A Sixth-Order Extension to the MATLAB bvp4c Software of J. Kierzenka and L. Shampine.

N. P. HALE
Department of Mathematics,
Imperial College London.
June 2006.

Supervised by D.R.Moore

Submitted for contribution in the degree of
Master of Science in Mathematics.

Correspondance address;
Nicholas Hale, 9 Applesham Way, Portslade, East Sussex, BN41 2LQ, UK.
nicholas.hale@hotmail.co.uk

Abstract

The aim of this project is to develop the existing MATLAB BVP ODE solver bvp4c ([1] 2001) into a sixth-order accurate MIRK solver named bvp6c.

This new solver will maintain the well established framework of the existing residual control based software although updated to use the sixth-order Cash-Singhal ([2] 1980) difference scheme and the recently developed Cash-Moore ([5] 2003) MIRK6 interpolant.

It shall be demonstrated that this new software will improve upon both the accuracy and efficiency of the current MATLAB ODE package whilst maintaining the generality and robustness of the original algorithm.

The new software shall be submitted to The Mathworks, Inc - the producers of the MATLAB environment, and ACM - the journal in which bvp4c was introduced, along with a research paper justifying the development of bvp6c.

A draft copy of this paper will be included within this report and although coauthored with D.R.Moore, only those sections of which were solely contributed by N.Hale are included.

Acknowledgements

I would like to take this opportunity to thank my supervisor Daniel Moore for his continued and invaluable support throughout this project and my undergraduate degree. Dan was always on hand to give sound advice of both a conceptual and practical nature as well as motivation to produce what I hope the reader will find a concise report and a useful mathematical tool.

In addition, a great amount of credit should be given to Jacek Kierzenka of The Mathworks and Lawrence Shampine from the Southern Methodist University. These two gentlemen were responsible for pioneering the development and implementation of the bvp4c software, without which this project would have had no basis.

Further Notice

The bvp6c software is available online in the MATLAB Central File Exchange and is also hosted at http://www.ma.ic.ac.uk/d̃rmii/bvp6c

Contents

CHAPTER 1

Introduction

The study of the ordinary differential equation (ODE) dates as far back as G.W.Leibniz and Isaac Newton in the seventeenth century. Although in recent times giving some headway to the study of partial differential equations (PDES), the understanding and solution of the ODE is a vital mathematical process. Indeed a large number of physical phenomena arising from mathematics, engineering, physics and the other sciences are describable through such equations.

The motion of planets, nonlinear optics, oceanography, meteorology, projectiles and population dynamics are but a few examples. Furthermore a common technique of solving partial differential equations to reduce them to ordinary equations by suitable spatial discretizations. Since the study of ODEs is much more developed and understood field, solutions to these problems can then be found.

Of particular discussion in this work are boundary value problems (BVPs) whereby in addition to the differential equation, further conditions (BCS) are imposed on the solution at a number of boundary points. It shall be shown later that these conditions can have a dramatic affect on the form of the solution, indeed in a number of cases meaning the difference between a unique or infinitely many solutions, or no solution at all.

While there are a number of general techniques for analytically solving some classes of these ODEs, the only practical technique for many complicated systems is the use of numerical methods. Given the vastness of potential problems it is important then to develop a highly accurate and efficient solver capable of solving a wide variety BVPs. Furthermore the frequent occurrence of ODEs in engineering and industrial applications mean that they must often be solved by a user with limited computational or mathematical knowledge. There is a need then for a general purpose ODE BVP solver to be implemented by an inexperienced user and give fast, accurate solutions to a wide range of problems.

The MATLAB computing environment is a package used extensively throughout industry, research and education by users of a complete range in proficiency. MATLAB provides then an ideal platform to introduce such ann item of BVP software and indeed, Kierzenka and Shampine [1] developed the core BVP ODE software bvp4c to solve a large class of two-point boundary value problems of the form;

$$\begin{aligned} \underline{y}'(\underline{x}) &= \underline{f}(x, \underline{y}(x), \underline{p}) \\ g(x_L, x_R, \underline{y}(x_L), \underline{y}(x_R), \underline{p}) &= 0 \end{aligned}$$

where f is continuous and Lipschitz function in y and p is a vector of unknown parameters.

Their view was that a user solving a BVP of this form in MATLAB would be most interested in the graphical representation of a solution, and as such a modest-order solver would be appropriate for graphical accuracy. However the author believes that whilst a fourth-order solver is acceptable, recent developments mean that a sixth-order solver would supply not only greater accuracy but perform also more efficiently whilst maintaining the generality of the existing software.

This project will contain the following;

- An introduction to ODEs and Finite Difference solvers,
- A dissection of the bvp4c software,
- An account of the necessary developments for a sixth-order extension,
- An overview of the implementation of the new software,
- Testing and comparison between the two solvers on a variety of problems.

Once it has been determined that the new software compares favourably to that of the existing MATLAB ODE package, it is hoped that it shall be included in future releases of the MATLAB environment,

CHAPTER 2

ODES and Finite Differences

What is an ODE ?

Definition 2.0.1. Ordinary Differential Equation

An Ordinary Differential Equation is an equality involving a function of one variable and its derivatives with respect to that variable. An ODE of order n is an equation of the form

$$F(x, y, y', \dots, y^{(n)}) = 0$$

where y is a function of x, and $y^{(n)}$ is the n^{th} derivative with respect to x. A basic example of this is the equation describing simple harmonic motion,

$$y'' + \omega y = 0.$$

Many physical systems undergoing small displacements obey SHM, such as a pendulum, a mass on a spring and the flow of current in an electrical circuit.

Closely related is a system of ODES where the equations are coupled. In this instance a number of equations F_i involving functions y_j and their derivatives must be satisfied, i.e

$$\begin{aligned} F_1(x, y_1, y_1' \ldots, y_2, y_2', \ldots) &= 0 \\ F_2(x, y_1, y_1' \ldots, y_2, y_2', \ldots) &= 0, etc. \end{aligned}$$

The order of such a system is given by summing the degree of the highest derivatives of each of the functions appearing in the equations. An example of second-order system is the Lotka-Voltera equations which describe the population dynamics of a predator-prey biological system,

$$\begin{aligned} y'(x) &= y(x)(\alpha - \beta z(x)) \\ z'(x) &= -z(x)(\gamma - \delta y(x)). \end{aligned}$$

In general, an n^{th} order ODE system will have n linearly independent solutions and in order to impose uniqueness, further conditions must be specified on the solution. Conditions fall into two main categories - Boundary Conditions (BCs) and Initial Conditions (ICs) and give rise to Boundary Value Problems (BVPs) and Initial Value Problems (IVPs).

2.1 Reduction of Order - Canonical Form

It is convenient when solving an ODE system numerically to describe the problem in terms of a system of first-order equations. For example when solving an n^{th}-order problem numerically is common practice to reduce the equation to a system of n first-order equations, i.e.

$$\underline{y}' = \underline{f}(x, \underline{y}(x)). \tag{2.1}$$

Consider the following very simple problem, which shall be used throughout as a basic working example.

$$y''(x) = -\lambda^2 y(x) \tag{2.2}$$

$$\tag{2.3}$$

Then, by defining $y_1 = y(x), \quad y_2 = y'(x)$

$$\underline{y} = \begin{pmatrix} y(x) \\ y'(x) \end{pmatrix} \quad \text{and} \quad \underline{y}' = \underline{f}(x, \underline{y}) = \begin{pmatrix} y_2 \\ -\lambda^2 y_1 \end{pmatrix}.$$

A similar argument may be used to represent any given n^{th}-order problem, or a coupled ODE system of higher-order to the form (2.1).

