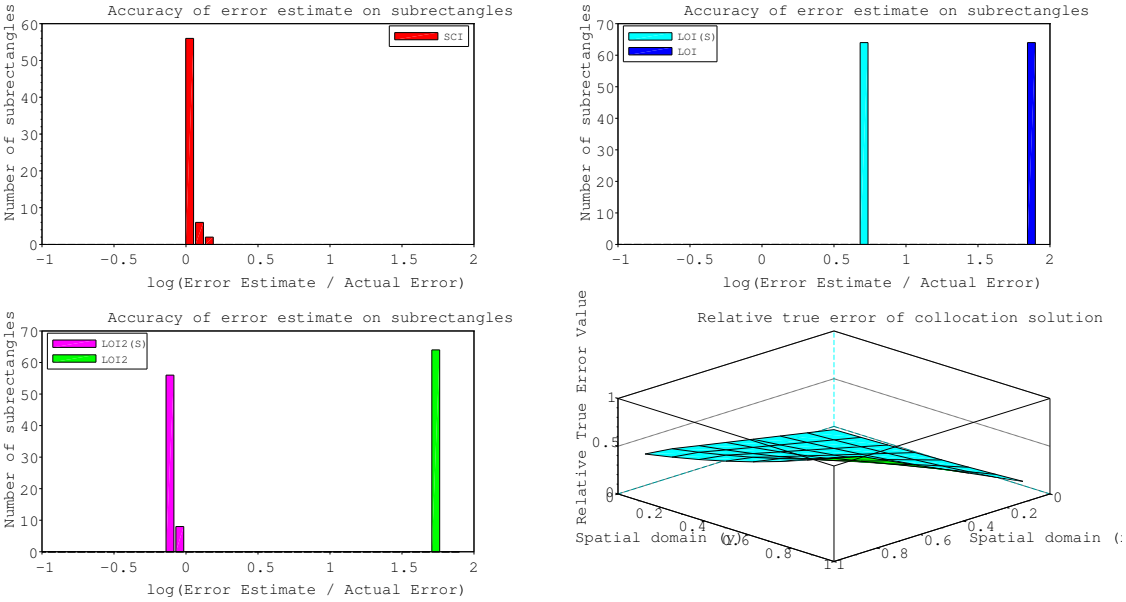

Figure 4.5: Error estimates for 2D non-time-dependent PDE Problem 1, degree 4, nint 8.

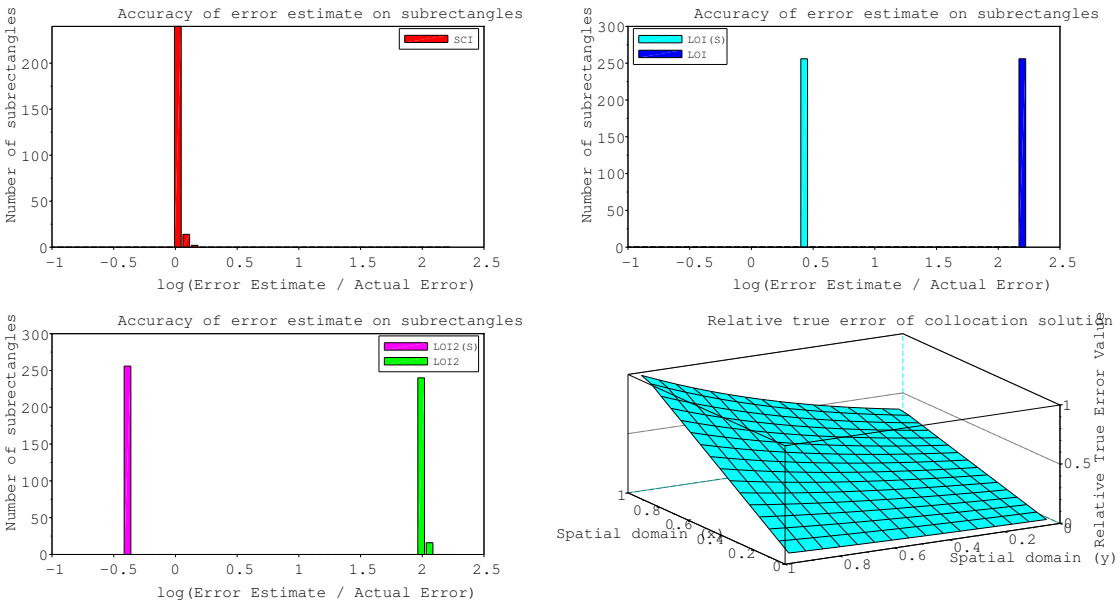

Figure 4.6: Error estimates for 2D non-time-dependent PDE Problem 1, degree 4, nint 16.

Problem 1, degree varies, nint = 8

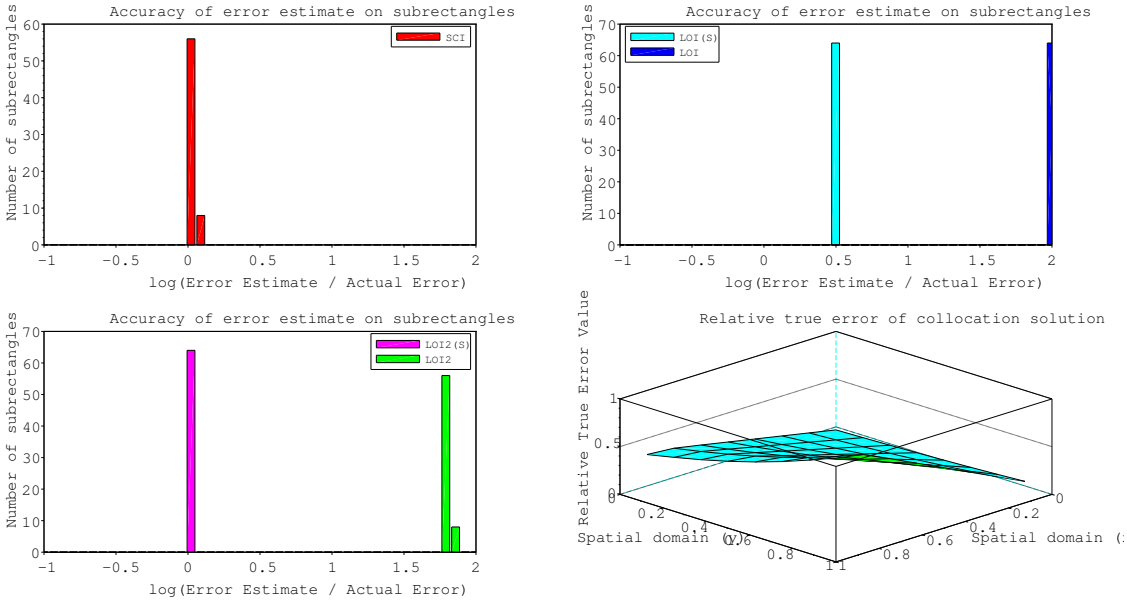

Figure 4.7: Error estimates for 2D non-time-dependent PDE Problem 1, degree 5, nint 8.

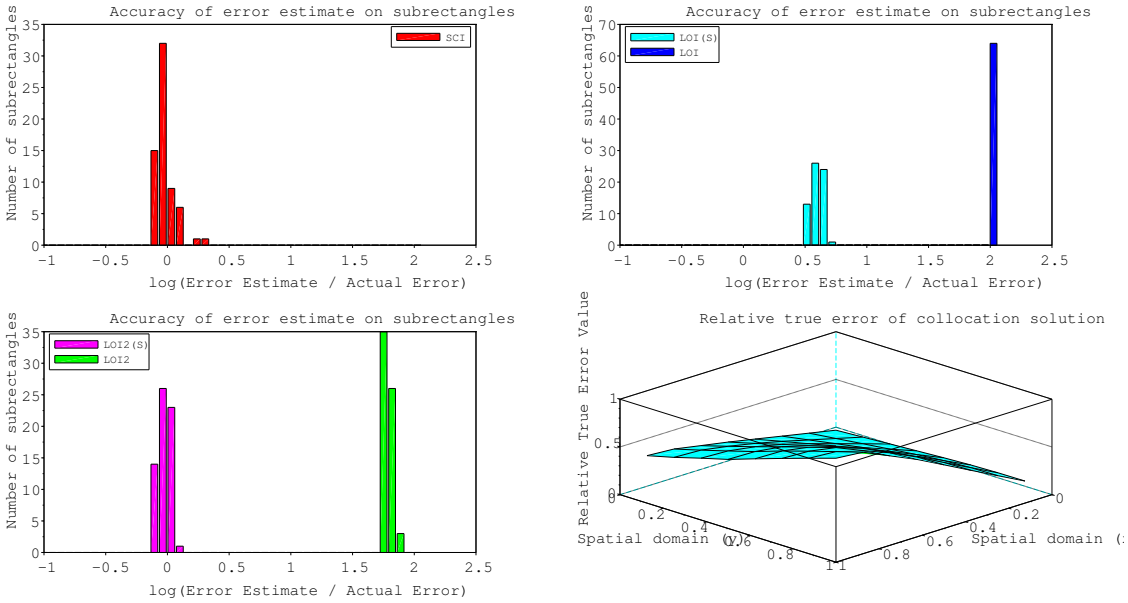

Figure 4.8: Error estimates for 2D non-time-dependent PDE Problem 1, degree 6, nint 8.