In most cases a general code used to solve a system in this form will be less efficient than a code written explicitly to solve a given problem. However, the focus here is on creating a generic solver able to solve a large class of problems irrespective of their original representation and also considering problems in this form allows a much more unified analysis of the existence and the uniqueness of solutions (see Keller [7]). There are however a subclass of problems reducible to a system of second-order systems of the form $\underline{y}'' = \underline{f}(x, \underline{y})$, and some important work such as [17], [18] into algorithms which can compute solutions to problems of this form far more efficiently. However these methods are not yet the subject of this work.

Note that for brevity the underline vector notation, the x dependance of $\underline{y}(x)$ and the $x, \underline{y}(x)$ dependance of $\underline{f}(x, \underline{y}(x))$ is usually dropped and hence the system (2.1) is denoted $y' = f(y)$.

2.2 IVPS

Using the notation given above, an IVP will take the form

$$y' = f(x, y) \tag{2.4}$$

$$y(x_0) = \underline{\eta}_0. \tag{2.5}$$

That is the conditions imposed on the function $y(x)$ are specified at one particular position x_0. Frequently x will correspond to time and η will correspond to the known initial position, velocity, etc of the solution - hence the term Initial Value Problem.

Although this project is particularly concerned with BVPS, the initial value form of a problem can be insightful as to the behaviour of the corresponding boundary problem and in some ways can be thought of as a boundary problem with only one boundary. Indeed there are are a family of numerical routines known as *shooting methods* which make use of this property. As such there may be some similarities (and indeed some important differences) in their analysis, which is why they merit discussion.

Definition 2.2.1. Lipschitz

Consider the region $\mathcal{D}= \{(x, y) : x \in [a, b], y \in [-\infty, \infty]\}$ with a, b finite and let $f(x, y)$ be well defined and continuous on $\mathcal{D}$. The function f is called **Lipschitz** in y iff there exists constant L st $\forall x, y_1, y_2$ where $(x, y_1), (x, y_2) \in\mathcal{D}$,

$$|f(x, y_1) - f(x, y_2)| \leq L|y_1 - y_2|$$

Theorem 2.2.2. PicardLindelf

If $y' = f(x, y)$ where f is continuous on $[a, b] \subset\mathcal{R}$ and Lipschitz on $y \in\mathcal{R}$ then there exists a unique solution y to (2.5) where y is continuous and differentiable $\forall x, y \in\mathcal{D}$.

Proof. A simple proof of this theorem [13] proceeds as follows;

$$\begin{aligned}\Psi_0(x) &= y_0 \\ \Psi_{i+1}(x) &= y_0 + \int_{x_0}^{x} f(t, y\Psi_i(t))dt\end{aligned}$$

Then Banach Fixed Point theorem will show that the sequence of iterates Ψ_i are convergent and that the limit of these is a solution to the problem. Furthermore, an application of Grnwall's lemma will show this solution to be unique.

This definition and theorem extends naturally to higher dimensions and hence justifies the development of solvers for the IVP formulation of (2.5).

2.3 BVPs

If information is specified at more than one point the problem (2.1) becomes a Boundary Value Problem. The most common types of BVP and those which are consider in this project are those for which information given at precisely two points. These are known as **two-point boundary value problems.**

For a Two-Point BVP the boundary data takes the form

$$g(y(x_L), y(x_R)) = 0 \tag{2.6}$$

of which the most general form is

$$g_L(y(x_L)) + g_R(y(x_R)) = \xi$$

known as a **non-separated** condition. However, if (2.6) may be reduced to

$$\begin{aligned}g_L(y(x_L)) &= \xi_L \\ g_R(y(x_R)) &= \xi_R\end{aligned}$$

then these are referred to as **separated** conditions.

Whilst separated conditions are more convenient from both a theoretical and practical viewpoint, it may impossible or at least nontrivial to reduce (2.6) to this form without the introduction of an additional variable to the ODE system. This should be avoided where possible, since increasing the size of the system will naturally increase the time needed to solve it.

In addition, the jump from an IVP to a BVP is more profound than it may at first seem and indeed, theories governing the existence and uniqueness a solution are far less general. Consider the following two-point BVP version of example (2.2).

$$y''(x) = -\pi^2 y(x)$$
$$y(0) = 0,\ y(1) = c.$$

If the constant $c \neq 0$ then the above system has no solution, whereas for $c = 0$ the function $y(x) = A\sin(\pi x)$ is a solution for any real valued A. The corresponding IVP with $y(0) = 0$, $y'(0) = c$. will always have a solution $y(x) = c\sin(\pi x)/\pi$.

It can be shown to that for a first-order system of size n it is necessary (although not sufficient) to supply n linearly independent BCs.

2.4 Finite Difference Schemes

Numerical methods for solving BVPs generally fall into one of two categories. Shooting methods reduce the two-point BVP to that of an initial value problem. The solver will start at either end of the solution (say x_L) integral and ensure that the boundary conditions here are met. Here some assumptions will be made as to the initial value of those elements which are not specified by the BVP BCs, then the algorithm will then integrate or *shoot* across the interval using an IVP solver until it reaches x_R. If this IVP solution obeys the BCs at the x_R boundary, then it is also a solution to the BVP. If not then the solution is used to refine the guess of the initial values and shooting is repeated until all boundary conditions are satisfied. Of course the decision to begin shooting from the x_L was entirely arbitrary and it is equally possible to start from the right. This gives rise to the *direction of integration.*

The second approach is that of finite differences and the underlying principal of any finite difference scheme is the division of the interval $[x_L, x_R]$ into a finite grid or *mesh* of points $x_L = x_1 < x_2 ... < x_N = x_R$. The scheme will approximate the derivative and function values based upon information from each of the intervals $[x_i, x_{i+1}]$ and n important difference between shooting and finite difference methods is that with the latter here is no direction of integration, i,e, the solution is obtained globally over the entire mesh.

Define

$$\begin{aligned} \underline{x} &= [x_0, x_1, ..., x_N]^T \\ Y_i &= \underline{y}(x_i) \\ \underline{Y} &= [Y_0, Y_1, ..., Y_N]^T \\ h_i &= x_{i+1} - x_i \end{aligned}$$

<u>One-step schemes</u>

An approximation is sought to the solution of $\underline{y}' = \underline{f}(x, \underline{y})$ on such a mesh and a one-step scheme will do so using data from only two mesh points For example the derivative y' is approximated by the forward difference

$$\underline{y}'\big|_{x_i} = \frac{Y_{i+1} - Y_i}{h_i}$$

This estimate is centred around $x_{i+\frac{1}{2}} = (x_{i+1} - x_i)/2$ and a given one-step finite difference method is defined via a function $\phi_i = \phi(x_i, x_{i+1}, Y_i, Y_{i+1}, h_i)$ which will approximate $\underline{f}(x_{i+\frac{1}{2}}, y_{i+\frac{1}{2}})$ and remove some of the error introduced in the crude estimate of the derivative. The derivation of such functions ϕ is treated lightly here since they are not the topic of investigation in this project. However it should be noted these functions which define a given scheme are the fundamental components of finite difference methods - over the last thirty years have been the focus of great deal of research and a large number of texts within numerical analysis. For further information the reader is directed to Lambert [10], Wendroff [8] and Keller [7] to name but a few items which treat this topic in more depth. Indeed for specific information on the finite difference scheme intended for use in the sixth-order solver introduced in this project can be found in Cash and Singhal [2].