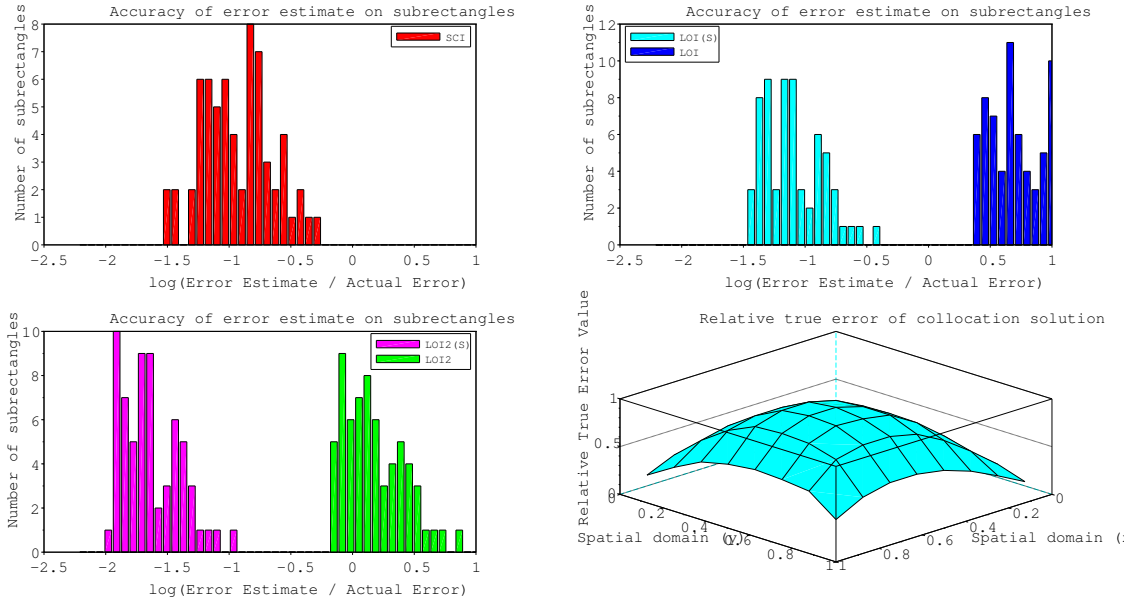

Figure 4.9: Error estimates for 2D non-time-dependent PDE Problem 1, degree 7, nint 8.

Problem 2, degree varies, nint = 8

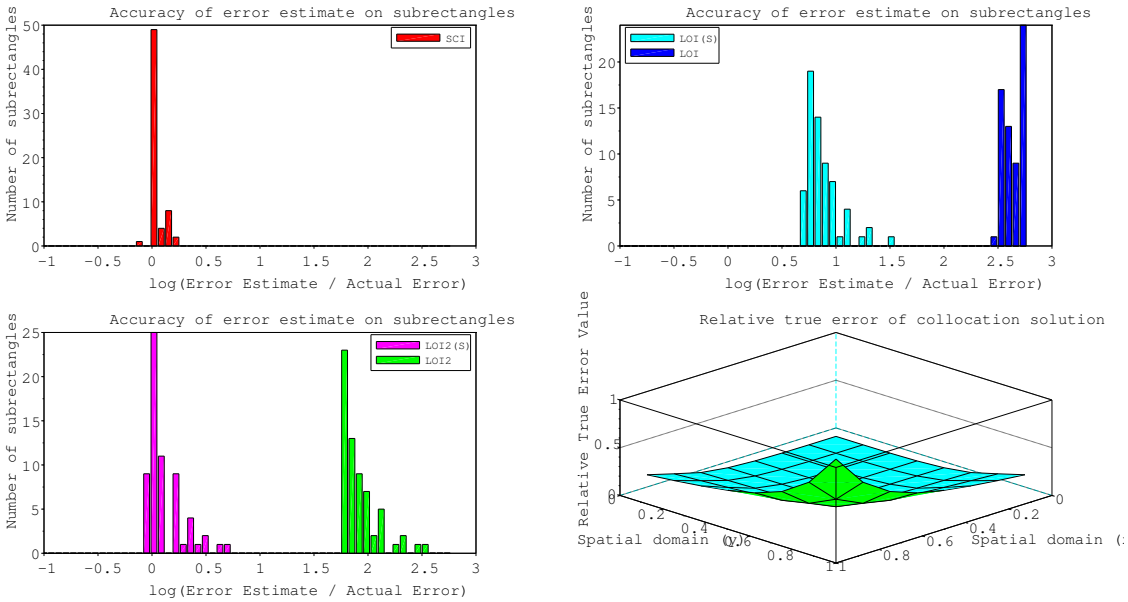

Figure 4.10: Error estimates for 2D non-time-dependent PDE Problem 2, degree 4, nint 8.

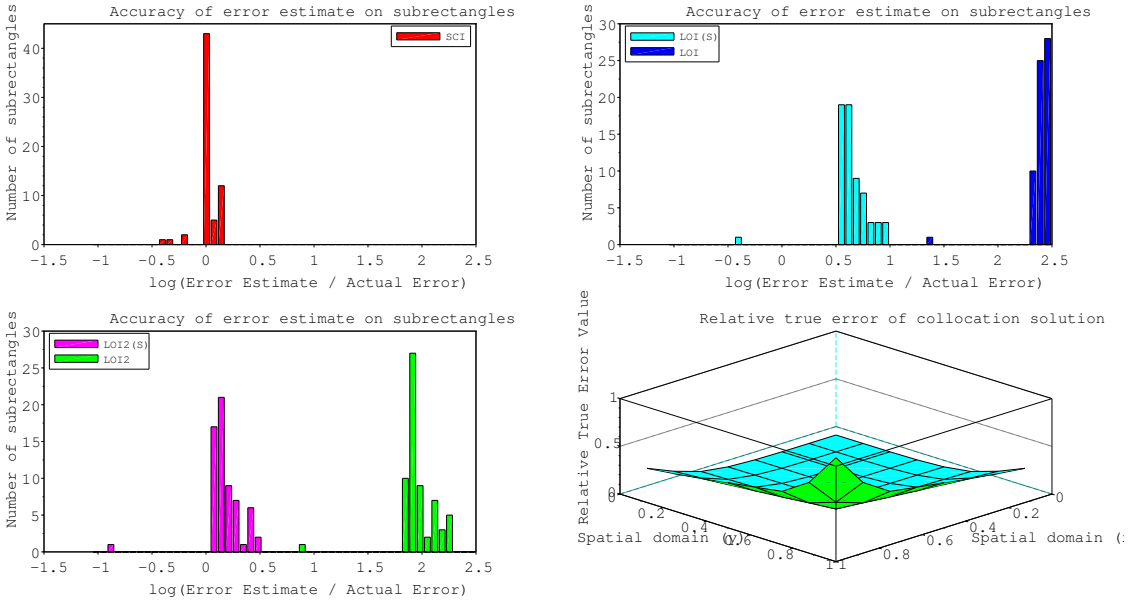

Figure 4.11: Error estimates for 2D non-time-dependent PDE Problem 2, degree 5, nint 8.

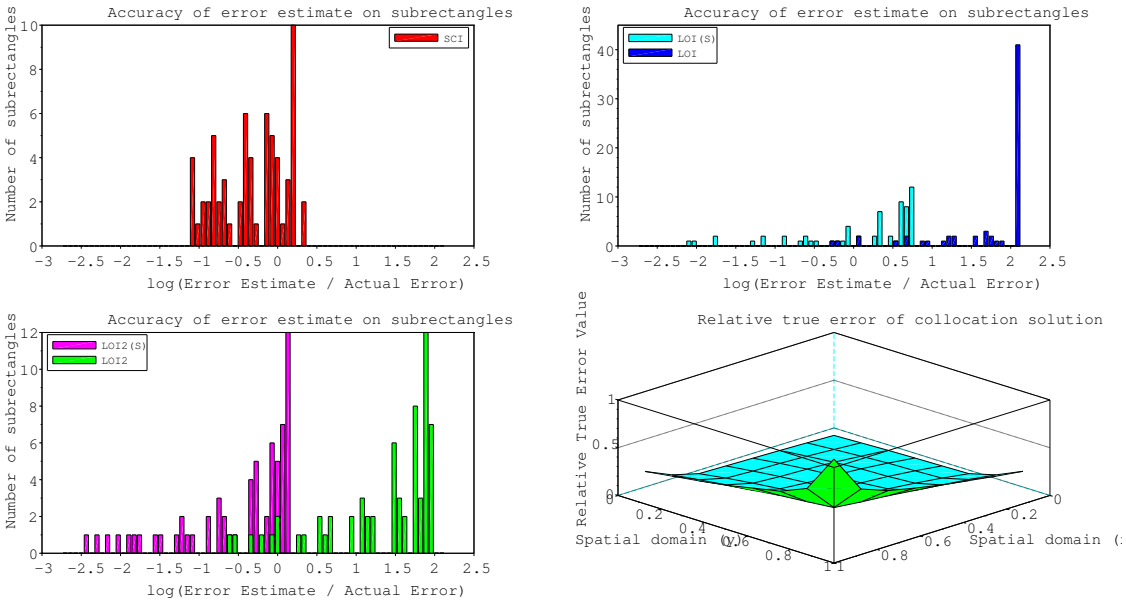

Figure 4.12: Error estimates for 2D non-time-dependent PDE Problem 2, degree 6, nint 8.

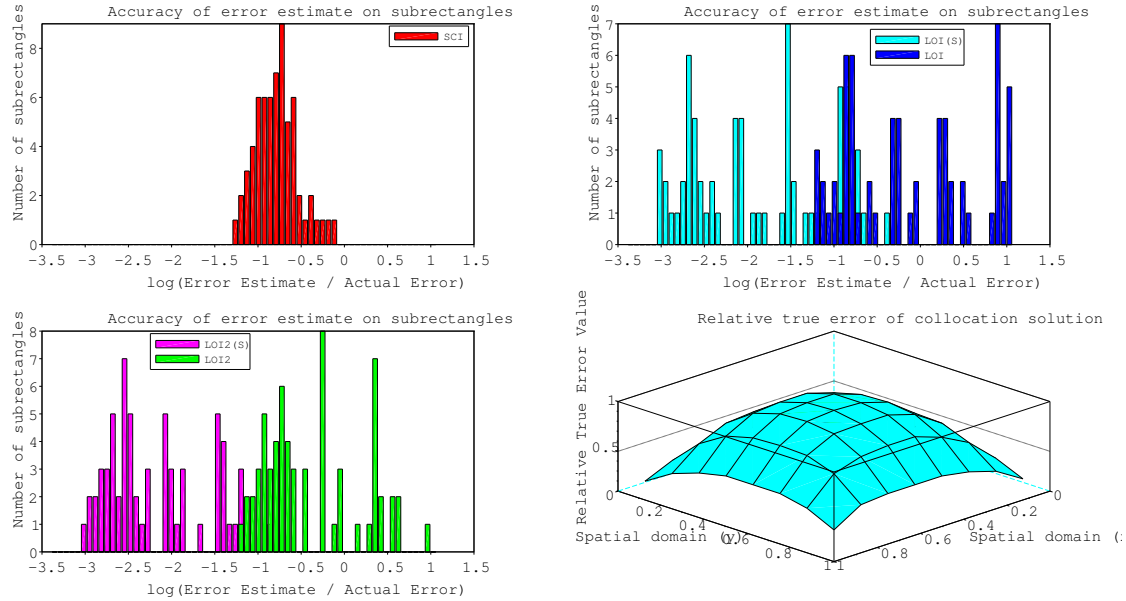

Figure 4.13: Error estimates for 2D non-time-dependent PDE Problem 2, degree 7, nint 8.

Problem 3, degree varies, nint = 8

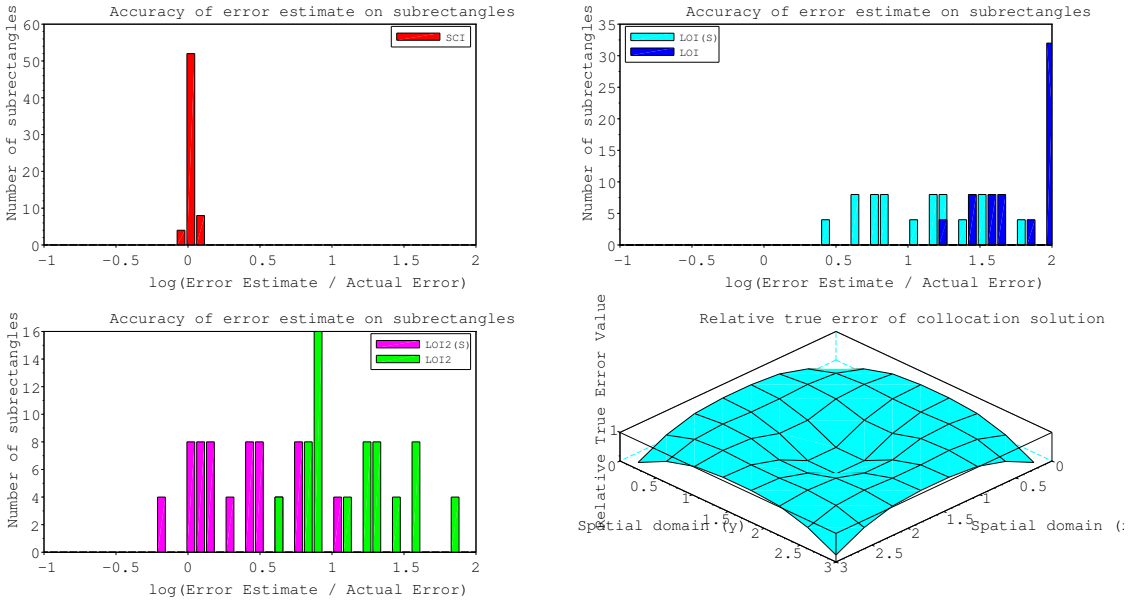

Figure 4.14: Error estimates for 2D non-time-dependent PDE Problem 3, degree 4, nint 8.

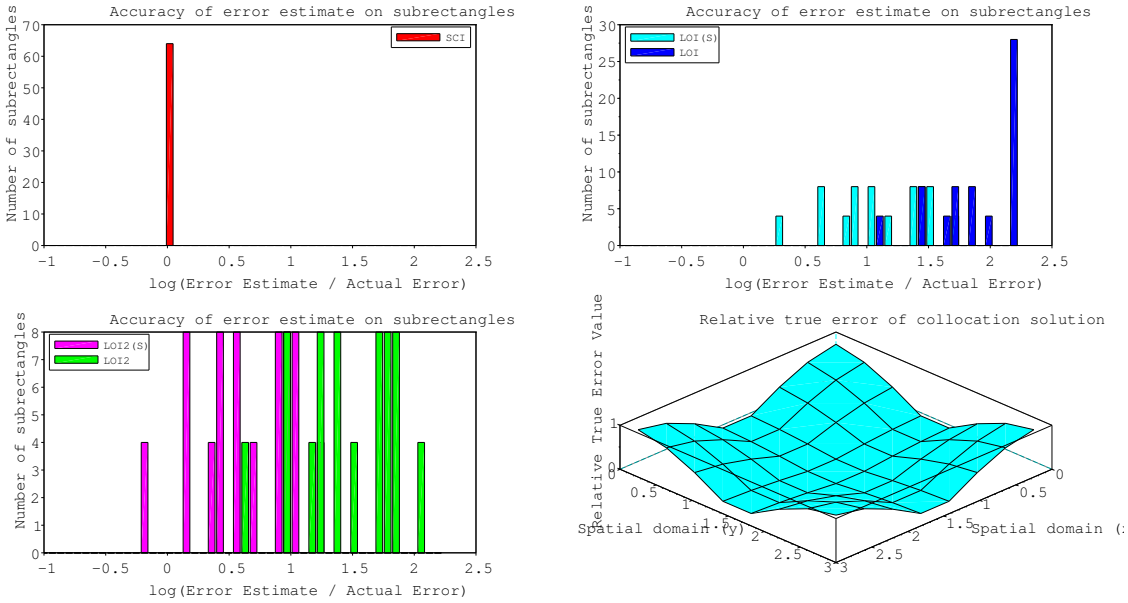

Figure 4.15: Error estimates for 2D non-time-dependent PDE Problem 3, degree 5, nint 8.

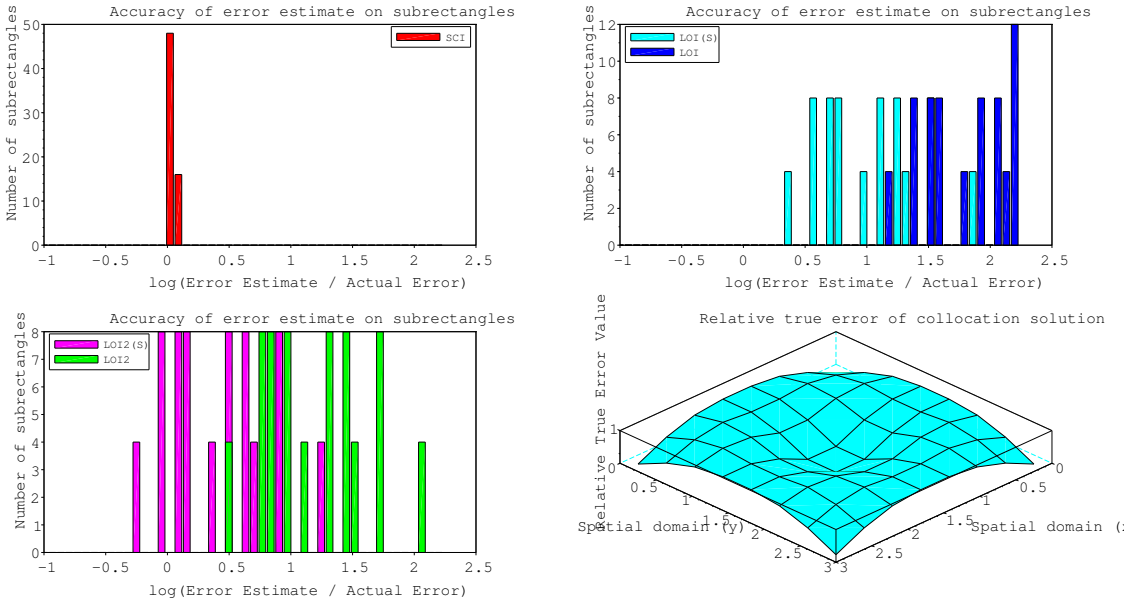

Figure 4.16: Error estimates for 2D non-time-dependent PDE Problem 3, degree 6, nint 8.

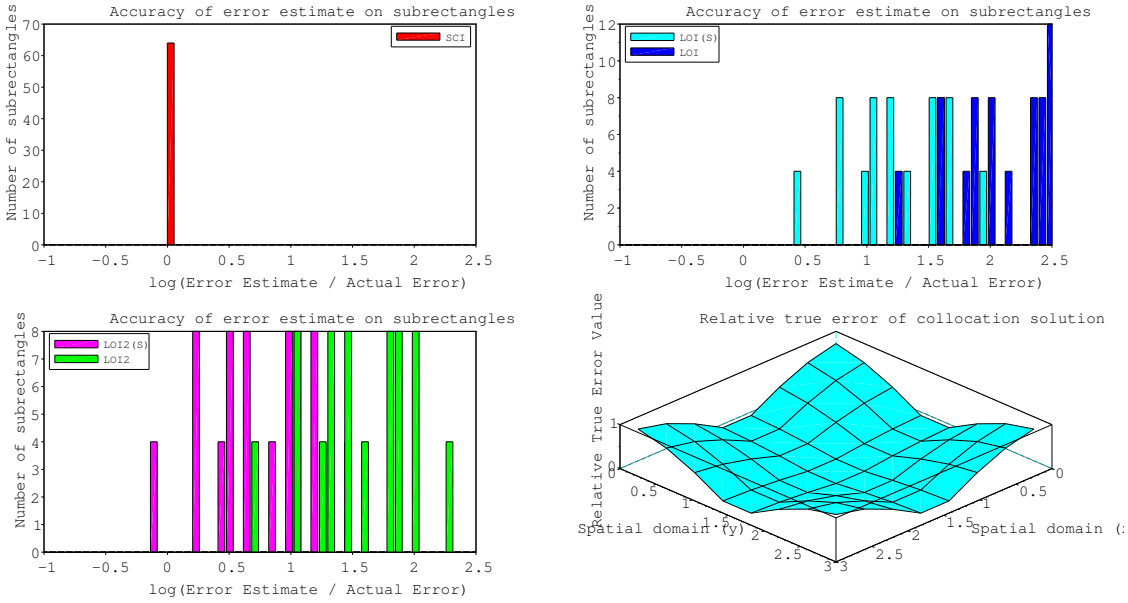

Figure 4.17: Error estimates for 2D non-time-dependent PDE Problem 3, degree 7, nint 8.