Definition 2.4.1. A finite difference scheme is said to be of **order p** if

$$\underline{y}(x_{i+1}) - \underline{y}(x_i) - h_i\phi_i = \mathcal{O}(h_i^{p+1})$$

and the scheme is **consistent** if

$$\phi_i \to f(x_i, \underline{y}(x_i)) \,\text{as}\, h_i \to 0$$

One-step schemes are useful because all the information required by the scheme is specific to each subinterval $[x_i, x_{i+1}]$. From a practical viewpoint this is beneficial as the algorithm can considering only one interval at a time, working independently of other intervals. Also from the perspective of error estimation and control this local independence of the solver hugely simplifies the analysis. Mono-Implicit Runge-Kutta (MIRK) and Collocation are two forms of such one-step difference schemes. A MIRK method is a particular family of which whereby the function ϕ is defined in the form

$$y_{i+1} = y_i + h_i \sum_{i=1}^{s} \beta_i k_i$$

where

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f(x_i + c_2 h_i, y_i + a_{21} h_i k_1) \\ k_3 &= f(x_i + c_3 h_1, y_i + a31 h_i k_1 + a32 h_i k_2), etc. \end{aligned}$$

The constants $\{c_i\}$ and $\{a_{ij}\}$ are chosen to reduce lower order error terms in the Taylor series expansion of f and the scheme can be shown to be consistent if they satisfy relation $\sum_{j=1}^{i-1} a_{ij} = c_i$. Again more information on the derivation of these formula is available in the above texts.

A Collocation method on the other hand approximates the solution $y(x)$ over the $[x_L, x_R]$ by a function piecewise polynomial over the each subinterval $[x_i, x_{i+1}]$. This function will satisfy the ODE at selected *collocation points* within each interval and will take values equal to an approximation y_i at each of the mesh points. Specifically a collocation polynomial $S(x)$ of degree s will satisfy

$$\begin{aligned} S(x_i) &= y_i \\ S'(x_i + c_k h_i) &= f(x_i + c_k h_, S(x_i + c_k h_i)), \quad k = 1, \ldots, s \end{aligned}$$

for some $\{c_k \in (0,1]\}_{k=1}^{s}$, Solving the resulting $s+1$ equations then uniquely determines the $s+1$ coefficients of the polynomial $S(x)$. One of advantage of collocation methods is that they provide a continuous approximation to the solution whereas many other numerical methods produce only a table of values of the approximate solution at discrete points. This said however, Keller ([7] 1976) shows in some generality than a $\mathcal{C}^0$ s-degree collocation approximation is equivalent to an s-stage MIRK method with a continuous interplant and for this reason the terms MIRK and Collocation can, in the correct context, be used interchangeably.

Define then

$$\begin{aligned}\Phi_0(x_0, x_N, Y_0, Y_N) &= g(x_0, x_N, Y_0, Y_N, p) && (2.7)\\ \Phi_i(x_i, x_{i+1}, Y_i, Y_{i+1}) &= Y_{i+1} - Y_i - h_i\phi(x_i, x_{i+1}, Y_i, Y_{i+1}, h_i) && (2.8)\end{aligned}$$

These are known as the **collocation** equations of the finite difference scheme, and the corresponding algorithm will seek to solve $\Phi(\underline{x}, \underline{Y}) = \underline{0}$.

2.5 Newton's Method

Suppose that the algorithm is given an initial approximation $\underline{Z}$ to the solution $\underline{Y}$ such that $\Phi(\underline{X}, \underline{Z}) \neq \underline{0}$. The task is then to find the **Newton direction** $\underline{\delta Z}$ such that

$$\underline{Z} + \underline{\delta Z} = \underline{Y}$$

Assuming that $\underline{Z}$ is close to $\underline{Y}$ so that $\underline{\delta Z}$ is sufficiently small, then the following is valid

$$\underline{0} = \Phi(\underline{x}, \underline{Y}) = \Phi(\underline{x}, \underline{Z} + \underline{\delta Z}) \approx \Phi(\underline{x}, \underline{Z}) + \left.\frac{\partial \Phi}{\partial Z}\right|_{\underline{x},\underline{Z}} \underline{\delta Z}$$

Hence $\underline{\delta Z}$ can be approximated by the solution to the matrix equation

$$\left.\frac{\partial \Phi}{\partial Z}\right|_{\underline{x},\underline{Z}} \underline{\delta Z} = -\Phi(\underline{x}, \underline{Z}) \qquad (2.9)$$

For a one-step scheme the value of Φ at one mesh point is dependant only on data at the current and preceding points and so for each i from 1 to N by denoting here Φ_{i+1} by Φ and removing the x dependance for brevity

$$0 = \Phi(\underline{Y}) \equiv \Phi(Y_i, Y_{i+1}) = \Phi(Z_i + \delta Z_i, Z_{i+1} + \delta Z_{i+1}) \approx \Phi(Z_i, Z_{i+1}) + \left.\frac{\partial \Phi_{i+1}}{\partial Z_i}\right|_{Z_i, Z_{i+1}} \delta Z_i + \left.\frac{\partial \Phi}{\partial Z_{i+1}}\right|_{Z_i, Z_{i+1}} \delta Z_{i+1}$$

A similar result must be shown for the boundary collocation function Φ_0, and indeed

$$0 = \Phi_0(Y_0, Y_N) \equiv g(Y_0, Y_N) = g(Z_0 + \delta Z_0, Z_N + \delta Z_N) \approx g(Z_0, Z_N) + \left.\frac{\partial g}{\partial Z_0}\right|_{Z_0, N_N} \delta Z_0 + \left.\frac{\partial g}{\partial Z_0}\right|_{Z_0, N_N} \delta Z_N$$

The matrix system from (2.9) then has the form

$$\begin{bmatrix} \frac{\partial g}{\partial Z_0} & 0 & \cdots & \frac{\partial g}{\partial Z_N} \\ \frac{\partial \Phi}{\partial Z_0} & \frac{\partial \Phi}{\partial Z_1} & \ddots & 0 \\ 0 & \ddots & \ddots & 0 \\ \vdots & 0 & \frac{\partial \Phi}{\partial Z_{N-1}} & \frac{\partial \Phi}{\partial Z_N} \end{bmatrix} \begin{bmatrix} \delta Z_0 \\ \delta Z_1 \\ \vdots \\ \delta Z_N \end{bmatrix} = - \begin{bmatrix} g(Z_0, Z_N) \\ \Phi_1(Z_0, Z_1) \\ \vdots \\ \Phi_N(Z_{N-1}, Z_N) \end{bmatrix} \tag{2.10}$$