4.5.2 2D, Time-Dependant Case

Figures 4.21 through 4.27 present the error estimation results for each interpolant for each of the two time-dependent test problems, for degrees 5 through 7, and with 8 subintervals in each spatial dimension. The first three figures, 4.18 to 4.20, consider problem 1, degree 4, and 4, 8, and 16 subintervals in each dimension. In these first three figures, we see that the number of subintervals

makes no meaningful difference to the consistency or accuracy of the error estimates.

For all of the interpolants, as the degree increases so does the variance of the error estimates. The SCI based error estimate seems to be more impacted by the degree than in the non-time-dependent case. The SCI is able to provide very good quality and consistent error estimates for some of the test cases when the degree is low. However, for degree 7, the SCI based error estimate underestimates the true error.

The LOI and LOI2 based error estimates also provide a smaller error estimate than their non-time-dependent counterparts. This means that scaled versions of the error estimates are the better choice for degree 4. The test cases where the LOI and LOI2 based error estimates perform very erratically are also when the SCI based error estimates do not perform well. This could be due to the nature of the test problems or the accuracy with which the collocation solutions were calculated.

Problem 1, degree 4, nint varies

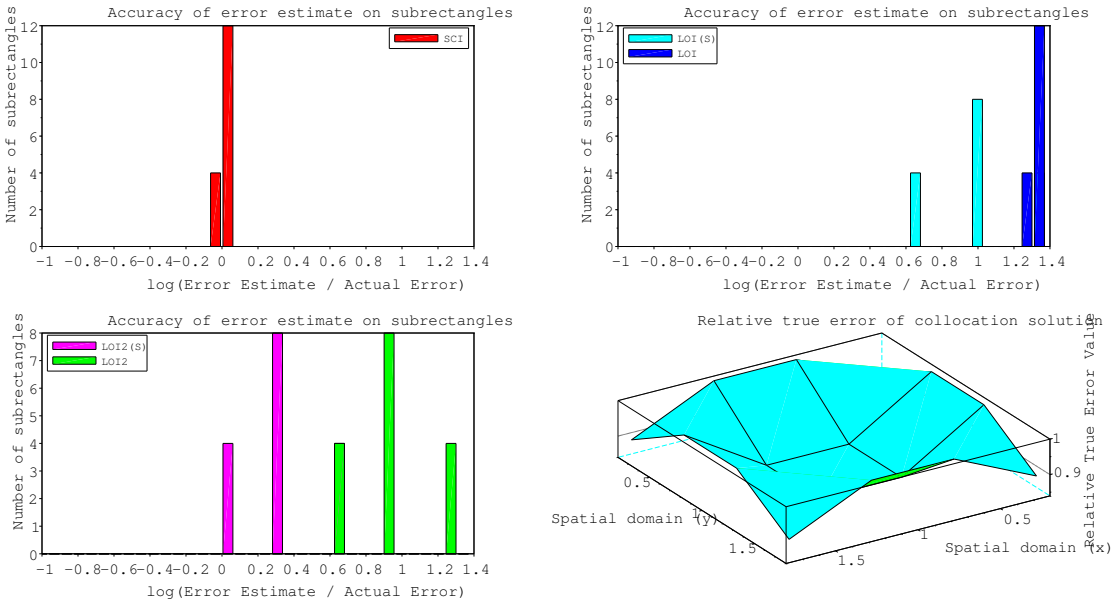

Figure 4.18: Error estimates for 2D time-dependent PDE Problem 1, degree 4, nint 4.

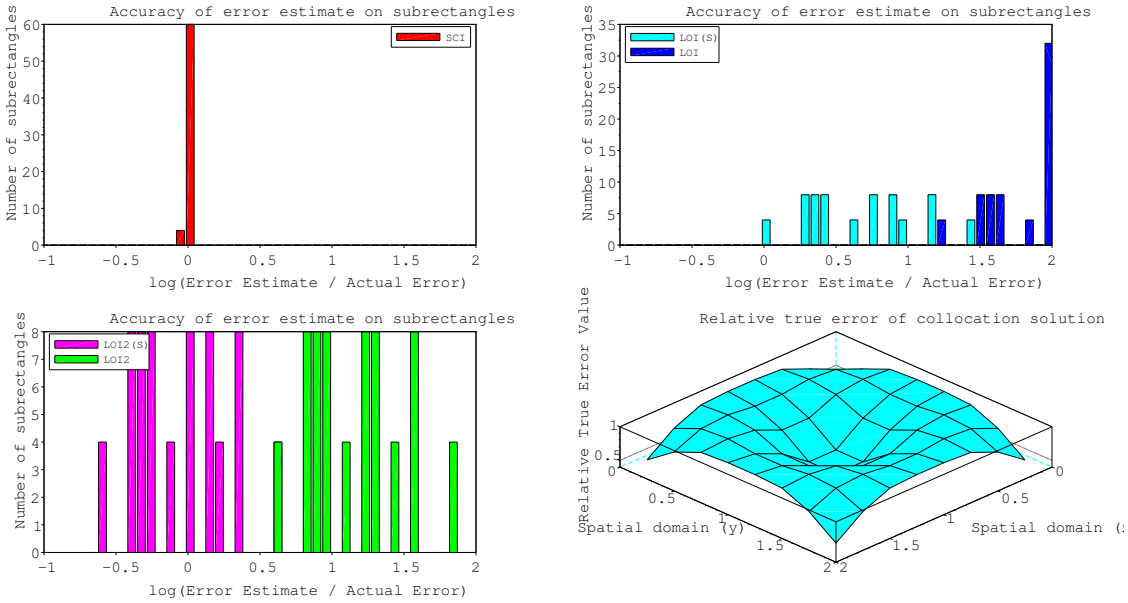

Figure 4.19: Error estimates for 2D time-dependent PDE Problem 1, degree 4, nint 8.

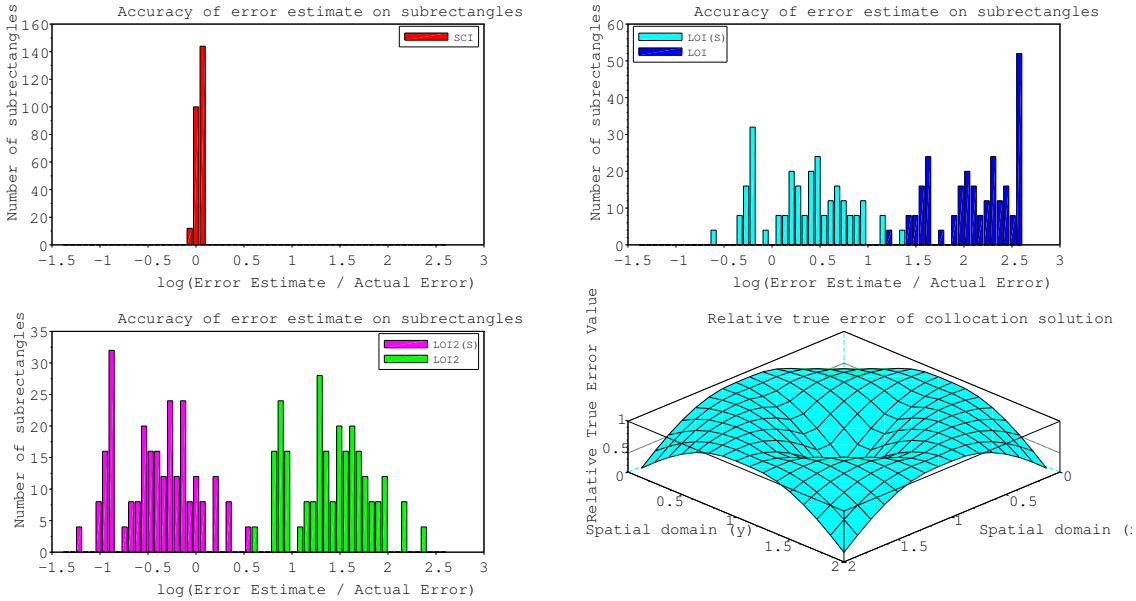

Figure 4.20: Error estimates for 2D time-dependent PDE Problem 1, degree 4, nint 16.

Problem 1, degree varies, nint = 8

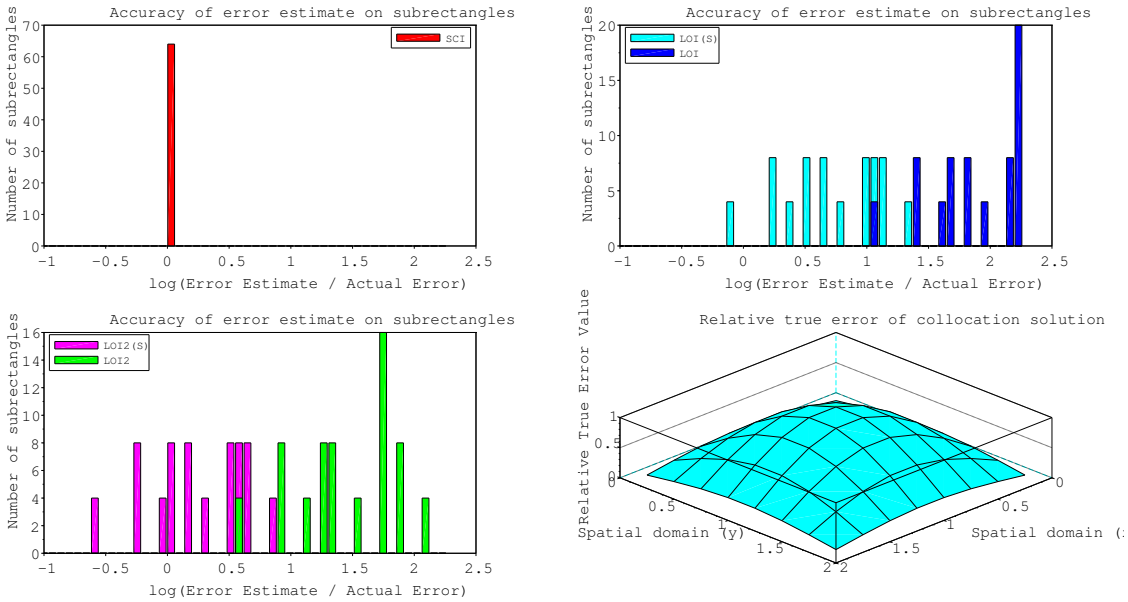

Figure 4.21: Error estimates for 2D time-dependent PDE Problem 1, degree 5, nint 8.

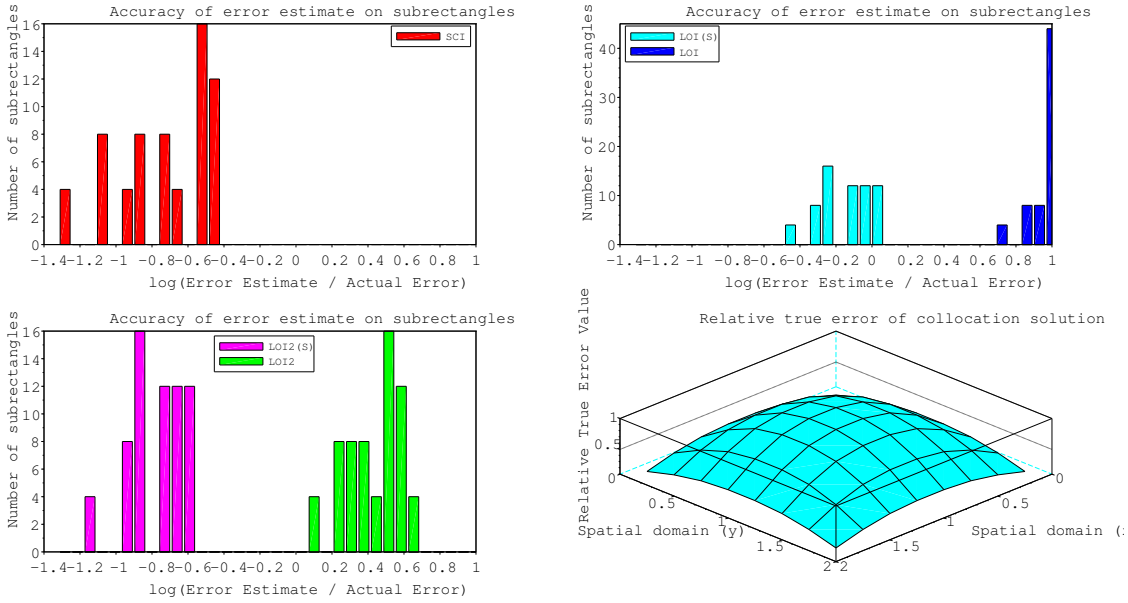

Figure 4.22: Error estimates for 2D time-dependent PDE Problem 1, degree 6, nint 8.

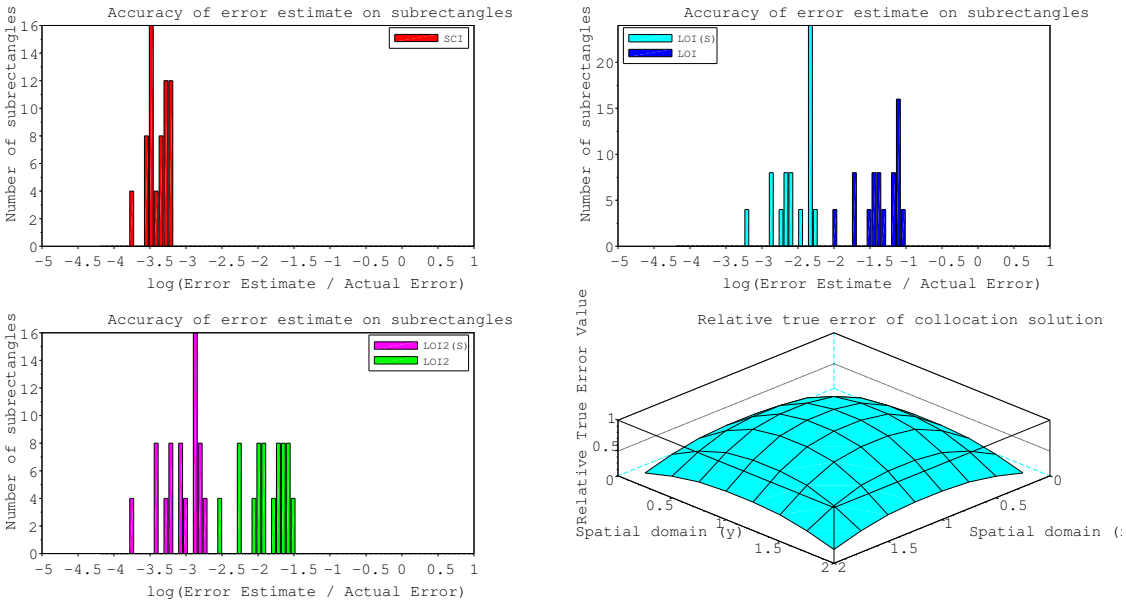

Figure 4.23: Error estimates for 2D time-dependent PDE Problem 1, degree 7, nint 8.

Problem 2, degree varies, nint = 8

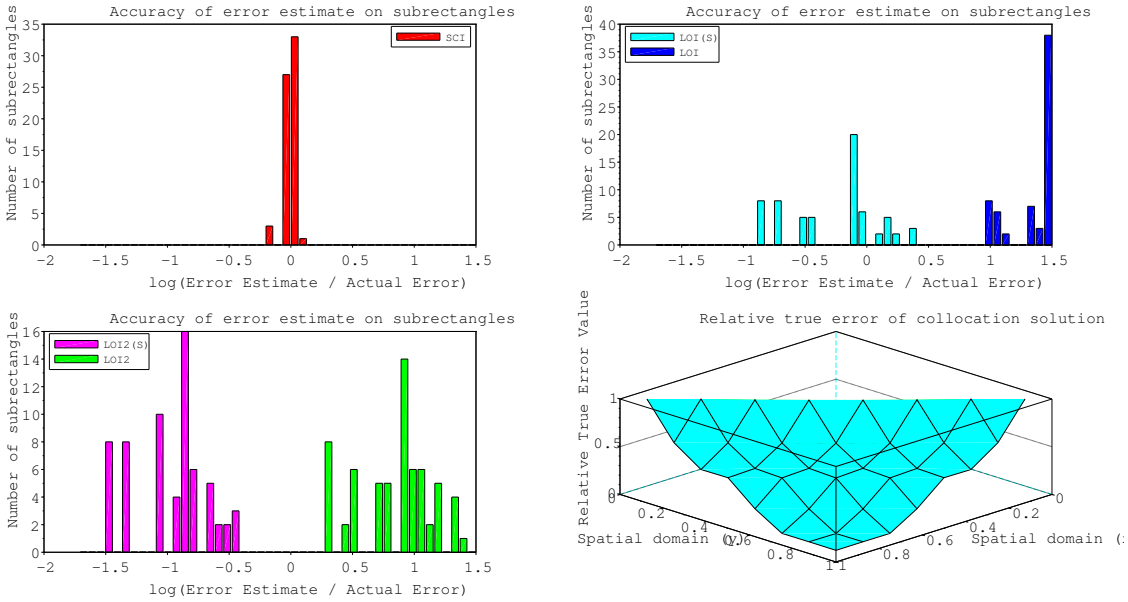

Figure 4.24: Error estimates for 2D time-dependent PDE Problem 2, degree 4, nint 8.

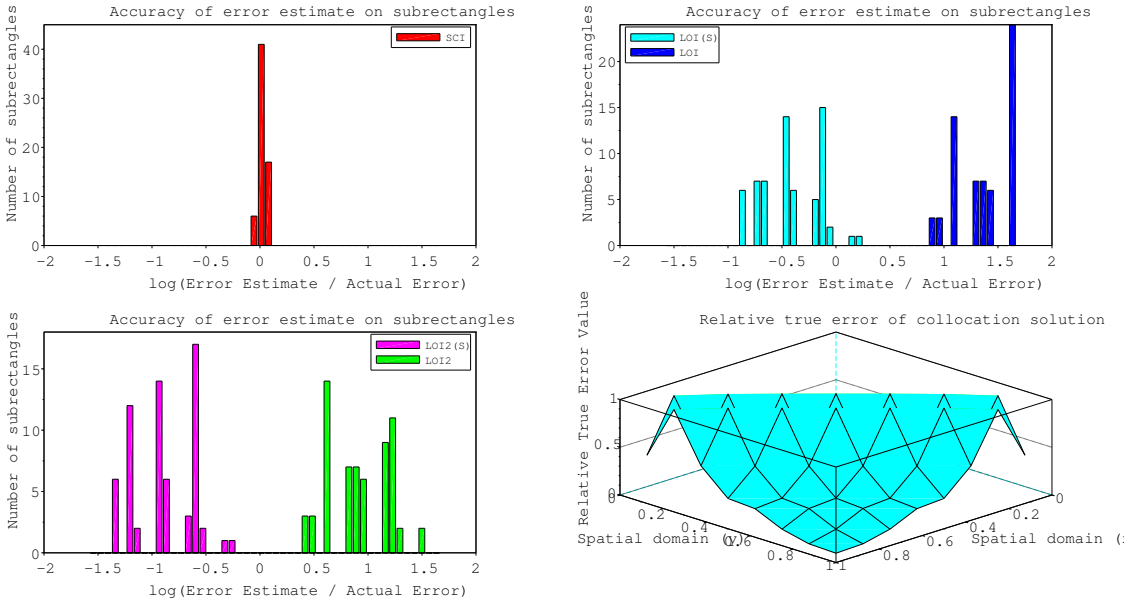

Figure 4.25: Error estimates for 2D time-dependent PDE Problem 2, degree 5, nint 8.