However in order to solve this system we must first compute the $\partial \Phi i + 1/\partial Z_i$ and the $\partial \Phi i + 1/\partial Z_{i+1}$ - known as the **Global Jacobians** of the system (2.7) - and also the partial derivatives of the boundary condition function. Although the Global Jacobians are specific to the definition of Φ, they will take the form of a function in terms like (for $\omega \in [0, 1]$)

$$\left. \frac{\partial f}{\partial Y} \right|_{Y_{i+\omega}} \tag{2.11}$$

Such values are known as **Local Jacobians** and may be computed either analytically or numerically using finite differences. For a general non-linear f, the derivative in (2.11) is found by perturbing Y in the Y_i direction and dividing by the size of this shift, i.e

$$\frac{\partial f}{\partial Y_i} \approx \left[f(Y_1, Y_2, \ldots, Y_i + \delta, \ldots, Y_N) - f(Y_1, Y_2, \ldots, Y_i, \ldots, Y_N) \right] / \delta.$$

From a practical viewpoint δ is generally taken as $sqrt(eps) * range(Y_i(x))$ where eps is the largest number such that $1 + eps = 1$ in floating point. Other more accurate formulae are available, but will be even slower than the above which is itself an expensive computational process. It shall be shown later that the use of analytic Jacobians or indeed their judicious approximation can hugely reduce run times for particularly malicious righthand sides.

Solving the Matrix System

Once the Jacobian Matrix of (2.9) has been computed, the next task is to solve the system (2.10). Solving matrix equations is something that MATLAB does very well and with a variety of different methods, one of which being the application an **LU** factorisation. The

MATLAB command $[L, U, P] = \mathbf{lu}(A)$ returns an upper triangular matrix in U, a lower triangular matrix with a unit diagonal in L and a permutation matrix in P such that $LU = PA$. Solving a matrix equation $Ax = b$ is then reduced to solving $LUx = Pb$ and hence the two triangular systems

$$\begin{aligned} Ly &= Pb \\ Ux &= y \end{aligned}$$

The advantage of this is that although computation of LU is less efficient than solving $Ax = b$ directly, if the same system must later be solved with a different RHS, (say $Ax = c$) then the two corresponding triangular systems are already formed and are easy to solve. Additionally, the matrix from (2.10) is clearly sparse (i.e. having a proportionately small number of non zero elements) and many of the built-in MATLAB routines including *lu* can optimised to take advantage of this sparsity.

Line Searching

Once the Newton direction $\underline{\delta Z}$ has been computed, the next step is to define the new approximation to the solution $\underline{Z}^{[new]} = \underline{Z} + \lambda\underline{\delta Z}$. For a sufficiently smooth problems a *full* newton step $\lambda = 1$ is be taken, but for problems which are less well behaved it is good practice to search for the optimal reduction in the newton direction. This is obtained via a weak line search.

In optimisation a weak line search is a way of minimising the value of an objective function $f : \mathcal{R}^n \rightarrow \mathcal{R}$ at a point x_i in descent direction ρ_i by asking for a sufficient decrease in the function $\psi(\alpha) \equiv f(x_i + \alpha\rho_i)$. Once value of α_i has been determined, the next iteration point is defined by $x_{i+1} = x_i + \alpha_i\rho_i$ and the process repeated with a new descent direction.

Recall that the vector $\Phi(\underline{x}, \underline{Z})$ is the error by which the current approximation satisfies the collocation system (2.7). The objective function here then is the norm of the solution to the matrix equation (2.10) with the RHS defined by the new tentative approximation $\underline{Z} + \lambda\underline{\delta Z}$. The line search will ensure that this shows a sufficient reduction when compared to the norm of the full Newton direction $\underline{\delta Z}$. If this is not achieved then the value λ is reduced by a factor 1/2 until it such a reduction is achieved.

Finally once an acceptable step length λ for the Newton direction $\underline{\delta Z}$ has been found, the next approximation $\underline{Z}^{[1]} = \underline{Z} + \lambda\underline{\delta Z}$ is defined. The entire process above is then repeated iteratively and since at each stage the size of $\underline{\delta Z^{[k]}} = \underline{Y} - \underline{Z^{[k]}}$ is reduced, the approximations $Z^{[k]}$ will approach the true solution $\underline{Y}$ (Keller [7]).

2.6 Additional notes

Initial Guesses

In the previous section the expansions of the collocation function Φ were based upon the assumption that $\underline{\delta Z}$, the difference between the current iterative guess and the solution to the posed problem at each of the mesh points x_i was sufficiently small. For a poorly defined, inaccurate in initial guess this vector can be large enough to cause this analysis to break down and prevent the routine from converging to the true solution. On the other hand, constructing an accurate guess can be difficult when the form of the solution is not known in addition to being very time consuming to produce and in many cases may unnecessary.

If a crude guess applied to a problem will not converge then there are a number of alternatives strategies for forming an accurate guess. One such method is known as continuation and an example of this is shown later in chapter 5. Continuation involves solving a simpler problem, similar too but better-behaved than the one posed, for which such an accurate initial guess is not necessary. The solution from this alternative problem is then used as the initial guess for the one required and is in most cases sufficiently accurate to allow convergence of the Newton method.

Parameters

Another complication that may arise in a BVP is the presence of an unknown parameter such as in the working example.

$$y''(x) = -\lambda^2 y(x) \tag{2.12}$$

$$y(0) = y(1) = 0, y'(0) = 1. \tag{2.13}$$

The value of λ is indeed an unknown and must be calculated in addition to the solution function $y(x)$. An alternative deconstruction of (2.12) would be to introduce the value of λ into $\underline{y}$ and since as λ is a constant the corresponding value of $\underline{f}(\underline{y})$ is 0.

That is

$$\underline{y} = \begin{pmatrix} y(x) \\ y'(x) \\ \lambda \end{pmatrix} \quad \text{and} \quad \underline{y}' = \underline{f}(x, \underline{y}) = \begin{pmatrix} y_2 \\ -\lambda^2 y_1 \\ 0 \end{pmatrix}.$$

However there is an obvious disadvantage to posing the problem in this form since increasing the dimension of the problem will increase both computational costs and solution time.

An efficient BVP algorithm would allow the problem to be entered as was it was reduced originally, i.e.

$$\underline{y} = \begin{pmatrix} y(x) \\ y'(x) \end{pmatrix} \quad \text{and} \quad \underline{y}' = \underline{f}(x, \underline{y}) = \begin{pmatrix} y_2 \\ -\lambda^2 y_1 \end{pmatrix}.$$

Hence a general problem can be posed in the form

$$\begin{aligned} \underline{y}'(\underline{x}) &= \underline{f}(x, \underline{y}(x), \underline{p}) \\ g(x_L, x_R, \underline{y}(x_L), \underline{y}(x_R), \underline{p}) &= 0 \end{aligned}$$

and in this case the boundary conditions must suffice to determine the values of $\underline{p}$.

MATLAB Vectorisation

It is intended that the software intended to be produced in this project shall be written for the MATLAB computing environment. The "Matrix Laboratory" provides many convenient ways for creating and manipulating arrays of various dimensions. In the MATLAB jargon a vector refers to a one dimensional ($1N$ or $N1$) matrix and in which form it is natural to represent the mesh and solution values of a finite difference solver. Mentioned previously was MATLAB's efficiency in solving Matrix equations of the form (2.10) specifically with regards to LU factorisation.

Further advantage of this matrix-vector design can when computing the Local Jacobians (2.11) numerically with finite differences. In many cases it is easy to vectorise the RHS function $\underline{f}(x, \underline{y})$ such that it accepts arguements of the form $\underline{x} = [x_1, x_2, \dots]$ and $\underline{y} = [\underline{y}_1, \underline{y}_2, \dots]$ and returns the array $[f(x_1, \underline{y}_1), f(x_2, \underline{y}_2), \dots]$. In this manner the partial derivative at a number of values can be computed using far fewer function evaluations. odenumjacC is an exceptionally robust built-in scheme for the approximation of partial derivatives when integrating a system of ODEs of the form (2.1) which takes advantage of this vectorisation of the RHS.

CHAPTER 3

bvp4c Framework

Kierzenka and Shampine [1] developed the software bvp4c to solve a large class of two-point boundary value problems of the form;

$$\underline{y}'(\underline{x}) = \underline{f}(x, \underline{y}(x), \underline{p}) \quad (3.1)$$

$$g(x_L, x_R, \underline{y}(x_L), \underline{y}(x_R), \underline{p}) = 0 \quad (3.2)$$

where f is continuous and Lipschitz function in y and p is a vector of unknown parameters.

Their view was that a user solving a BVP of form (3.1) in MATLAB would be most interested in the graphical representation of a solution, and as such a modest-order solver with an underlying MIRK4-based Simpson Method would be appropriate for graphical accuracy.

The author believes that whilst a fourth-order solver is acceptable, recent developments mean that a sixth-order solver would supply not only greater accuracy but also perform more efficiently. Thus bvp6c is intended to solve these same problems to a higher degree of accuracy, whilst maintaining the original framework and hence the efficiency, robustness and generality of the existing software.

Although MIRK formulae become more complex as order increases - needing more function evaluations, more interior points and more arithmetic operations at each mesh point - it shall be demonstrated that a higher order method will give comparable accuracy on a sufficiently coarser mesh (or alternatively greater accuracy on a comparable mesh), and thus prove more efficient than its lower-order counterpart for majority of examples. Furthermore, Cash and Moore [5] present a sixth-order interpolant fitting the MIRK6 data of Cash and Singhal [2], which shall prove vital in obtaining an accurate interpolation of the solution at little extra cost.

Before developing the new software it is important to understand how the bvp4c framework operates so that the extension to a sixth-order solver maintains those desirable properties of the original.

bvp4c is introduced as a *Residual control based, adaptive mesh solver.* An adaptive mesh solver is an alternative approach to that of a uniform mesh, which would specify a uniform grid of data points x_i over the interval $[x_i, x_{i+1}]$ and solve accordingly. The adaptive solver will adjust the mesh points at each stage in the iterative procedure, distributing them to points where they are most needed. This can lead to obvious advantages in terms of computational and storage costs as well as allowing control over the grid resolution. The concept of a residual is introduced and discussed in chapter (3.3) and is the cornerstone of the bvp4c framework, being responsible for both error control and mesh selection.

3.1 Initial Setup

Clearly the first step in solving a problem is defining it in a way the software can understand. The bvp4c framework uses a number of subfunctions which make it as simple as possible for the user to enter the ODE function, initial data and parameters for a given problem.

By way of the following example we see exactly how a problem is supplied too and solved by bvp4c. Consider;

$$y(x)'' = -\lambda^2 y(x), \tag{3.3}$$

$$y(0) = y(1) = 0,\ y'(0) = 1. \tag{3.4}$$

which on the interval $[0, 1]$ has the solution

$$y(x) = \sin(\lambda \mathrm{x})/\lambda,\ \lambda = \mathrm{n}\pi$$

Recall that λ is an **unknown** parameter and it is an important feature of bvp4c to be able to solve such a problem efficiently.

The first task is to reduce the equation above to a system of first order equations (as described in the chapter 2.1) and define in MATLAB a function to return these. Similarly the user then rewrites the boundary conditions to correspond to this form of the problem.

```
function dydx = ode(x,y,lambda)
  dydx = [ y(2)
           -lambda*lambda*y(1)];

function res = bc(ya,yb,lambda)
  res = [ ya(1)
          yb(1)
          ya(2)-1];
```

The next step is to create an initial guess for the form of the solution using a specific MATLAB subroutine called **bvpinit** . The user passes a vector x and an initial guess on this mesh in the form bvpinit(x, Yinit), which is then converted into a structure useable by bvp4c. Aside from a sensible guess being necessary for a convergent solution the mesh vector passed to bvpinit will also define the boundary points of the problem, i.e. $x_L = x[1]$ and $x_R = x[\text{end}]$.

The initial guess for the solution may take one of two forms. One option is a vector where Yinit(i) is a constant guess for the i-th component y(i,:) of the solution at all the mesh points in x. The other is as a function of a scalar x, for example bvpinit(x,@yfun) where for any x in $[a, b]$, yfun(x) returns a guess for the solution $y(x)$.

The other role of bvpinit is to define and provide an initial guess for any unknown parameters involved in the problem, such as λ in the working example. When more than one parameter is present, they must be entered in the order in which they appear in the ode function.

The next subroutine to look at is **bvpset**, and whereas bvpinit defined the initial properties of the problem, this function specifies which options bvp4c should be use in solving it. The function is called *options = bvpset('name1',value1,...)* and since MATLAB documentation gives an in depth account of each of the options only a brief outline of those notable is given here.

RelTol - Relative tolerance for the residual [positive scalar 1e-3]

We sahll see that the computed solution $S(x)$ is the exact solution of $S'(x) = F(x, S(x)) + res(x)$. On each subinterval of the mesh, component i of the residual must satisfy

$$\text{norm}\left(\frac{\text{res}(i)}{\max(\text{abs}(F(i)), \text{AbsTol}(i)/\text{RelTol})}\right) \leq \text{RelTol}.$$

AbsTol - Absolute tolerance for the residual [positive scalar or vector 1e-6]

Elements of a vector of tolerances apply to corresponding components of the residual vector. AbsTol defaults to 1e-6. See RelTol.

FJacobian \ **BCJacobian** - Analytical partial derivatives of ODEFUN \ BCFUN

As mentioned previously computation of the Jacobian matrix at each mesh point can be a very expensive process. By passing an analytic derivative of the ODE and BC functions the user can greatly reduce computational time. For example when solving $y' = f(x, y)$, setting FJacobian to @FJAC where $\partial f/\partial y = \text{FJAC}(x, y)$ evaluates the Jacobian of f with respect to y.

Stats - Display computational cost statistics [on — off]

Vectorized - Vectorized ODE function [on — off]

As discussed in chapter (2.6) bvp4c is able to accept a vectorised RHS function $f(x, y)$ which can markedly increase the efficiency of calculating local Jacobians over using finite differences with the odenumjac subroutine.