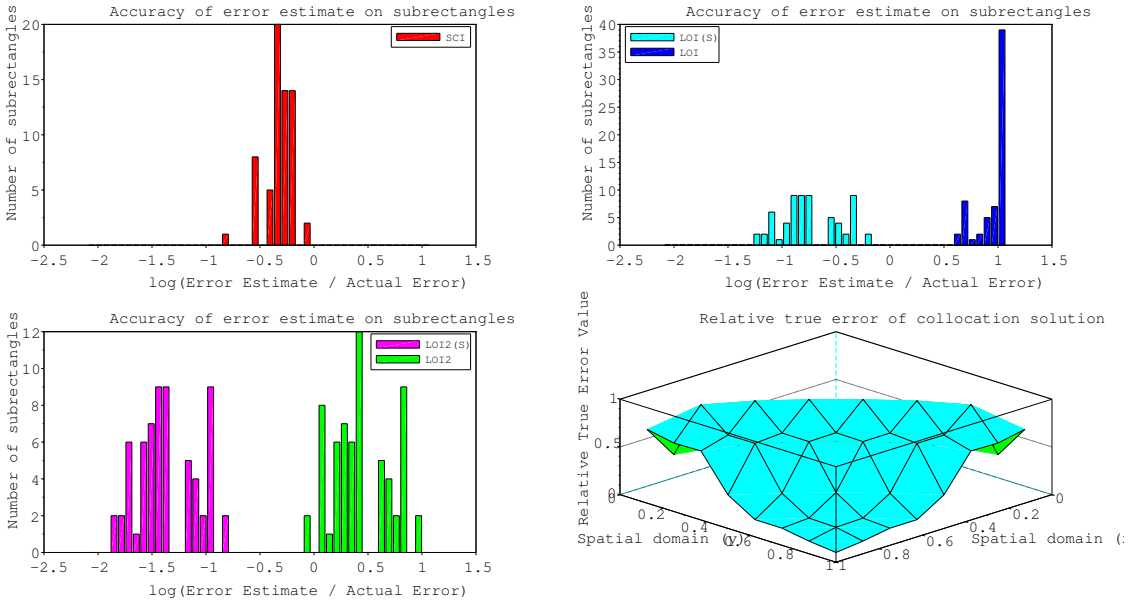

Figure 4.26: Error estimates for 2D time-dependent PDE Problem 2, degree 6, nint 8.

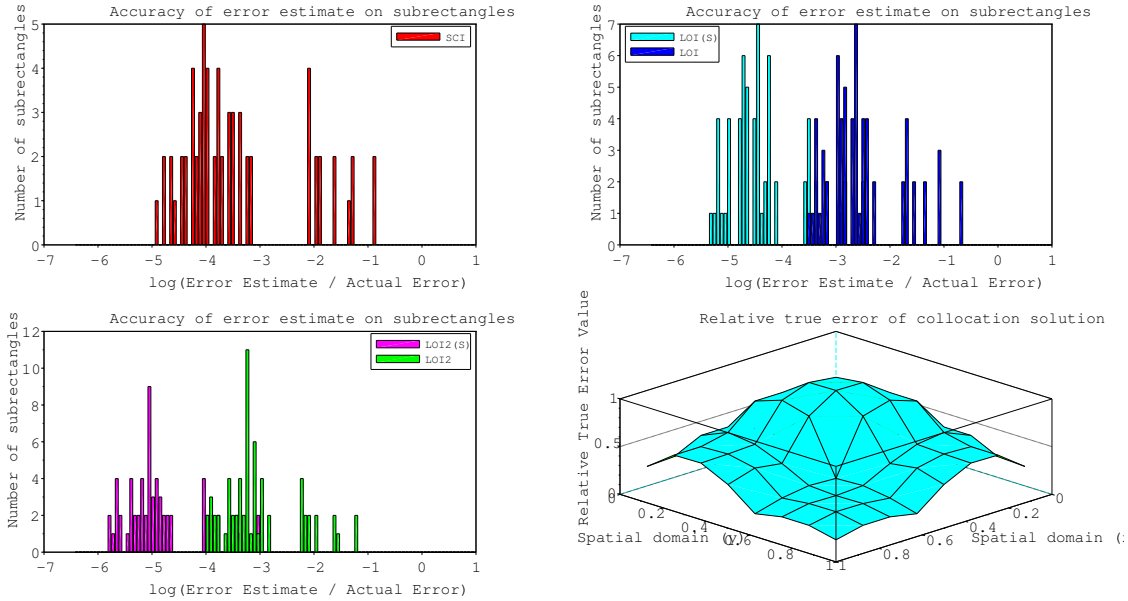

Figure 4.27: Error estimates for 2D time-dependent PDE Problem 2, degree 7, nint 8.

4.6 Error Estimation Results Discussion

From the error estimates for the time-dependent and non-time-dependent test cases, we see some common trends. One such trend is that the error estimates underestimate the true error as the degree increases. The number of subintervals per dimension does not seem to impact the error estimates. Another trend we see is that the LOI and LOI2 based error estimates have similar consistency. Also,

the error estimates provided by the LOI2 tend to give a smaller error estimate than those associated with the LOI. This means that the LOI2 gives a better error estimate for smaller degrees. This also means that at higher degrees the LOI2 will underestimate the true error by more than the LOI will.

The SCI based error estimates tend to be the most consistent. The SCI based error estimates are very accurate for degrees 4 and 5. For degrees 6 and 7, the SCI based error estimates give an underestimate of the true error. The LOI based error estimates give an overestimate of the true error by about one order of magnitude. The scaled version of the LOI based error estimates also give an overestimate of the true error, but the error estimates for most sub-rectangles are within an order of magnitude of the true error. The LOI2 based error estimates provide an error estimate which is between one and two orders of magnitude above the true error. The scaled LOI2 based error estimates give an estimate of the true error that is of the correct order. However, some sub-rectangles have an error estimate that is one order of magnitude larger than the true error.

Since the error estimates give underestimates of the true error as the degree increases, it may be worthwhile to consider introducing a scaling factor for the error estimates which depends on the degree. The performance of the SCI based error estimates also needs to be examined further. The convergence results we saw in Section 4.3 indicate that the SCI should not provide a reliable error estimate; however, we have seen from the results of this section that the SCI provides good-quality error estimates in some cases. As well, the inconsistency of the LOI and LOI2 based error estimates in cases where the SCI based error estimate is consistent needs further examination.

Chapter 5

Conclusion

From the results of Chapter 3 we saw that error control can have a significant impact on the solution of COVID-19 type PDE models. While the error in an approximate solution with any error tolerance increased as the time integration progressed, the errors remained relatively controlled.

From the results of Chapter 4, where we considered 2D collocation methods and 2D interpolation based error estimates, we were able to experimentally confirm that the global convergence rate of a collocation solution remains the same for two dimensions as in one dimension. However, at the mesh points of the 2D collocation solution, we do not observe experimentally the higher order of convergence which is observed in the one-dimensional case. The effects of this are seen in the convergence rate of the 2D interpolants, as the 2D SCI converges at the same rate as the collocation solution instead of one order higher, as in the one-dimensional case.

Despite not converging at the expected rate, the 2D SCI is still able to provide accurate and reliable error estimates for most of the test problems. The 2D LOI and 2D LOI2 based error estimates behave similarly to each other, although the 2D LOI2 will typically give a smaller error estimate than that given by the 2D LOI. The 2D LOI and 2D LOI2 were able to provide accurate error estimates in some cases although they did not match the accuracy or consistency of the 2D SCI. As these two interpolants tend to overestimate the true error, the scaling that was applied to their associated error estimates resulted in a more accurate error estimate. However, as the degree

of collocation solution grows, the 2D LOI and 2D LOI2 overestimate the error by less. Therefore, a scaling factor which considers the degree of collocation solution could be more effective for these interpolants.

A valuable future project could include a theoretical analysis of the rate of convergence of a 2D collocation solution at the mesh points, i.e., at the corners of the rectangles into which the spatial domain is subdivided. Also, the 2D interpolation-based error estimation methods could be implemented within collocation software featuring error control to see if they provide error estimates with enough quality to optimize the computation of approximate solutions. An error control algorithm based on adaptive moving mesh algorithm [11] would appear to be a good framework within which the error estimates considered in this thesis could be employed.

Bibliography

[1] Arcede, J., Caga-anan, R., Mentuda, C., and Mammeri, Y. Accounting for Symptomatic and Asymptomatic in a SEIR-type model of COVID-19. *Math. Model Nat. Phenom. 15* (2020), Article 34.

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

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

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

[5] Brown, P., Hindmarsh, A., and Petzold, L. Consistent initial condition calculation for differential-algebraic systems. *SIAM Journal on Scientific Computing 19* (09 1998), 1495–1512.

[6] Christara, C. Private Communication, December 2020.

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

[8] Diaz, J., Fairweather, G., and Keast, P. 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 (1983), 376–380.

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

[10] Holmes, E., Lewis, M., Banks, J., and Veit, D. Partial differential equations in ecology: Spatial interactions and population dynamics. *Ecology 75* (01 1994), 17–29.

[11] Huang, W., and Russell, R. *Adaptive Moving Mesh Methods*, vol. 174. 01 2011.

[12] Kermack, W., and McKendrick, A. A contribution to the mathematical theory of epidemics. *Proc. R. Soc. Lond. A 115* (1927), 700–721.

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

[14] Papadomanolaki, M., and Saridakis, Y. Hermite-collocation for one dimensional tumor invasion model with heterogeneous diffusion. In *Proceedings 9th HERCMA Conference* (2009).

[15] Petzold, L. R. Description of DASSL: a differential/algebraic system solver. Tech. rep., Sandia Labs, Livermore, CA, 1982.

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

[17] Sarkar, K., Khajanchi, S., and Nieto, J. Modeling and forecasting the COVID-19 pandemic in India. *Chaos, Solitons Fractals 139* (2020), Article 11049.

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

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 = 2 // 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 = 1 // 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


// 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, status] = 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
    status = -1
    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 with DASKR tolerance of " +
        string(atol*0.1))
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))
    status = 1
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, fg)
    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
    info = -1
    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

    // Check if a first guess of coefficients was supplied
    if argn(2) < 3 then
        fg = zeros(ncpts,1)
    end


    // 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

    jacMode = 1;
    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
global t
global mockY mockYdot


// 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 [c, status, iCoeffs, iCoDer] = collocate(deg, nints, ti, tOut,
    iCoeffs, iCoDer)
    status = 0 ;

    // Check for proper number of input parameters
    if ~(argn(2) == 4 | argn(2) == 6) then
        disp("Unsupported input parameters.")
        return
    end

    // 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
tc = ti
colInds = saveColInds()
xBndInds = savexBndInds()
yBndInds = saveyBndInds()
cornInds = saveCornInds()
if verbose > 0 then
    disp("Calculating initial coefficients.")
    if verbose > 1 then
        tic
    end
end
jacMode = 0
if argn(2) == 4 then
    [coeffs, initErr] = initCoeffs()
else
    r = max(abs(fsol(iCoeffs)))
    disp("Previous fsolve residual: " + string(r))
    [coeffs, initErr] = initCoeffs(iCoeffs)
end

if initErr > atol then
    status = -4
    disp("Initial coefficients could not meet tolerance.")
    return
end

iCoeffs = coeffs
dasTol = max(initErr*0.1, atol) ;

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

jacMode = 0
if argn(2) == 4 then
    coDer = initCoeffsDer()
else
   coDer = iCoDer
end
jacMode = 1

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 with DASKR tolerance of " +
        string(dasTol))
end

// INFO list which controls running of DASKR
info = list([], 1, [], [], [], 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, [t0, tOut], dasTol, resV,ng
        , gr1, info)
elseif jacMode == 1 then
    [r, nn] = daskr([coeffs, coDer], t0, [t0, tOut], dasTol, resV,
        qjacResV,ng, gr1, info)
end

// Check for failed first step
if size(r)(2) == 1 then
    if verbose > 0 then
        disp("DASKR failed on first step.")
    end
    status = -1 ;
    return
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))
allCoeffs = r(2:ncpts+1, :)
tc = r(1, size(r)(2))
c = coeffs
status = 1;
iCoeffs = r(2:ncpts+1,1)
iCoDer = r(ncpts+2:2*ncpts+1)
```

```
endfunction

// Efficient approximation of Jacobian requested by DASKR.
function r = qjacRes4(t, y, ydot, cj)
    tic
    inds = nonZeroInds(1)
    mockY = y
    mockYdot = ydot

    for k = 1:size(inds)(1)
        ind = inds(k)
        mockYdot(1) = ydot(1)+dy
        mockY(1) = y(1)+dy
        rdp1 = resI(t, y, mockYdot, ind)
        rp1 = resI(t, mockY, ydot, ind)
        mockYdot(1) = ydot(1)-dy
        mockY(1) = y(1)-dy
        rdm1 = resI(t, y, mockYdot, ind)
        rm1 = resI(t, mockY, ydot, ind)
        mockYdot(1) = ydot(1)-2*dy
        mockY(1) = y(1)-2*dy
        rdm2 = resI(t, y, mockYdot, ind)
        rm2 = resI(t, mockY, ydot, ind)
        mockYdot(1) = ydot(1)+2*dy
        mockY(1) = y(1)+2*dy
        rdp2 = resI(t, y, mockYdot, ind)
        rp2 = resI(t, mockY, ydot, ind)
        r(ind,1) = (((rm2)/12) + ((-2*rm1)/3) + ((2*rp1)/3) + ((-rp2)
            /12))/dy
        r(ind,1) = r(ind,1) + cj * (((rdm2)/12) + ((-2*rdm1)/3) + ((2*
            rdp1)/3) + ((-rdp2)/12))/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)
            mockYdot(i) = ydot(i)+dy
            mockY(i) = y(i)+dy
            rdp1 = resI(t, y, mockYdot, ind)
            rp1 = resI(t, mockY, ydot, ind)
            mockYdot(i) = ydot(i)-dy
            mockY(i) = y(i)-dy
            rdm1 = resI(t, y, mockYdot, ind)
            rm1 = resI(t, mockY, ydot, ind)
            mockYdot(i) = ydot(i)-2*dy
            mockY(i) = y(i)-2*dy
            rdm2 = resI(t, y, mockYdot, ind)
            rm2 = resI(t, mockY, ydot, ind)
            mockYdot(i) = ydot(i)+2*dy
            mockY(i) = y(i)+2*dy
            rdp2 = resI(t, y, mockYdot, ind)
            rp2 = resI(t, mockY, ydot, ind)
```

```
            r(ind,i) = (((rm2)/12) + ((-2*rm1)/3) + ((2*rp1)/3) + ((-rp2
                )/12))/dy
            r(ind,i) = r(ind,i) + cj * (((rdm2)/12) + ((-2*rdm1)/3) +
                ((2*rdp1)/3) + ((-rdp2)/12))/dy
        end
    end
    time=toc()
    disp(string(time) + " seconds spent in qjacres4.")
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

function r = qjacResV(tin, y, ydot, cj)
    global mockY mockYdot t
    inds = nonZeroInds(1)
    mockY = y
    mockYdot = ydot
    t = tin
```

```
    mockY(1) = y(1)+dy
    mockYdot(1) = ydot(1)+dy
    rp1 = feval(inds, resIV)
    rdp1 = feval(inds, resdIV)
    mockY(1) = y(1)-dy
    mockYdot(1) = ydot(1)-dy
    rm1 = feval(inds, resIV)
    rdm1 = feval(inds, resdIV)
    r(inds,1) = (0.5*rp1 - 0.5*rm1)/dy
    r(inds,1) = r(inds,1) + cj*(0.5*rdp1 - 0.5*rdm1)/dy

    for i = 2:ncpts
        inds = nonZeroInds(i)
        mockY(i-1) = y(i-1)
        mockYdot(i-1) = ydot(i-1)
        mockY(i) = y(i)+dy
        mockYdot(i) = ydot(i)+dy
        rp1 = feval(inds, resIV)
        rdp1 = feval(inds, resdIV)
        mockY(i) = y(i)-dy
        mockYdot(i) = ydot(i)-dy
        rm1 = feval(inds, resIV)
        rdm1 = feval(inds, resdIV)
        r(inds,i) = (0.5*rp1 - 0.5*rm1)/dy
        r(inds,i) = r(inds,i) + cj*(0.5*rdp1 - 0.5*rdm1)/dy
    end
endfunction

function r=resIV(ind)
    global mockY y t
    r=resI(t, mockY, ydot, ind)
endfunction

function r=resdIV(ind, t)
    global mockYdot ydot t
    r=resI(t, y, mockYdot, ind)
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

function u = savedUV(ind)
    [xind,yind] = toCoord(ind)
    u = savedU(xind,yind)
endfunction

function r = fjacV(y)
    global coeffs t0 dy

    inds = nonZeroInds(1)
    coeffs(1) = y(1)+dy
    rp1 = feval(inds, savedUV)
    coeffs(1) = y(1)-dy
    rm1 = feval(inds, savedUV)
```

```
    r(inds,1) = (0.5*rp1 - 0.5*rm1)/dy

    for i = 2:ncpts
        inds = nonZeroInds(i)
        coeffs(i-1) = y(i-1)
        coeffs(i) = y(i)+dy
        rp1 = feval(inds, savedUV)
        coeffs(i) = y(i)-dy
        rm1 = feval(inds, savedUV)
        r(inds,i) = (0.5*rp1 - 0.5*rm1)/dy
    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

function r = resFV(x, y)
    global t
    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 [coeffs,r] = initCoeffs(iCoeffs)
    if argn(2) == 0 then
        iniCoeffs()
        r = atol;
    else
        coeffs = iCoeffs
        r = max(abs(fsol(iCoeffs)))

    end
    //coeffs = ones(ncpts, 1)
    info = 4
    if r < atol then
            disp("Reusing last initial coeffs.")
            return
    elseif jacMode == 0 then
        [coeffs, n, info] = fsolve(coeffs, fsolV, atol)
    elseif jacMode == 1 then
        [coeffs, n, info] = fsolve(coeffs, fsolV, fjacV, atol)
    end

    rm1 = r;
    r = max(abs(fsol(coeffs)))

    if verbose > 1 then
        disp("fsolve info: " + string(info))
        disp("fsolve residual: " + string(r))
    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


function c = fsolV(coefs)
    // Set coefs to global coeffs
    global coeffs
    coeffs = coefs

    evalX = []
    evalY = []
    if (N~=M || p~=q) then
        disp("fsolV only works with N=M and p=q")
```

```
            abort
            return
        end
        evalX(1) = A
        evalY(1) = C
        ind = 1
        for i = 1:N
            for j = 1:p-1
                ind = ind + 1
                evalX(ind) = colX(i, j)
                evalY(ind) = colY(i, j)
            end
        end
        evalX(ncptX) = B
        evalY(ncptY) = D
        tru = feval(evalX, evalY, uinit)


        // Set x = A first
        c = []
        tru = tru - feval(1:ncptX, 1:ncptY, savedU)
        for i = 1:ncptX
            c(((i-1)*ncptY+1:(i*ncptY))) = tru(i,:)'
        end
    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] = resV(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)
    if argn(2) == 0 then
        fg = zeros(ncpts, 1)
    end
   y0p = fg
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
    //tic
```

```
    ptDer = getPtDer(up)
    coeffs = u

    // 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
   // time=toc()
    //disp(string(time) + " seconds spent in res.")
endfunction

function [c, ires] = resV(tin, u, up)
    //tic
    global coeffs t
    t = tin
    ptDer = getPtDer(up)
    coeffs = u

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

    c((1:(ncptX - 2))*ncptY+1) = feval(2:(ncptX - 1), t, resBndyc)'

    //colRes = feval(2:(ncptX - 1), 2:(ncptY - 1), resFV)

    c((2:(ncptX - 1))*ncptY) = feval(2:(ncptX - 1), t, resBndyd)'