Hence in the working example, the user would define

```
  options = bvpset('stats','on','reltol',1e-4,'abstol',1e-4);
  solinit = bvpinit(linspace(0,1,5),[1 0],3.14);
```

and call the bvp4c routine with

```
  sol = bvp4c(@ode,@bc,solinit,options);
```

The above essentially ends the user input in solving the BVP system and the rest is left to bvp4c. Within the framework there are several notable steps which should be expounded.

3.2 Collocation and Jacobians

colloc_RHS

Referring back to the matrix system (2.10), this function determines the vector on the right-hand side based upon the current guess for the solution.

colloc_JAC

Whereas the above routine determined the RHS of (2.10) this function is responsible for formulating the matrix of partial derivatives $\partial\Phi/\partial Z$. This is done either by with user-defined analytic Jacobians from FJacobian or by computing them numerically using the MATLAB function **odenumjac**.

The next step is to LU factorise this matrix equation (c.f. section 2.5) and hence solve $LU\delta Z = \Phi$ to obtain the Newton Direction δ. A weak-line search - with a restriction of only four probes - is then employed to determine the next approximation to the solution.

The above process is then repeated four times to allow the difference method to obtain sufficient accuracy on the current mesh. No explanation is given for the choice of four iterations and the author assumes it to be a heuristic figure.

Approximation of Jacobians

Even when vectorisation of the RHS (2.6) is used, computation of the local Jacobians is still an expensive process. Like many other finite difference codes the costs of computing these codes numerically in bvp4c is reduced by approximating some of them. Specifically if the Jacobians at the two mesh points J_i and J_{i+1} satisfy

$$\|J_i - J_{i+1}\|_1 \leq 0.25\,(\|J_i\|_1 + \|J_{i+1}\|_1) \tag{3.5}$$

that is the values of the Jacobian are not changing rapidly over the interval,then $J_{i+1/2}$ the Jacobian at the midpoint will be approximated as the average $(J_i + J_{i+1})/2$. The test (3.5) is made practical by the fast built-in MATLAB function for computing norms.

3.3 Residual

A natural question to ask when solving an ODE numerically is how to quantify the accuracy of a computed solution given that true solution is unknown. Suppose that at any given point during the solution process, holds an approximation y_i to the true solution $y(x_i)$ at each of the mesh points x_i. Clearly one way of looking at the error in the approximation is to compare the difference in these values, however this is often unworkable in practice.

Instead consider the derivative values $f(x, y_i)$ and $f(x, y(x_i))$. The approximation y_i will satisfy $y_i' = f(x_i, y_i) + r(x_i)$ where the residual r gives a measure of the error in the solution at the x_i. Now rather than simply considering the pointwise error of the solution at the mesh points, a much better indication as to the accuracy of the solution can be found by computing the residual over each of the intervals $[x_i, x_{i+1}]$.

To this end, consider the natural $\mathcal{C}^1$ cubic spline interpolant $S(x)$. Spline interpolation uses low order polynomials over the intervals between data points (here the x_i) which join smoothly where they connect. Here $S(x)$ is chosen such that it satisfies the boundary conditions, is a cubic polynomial on each interval $[x_i, x_{i+1}]$ and collocates at both each of the mesh and mid-points. It is continuous at the end points of each interval which alongside to collocation implies that $S(x) \in \mathcal{C}^1[x_L, x_R]$. Using the degrees of freedom available on each interval, we specify that at each mesh point $S(x)$ satisfies $S(x_i) = y_i$ and $S'(x_i) = f(x_i, y_i)$.

From this construction of the spline function define the residual function

$$r(x) = S'(x) - f(x, S(x))$$

from which it can be seen that $S(x)$ satisfies the relation

$$S'(x) = f(x, S(x)) + r(x).$$

Hence when the residual is small $S(x)$ is a good solution, since it is the exact solution of a problem close to that which is posed.

Therefore once the new approximation has been computed the size of the residual $S(x)' - f(S(x), x)$ must be calculated on each interval which can be either with an L_2 or L_∞ norm of the residual vector. In [1] Kierzenka and Shampine justify that an L_∞ norm on a coarse mesh may not give a credible estimate of the residual over the interval.

As such the L_2 norm is favoured and five-point lobatto quadrature scheme used to compute the value of the integral over each subinterval. Since [1] also demonstrates that to leading order the residual function is a cubic polynomial in x, this five-point scheme will provide an asymptotically correct estimate. More discussion on Quadrature schemes is included in the section 4.4 during development of the bvp6c residual.

By construction the spline interpolant gives zero contribution at the end points of the interval and Kierzenka and Shampine go on to demonstrate that the contribution from the mid-point is proportional to the degree of which the current approximation satisfies the collocation equations (2.7). This observation is doubly important for the efficiency of bvp4c. Firstly it removes the need to recalculate the derivative of the interpolant at the midpoint - saving an all important function evaluation - and secondly it gives a direct relation between the residual and the accuracy of the solution. The additional abscissa and weights of the five-point lobatto scheme on the interval $[0, 1]$ are

x_i		w_i	
0	0.0	$1/10$	0.1000
$(1-\sqrt{3/7})/2$	0.1727	$49/90$	0.5444
$1/2$	0.5	$32/45$	0.7111
$(1+\sqrt{3/7})/2$	0.8273	$49/90$	0.5444
1	1.0	$1/10$	0.1000

Table 3.1: Abscissa and Weights of 5-point Lobatto Quadrature

Although the value of the residual function $r(x)$ is already known at the end and mid-points, the value of the computed solution at the two new quadrature points $x[i + (1 \pm \sqrt{3/7})/2]$ is not known. To determine these values, bvp4c uses a Hermite representation of the cubic spline to interpolate the solution on each interval. This process is described in more detail within the later chapter of development to the sixth-order solver.

new_profile

If the residual computed above does not meet the specified tolerance RelTol on some interval then bvp4c will introduce addiotnal mesh points within $[x_i, x_{i+1}]$. Either one or two mesh points will be added depending on the amount by which the residual exceeds this value. If the residual is significantly below the required level on a given interval, then the algorithm will attempt to reduce the number of points by replacing three existing intervals with two new ones based upon an L_∞ estimate of the new residual. Clearly removing unnecessary mesh points reduces computational costs.

The algorithm will then repeat each of the above until such a time as the residual of a solution is is satisfactorily small on each of the intervals or until the maximum allowable number of mesh points is breached.

3.4 Output

Once the above routine has arrived at a solution, one which either satisfies the tolerance on the residual or exceeds the mesh points, bvp4c will output a structure containing the results and other information on the solution.

sol.x - will contain an ordered list of the mesh points at which the final solution is given.

sol.y - contains the computed solution at the points given in sol.x

sol.yp - this field stores the slope data for each of the mesh points, that is for each of the x_i it contains the vector $f(x_i, y_i)$ where y_i is the computed solution.

sol.parameters - will contain the value of any unknown parameters entered into the system.

The information output by the working example is as follows;

```
sol =

          x: [1x17 double]
          y: [2x17 double]
         yp: [2x17 double]
     solver: 'bvp4c'
 parameters: 3.14159882323

 x[i]      y[1][i]   y[2][i]  yp[1][i]  yp[2][i]
      0         0    1.0000    1.0000         0
 0.0625    0.0621    0.9808    0.9808   -0.6129
 0.1250    0.1218    0.9239    0.9239   -1.2022
 0.1875    0.1768    0.8315    0.8315   -1.7454
 0.2500    0.2251    0.7071    0.7071   -2.2214
 0.3125    0.2647    0.5556    0.5556   -2.6121
 0.3750    0.2941    0.3827    0.3827   -2.9025
 0.4375    0.3122    0.1951    0.1951   -3.0812
 0.5000    0.3183   -0.0000   -0.0000   -3.1416
 0.5625    0.3122   -0.1951   -0.1951   -3.0812
 0.6250    0.2941   -0.3827   -0.3827   -2.9025
 0.6875    0.2647   -0.5556   -0.5556   -2.6121
 0.7500    0.2251   -0.7071   -0.7071   -2.2214
 0.8125    0.1768   -0.8315   -0.8315   -1.7454
 0.8750    0.1218   -0.9239   -0.9239   -1.2022
 0.9375    0.0621   -0.9808   -0.9808   -0.6129
 1.0000    0.0000   -1.0000   -1.0000   -0.0000
```

3.5 Devaluating

Since bvp4c is an adaptive mesh algorithm, the mesh points on which the solution is to be obtained cannot be predicted in advance. If the user requires the solution for specific

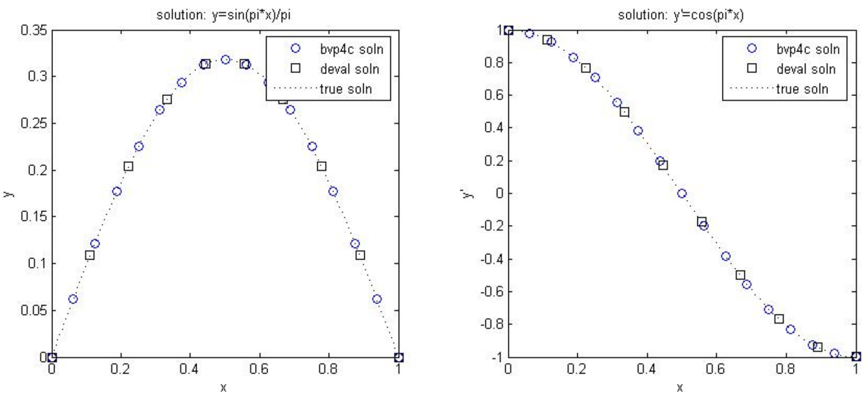


Figure 3.1: Demonstration of bvp4c solution for working example

x values then these can be obtained by using the interpolating subroutine **deval**.

Suppose **Xint** is an ordered vector of those points at which the solution is required and **sol** is the solution structure returned by bvp4c. A call **deval(sol, Xint)**will use the same Hermite polynomial as used within bvp4c to calculate the values of the solution based up the values of y and yp contained within sol. Importantly [8] shows that this polynomial gives a fourth-order approximation to $S(x)$ and hence to $y(x)$, which is a key result also for determining the asymptotic behaviour of the residual.

This concludes the working of bvp4c. The user now has a fourth-order computed solution which they are then free to analyse or display with any of MATLAB's range of mathematical tools. Below the bvp4c solution to 3.3 has been interolated at 10 evenly spaced mesh points - Note how boundary conditions $y(0) = y(1) = 0$ and $y'(0) = 1$ are satisfied.

```
x = linspace(0,1,10);
devaly = deval(sol,x);

figure(1)
plot(sol.x,sol.y(1,:),'ob');hold on;
plot(x,devaly(1,:),'sk');
```

CHAPTER 4

bvp6c Development

4.1 Collocation Equations and Jacobians

The key difference between bvp6c and the original software is that the Cash-Singhal sixth-order scheme accurate scheme will replace the 3-stage Lobatto IIIa formula as the underlying set of difference equations. By positioning formula for these two schemes side-by-side it is apparent that the sixth-order scheme is the more complex and requires the calculation of more interior points on each mesh interval - this is the price paid for the higher-order accuracy. As usual the notation f_i is used to represent $f(x_i, y_i)$.

$$\Phi^{[4]}_{i+1}(X,Y) \quad = \quad y_{i+1} - y_i - \frac{h_i}{6}\left[f_i + 4f_{i+\frac{1}{2}} + f_{i+1}\right] \tag{4.1}$$

where

$$y_{i+\frac{1}{2}} = \frac{1}{2}\left[y_i + y_{i+1}\right] + \frac{h_i}{8}\left\{f_i - f_{i+1}\right\}$$

compared to

$$\Phi^{[6]}_{i+1}(X,Y) \quad = \quad y_{i+1} - y_i - \frac{h}{90}\left[7f_i + 32f_{i+\frac{1}{4}} + 12f_{i+\frac{1}{2}} + 32f_{i+\frac{3}{4}} + 7f_{i+1}\right] \qquad (4.2)$$

where

$$\begin{aligned}
y_{i+\frac{1}{4}} &= \frac{1}{64}\left[54y_i + 10y_{i+1} + h_i\left\{9f_i - 3f_{i+1}\right\}\right],\\
y_{i+\frac{3}{4}} &= \frac{1}{64}\left[10y_i + 54y_{i+1} + h_i\left\{3f_i - 9f_{i+1}\right\}\right],\\
y_{i+\frac{1}{2}} &= \frac{1}{2}\left[y_i + y_{i+1}\right] - \frac{h_i}{24}\left\{5f_i - 16f_{i+\frac{1}{4}} + 16f_{i+\frac{3}{4}} - 5f_{i+1}\right\}
\end{aligned}$$

colloc_RHS

The first step then in updating the bvp4c framework is to use the sixth-order collocation equations to define the RHS of the matrix equation (2.10). Since these equations require them, it is in this routine also the values and derivatives for the interior mesh points are calculated. For a complex objective function these can be computationally expensive and whereas the fourth-order algorithm used only a single interior mesh point, here the two quarter-points mean that an additional two function evaluations on each interval.

colloc_JAC

As described in the section 2.4 on finite difference methods, the Global Jacobian matrices $\partial\Phi_{i+1}/\partial y_i \text{and} \partial\Phi_{i+1}/\partial y_{i+1}$ must be calculated, as well as the parameter Jacobian $\partial\Phi_{i+1}/\partial y_i|_p$ for any unknown parameters p. Kierzenka and Shampine [1] show that the Jacobian for the fourth-order method finite difference method takes the form

$$\frac{\partial\Phi^{[4]}_{i+1}}{\partial y_i} = -\left[I + \frac{h_i}{6}\frac{\partial f_i}{\partial y} + \frac{2h_i}{3}\frac{\partial f_{i+1/2}}{\partial y}\left(\frac{1}{2}I + \frac{h_i}{8}\frac{\partial f_i}{\partial y}\right)\right]$$

$$\frac{\partial\Phi^{[4]}_{i+1}}{\partial y_{i+1}} = I - \frac{h_i}{6}\frac{\partial f_{i+1}}{\partial y} - \frac{2h_i}{3}\frac{\partial f_{i+1/2}}{\partial y}\left(\frac{1}{2}I - \frac{h_i}{8}\frac{\partial f_{i+1}}{\partial y}\right)$$