    // Fill col pt values
    for i = 1:(ncptY - 2)
        c(i*ncptX+(2:(ncptX - 1))) = feval(i+1,2:ncptX-1, resFV) - ptDer
            (i*ncptX+(2:(ncptX - 1)))';
    end

    // Loop through boundary conditions on x = B
    c((ncptX-1)*ncptY+(1:ncptY)) = feval(1:ncptY, t, resBndxb)
    ires = 0
    //[cc, ires] = res(tin, u, up)
    //disp(max(cc-c))
    //time=toc()
```

```
    //disp(string(time) + " seconds spent in resV.")
endfunction

function [c, ires] = res2(t, u, up)
    global coeffs

    ptDer = getPtDer(up)
    coeffs = u

    clear evalX evalY
    evalX(1) = A
    evalX(ncptX) = B
    evalY(1) = C
    evalY(ncptY) = D
    ind = 2
    if ncptX == ncptY then
        for i = 1:N
            for j = 1:p-1
                evalX(ind) = colX(i, j)
                evalY(ind) = colY(i, j)
            end
        end
    else

        for i = 1:N
            for j = 1:p-1
                evalX(ind) = colX(i, j)

            end
        end
        ind = 2
        for i = 1:M
            for j = 1:q-1
                evalY(ind) = colY(i, j)
            end
        end
    end

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

    // Loop through the vertical lines
    for i = 2:(ncptX - 1)
        c((i-1)*ncptY+1) = bndyc(evalX(i), t)
        for j = 2:(ncptY - 1)
            c((i-1)*ncptY+j) = resF(t, i, j) - ptDer((i-1)*ncptY+j)
            f(t, evalX(i), evalY(j))
            c((i-1)*ncptY+j) =  - ptDer((i-1)*ncptY+j)
        end
        c(i*ncptY) = bndyd(evalX(i), t)
    end
```

```
    // Loop through boundary conditions on x = B
    for i = 1:ncptY
        c((ncptX-1)*ncptY+i) = bndxb(evalY(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)
global eps
eps = 0.1
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)
    case 3 then
        p = U(x,y,3,1) + U(x,y,1,3) + (cos(x+y)+cos(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)
    case 3 then
        p = savedU(xInd,yInd,3,1)+savedU(xInd,yInd,1,3)+(cos(x+y)+cos(x-
            y))
    end
endfunction

// Boundary condition along x = A
function z = bndA(y)
    z= truu(A, y)
    return
    select probNum
    case 1 then
        z = 0
    case 2 then
        z = 1
    case 3 then
        z = cos(y)
    end
endfunction

// Boundary condition along x = B
function z = bndB(y)
    z = truu(B, y)
    return
    select probNum
    case 1 then
        z = 2 * %e^y
    case 2 then
        z = %e^(2 * y)
    case 3 then
        z = -cos(y)
    end
endfunction

// Boundary condition along y = C
function z = bndC(x)
    z = truu(x, C)
    return
    select probNum
    case 1 then
        z = x
    case 2 then
        z = 1
    case 3 then
        z = cos(x)
    end
endfunction

// Boundary condition along y = D
function z = bndD(x)
    z = truu(x, D)
    return
    select probNum
```

```
    case 1 then
        z = %e * x
    case 2 then
        z = %e^x
    case 3 then
        z = 0
    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 = 1
        C = 0
        D = 1
    case 2 then
        A = 0
        B = 1
        C = 0
        D = 1
    case 3 then
        A = 0
        B = %pi
        C = 0
        D = %pi
    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)
    case 3 then
        z = cos(x) * cos(y)
    end
endfunction

// The true solution
function z = trudx(x, y)
    select probNum
    case 1 then
        z = %e^y
```

```
    case 2 then
        z = y * %e^(x * y)
    case 3 then
        z = -sin(x) * cos(y)
    end
endfunction

// The true solution
function z = trudy(x, y)
    select probNum
    case 1 then
        z = x * %e^y
    case 2 then
        z = x* %e^(x * y)
    case 3 then
        z = cos(x) * -sin(y)
    end
endfunction

function z = trudxy(x, y)
    select probNum
    case 1 then
        z = %e^y
    case 2 then
        z = (x*y+1)* %e^(x * y)
    case 3 then
        z = sin(x)*sin(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)
    case 3 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, t, 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

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

function z = trudx(x, y)
    z = trudxt(x, y, tc)
endfunction
function z = trudxt(x, y, t)
    select probNum
    case 1 then
        z = cos((%pi*x)/2)*sin((%pi*y)/2)*(%pi/2)*%e^-((%pi^2/2)*eps*t)
    case 2 then
        z = -(%e^((-t+x+y)/(2*eps)))/((2*eps)*(%e^((-t+x+y)/(2*eps))+1)
            ^2)
    end
endfunction

function z = trudy(x, y)
    z = trudyt(x, y, tc)
endfunction
function z = trudyt(x, y, t)
    select probNum
    case 1 then
        z = sin((%pi*x)/2)*cos((%pi*y)/2)*(%pi/2)*%e^-((%pi^2/2)*eps*t)
    case 2 then
        z = -(%e^((-t+x+y)/(2*eps)))/((2*eps)*(%e^((-t+x+y)/(2*eps))+1)
            ^2)
    end
endfunction


function z = trudxy(x, y)
    z = trudxyt(x, y, tc)
endfunction
function z = trudxyt(x, y, t)
    select probNum
    case 1 then
        z =  cos((%pi*x)/2) * cos((%pi*y)/2) * (1/4)* %pi^2 * %e^-((1/2)
            * %pi^2 * eps * t)
    case 2 then

        z = ((50*%e^(10*(-t+x+y)))/((%e^(5*(-t+x+y))+1)^3)) - ((25*%e
            ^(5*(-t+x+y)))/((%e^(5*(-t+x+y))+1)^2))
    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=[]//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 = evalHBIlobbed(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 -- Does this cause the wrong points to be used for
           the derivatives?
        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

function z = evalHBI(x, y, sx, sy, wx, wy, Usx, Usy, Usw, Usxy)

    // 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

// Cross derivative terms
z = z + Hbarx(1) * Hbary(1) * Usxy(1,1)
z = z + Hbarx(1) * Hbary(2) * Usxy(1,2)
z = z + Hbarx(2) * Hbary(1) * Usxy(2,1)
z = z + Hbarx(2) * Hbary(2) * Usxy(2,2)

// X-derivative terms
z = z + Hbarx(1) * Hy(1) * Usx(1, 1)
z = z + Hbarx(2) * Hy(1) * Usx(2, 1)
for i = 1:numWy
    z = z + Hbarx(1) * Gy(i) * Usx(1, i+1)
    z = z + Hbarx(2) * Gy(i) * Usx(2, i+1)
end
z = z + Hbarx(1) * Hy(2) * Usx(1, numWy+2)
z = z + Hbarx(2) * Hy(2) * Usx(2, numWy+2)


// Y-derivative terms
z = z + Hbary(1) * Hx(1) * Usy(1, 1)
z = z + Hbary(2) * Hx(1) * Usy(2, 1)
for i = 1:numWx
    z = z + Hbary(1) * Gx(i) * Usy(1, i+1)
    z = z + Hbary(2) * Gx(i) * Usy(2, i+1)
end
z = z + Hbary(1) * Hx(2) * Usy(1, numWx+2)
z = z + Hbary(2) * Hx(2) * Usy(2, numWx+2)

// Solution value terms
z = z + Hx(1) * Hy(1) * Usw(1, 1)
z = z + Hx(1) * Hy(2) * Usw(1, numWx+2)
z = z + Hx(2) * Hy(1) * Usw(numWy+2, 1)
z = z + Hx(2) * Hy(2) * Usw(numWy+2, numWx+2)
for i = 1:numWy
    z  = z + Hx(1) * Gy(i) * Usw(1, i+1)
    z  = z + Hx(2) * Gy(i) * Usw(numWy+2, i+1)
end
for i = 1:numWx
    z  = z + Hy(1) * Gx(i) * Usw(i+1, 1)
    z  = z + Hy(2) * Gx(i) * Usw(i+1, numWx+2)
end
for i = 1:numWx
    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)
    ptsX = linspace(meshX(ix), meshX(ix+1), 7)
    ptsY = linspace(meshY(iy), meshY(iy+1), 7)
    e = max(abs(feval(ptsX, ptsY, func) - feval(ptsX, ptsY, U)))
    if hScale then
        e = e * (meshX(ix+1)-meshX(ix)) * (meshY(iy+1)-meshY(iy))
    end
endfunction
function e = evalRectErr2(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 * (xH-xL) * (yH-yL)
    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, Usxy] = saveFunctionEvals(sx, sy, wx, wy)
    pX = [sx(1); wx; sx(2)]
    pY = [sy(1); wy; sy(2)]
    Usx = feval(sx, pY, Ux)
    Usy = feval(pX, sy, Uy)
    Usw = feval(pX, pY, U)
    Usxy = feval(sx, sy, Uxy)
endfunction
function [Usx, Usy, Usw, Usxy] = saveFunctionEvals2(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

    // Cross-Derivatives
    Usxy(1,1) = U(sx(1), sy(1), 2, 2)
    Usxy(1,2) = U(sx(1), sy(2), 2, 2)
    Usxy(2,1) = U(sx(2), sy(1), 2, 2)
    Usxy(2,2) = U(sx(2), sy(2), 2, 2)


    // 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, Usxy] = saveFunctionEvals(sx, sy, wx, wy)
    Usy = Usy'
    z =  evalHBI(x, y, sx, sy, wx, wy, Usx, Usy, Usw, Usxy)
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, Usxy] = saveFunctionEvals(sx, sy, wx, wy)
    Usy = Usy'
    z =  evalHBI(x, y, sx, sy, wx, wy, Usx, Usy, Usw, Usxy)
endfunction

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

function e = rLoiErr(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, %F)
endfunction

// Returns the global error estimate from the LOI
function e = gLoiErrS()
    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, %F, xi, yi)
endfunction

function e = rL2ErrS(xi, yi)
    e = evalRectErr(l2, p+1, %T, xi, yi)
endfunction

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

// Returns the global error estimate from the L2
function e = gL2ErrS()
    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
```