A similar expression is needed for the sixth-order Global Jacobian and it is shown Appendix (B) that this takes the form

$$\begin{aligned}
\frac{\partial \Phi^{[6]}_{i+1}}{\partial y_i} = I + \frac{h}{90} & \left[7\frac{\partial f_i}{\partial y} + 27\frac{\partial f_{i+\frac{1}{4}}}{\partial y} + 6\frac{\partial f_{i+\frac{1}{2}}}{\partial y} + 5\frac{\partial f_{i+\frac{3}{4}}}{\partial y}\right] \quad + \\
\frac{h^2}{360} & \left[\frac{\partial f_{i+\frac{1}{2}}}{\partial y}\left(27\frac{\partial f_{i+\frac{1}{4}}}{\partial y} - 5\frac{\partial f_{i+\frac{3}{4}}}{\partial y}\right) + \left(18\frac{\partial f_{i+\frac{1}{4}}}{\partial y} - 10\frac{\partial f_{i+\frac{1}{2}}}{\partial y} + 6\frac{\partial f_{i+\frac{3}{4}}}{\partial y}\right)\frac{\partial f_i}{\partial y}\right] \\
+\frac{h^3}{240} & \left[\frac{\partial f_{i+\frac{1}{2}}}{\partial y}\left(3\frac{\partial f_{i+\frac{1}{4}}}{\partial y} - \frac{\partial f_{i+\frac{3}{4}}}{\partial y}\right)\frac{\partial f_i}{\partial y}\right]. \\
\frac{\partial \Phi^{[6]}_{i+1}}{\partial y_{i+1}} = -I + \frac{h}{90} & \left[5\frac{\partial f_{i+\frac{1}{4}}}{\partial y} + 6\frac{\partial f_{i+\frac{1}{2}}}{\partial y} + 27\frac{\partial f_{i+\frac{3}{4}}}{\partial y} + 7\frac{\partial f_{i+1}}{\partial y}\right] \quad + \\
\frac{h^2}{360} & \left[\frac{\partial f_{i+\frac{1}{2}}}{\partial y}\left(5\frac{\partial f_{i+\frac{1}{4}}}{\partial y} - 27\frac{\partial f_{i+\frac{3}{4}}}{\partial y}\right) - \left(6\frac{\partial f_{i+\frac{1}{4}}}{\partial y} - 10\frac{\partial f_{i+\frac{1}{2}}}{\partial y} + 18\frac{\partial f_{i+\frac{3}{4}}}{\partial y}\right)\frac{\partial f_{i+1}}{\partial y}\right] \\
-\frac{h^3}{240} & \left[\frac{\partial f_{i+\frac{1}{2}}}{\partial y}\left(\frac{\partial f_{i+\frac{1}{4}}}{\partial y} - 3\frac{\partial f_{i+\frac{3}{4}}}{\partial y}\right)\frac{\partial f_{i+1}}{\partial y}\right]. \\
\left[\frac{\partial \Phi^{[6]}_{i+1}}{\partial y_i}\right]_{p_1,\ldots p_N} = \tfrac{h}{90} & \left[7\left(\frac{\partial f_i}{\partial \lambda} + \frac{\partial f_{i+1}}{\partial \lambda}\right) + 32\left(\frac{\partial f_{i+\frac{1}{4}}}{\partial \lambda} + \frac{\partial f_{i+\frac{3}{4}}}{\partial \lambda}\right) + 12\frac{\partial f_{i+\frac{1}{2}}}{\partial \lambda}\right] \quad + \\
\tfrac{h_2}{180} & \left[\frac{\partial f_{i+\frac{1}{4}}}{\partial \lambda}\left(9\frac{\partial f_i}{\partial \lambda} - 3\frac{\partial f_{i+1}}{\partial \lambda}\right) + \frac{\partial f_{i+\frac{3}{4}}}{\partial \lambda}\left(3\frac{\partial f_i}{\partial \lambda} - 9\frac{\partial f_{i+1}}{\partial \lambda}\right)\right. \\
& \left. + \frac{\partial f_{i+\frac{1}{2}}}{\partial \lambda}\left(16\left(\frac{\partial f_{i+\frac{1}{4}}}{\partial \lambda} - \frac{\partial f_{i+\frac{3}{4}}}{\partial \lambda}\right) + 5\left(\frac{\partial f_{i+1}}{\partial \lambda} - \frac{\partial f_i}{\partial \lambda}\right)\right)\right] \quad + \\
\tfrac{h_3}{240} & \left[\frac{\partial f_{i+\frac{1}{2}}}{\partial \lambda}\frac{\partial f_{i+\frac{1}{4}}}{\partial \lambda}\left(3\frac{\partial f_i}{\partial \lambda} - \frac{\partial f_{i+1}}{\partial \lambda}\right) + \frac{\partial f_{i+\frac{1}{2}}}{\partial \lambda}\frac{\partial f_{i+\frac{3}{4}}}{\partial \lambda}\left(3\frac{\partial f_{i+1}}{\partial \lambda} - \frac{\partial f_i}{\partial \lambda}\right)\right]
\end{aligned}$$

Clearly the form taken by the Jacobians for this sixth-order method is *considerably* more complex than those from it's fourth-order counterpart. Not only are there more partial derivatives to be calculated (again using odenumjac) but the assembly of these derivatives requires a longer sequence of matrix computations. This can prove costly when dealing with problems of a high dimension hence it is vital to ensure when computing the Global Jacobians that every effort is made to optimise these multiplications.

Following the reasoning of bvp4c, these partial derivatives can often be approximated when there values are not changing rapidly. Since the sixth-order Global Jacobian is more dependant on the partial derivatives at the interior points, a more stringent test must be satisfied to allow this approximation. Here the derivatives at the mesh points must satisfy

$$\|J_i - J_{i+1}\|_1 \leq 0.125\,(\|J_i\|_1 + \|J_{i+1}\|_1) \tag{4.3}$$

in which case those within the interior are averaged as

$$\begin{aligned} J_{i+1/2} &= (J_i + J_{i+1})/2 \\ J_{i+1/4} &= (3J_i + J_{i+1})/4 \\ J_{i+3/4} &= (J_i + 3J_{i+1})/4. \end{aligned}$$

In this instance, averaging of the Local Jacobians is an even more judicious process since it removes the need for computing all three of the interior derivatives.

4.2 Error Approximation and Residual Control

When examining the workings of bvp4c it was shown that the discrete solutions computed by the algorithm were used to produce a piecewise cubic polynomial $S(x)$. This function satisfies the ODE system $S'(x) = f(x, S(x)) + r(x)$ and hence when the residual function $r(x)$ is small $S(x)$ satisfies exactly a system very close to (3.1).

The intention from the start has been that bvp6c will follow this same framework, although to a more accurate degree, so to this end consider a sixth-order $\mathcal{C}^1$ natural spline function $S(x)$ satisfying

$$S(x_i) = y_i, S'(x_i) = f_i, S(x_{i+1}) = y_{i+1}, S'(x_{i+1}) = f_{i+1}$$

$$S'(x_{i+1/2}) = f_{i+1/2}, S'(x_{i+1/4}) + S'(x_{i+3/4}) = f_{i+1/4} + f_{i+3/4}$$

This function will then be twice smooth over $[x_L, x_R]$, satisfy the boundary conditions and be sixth order accurate over each interval - i.e. if we assume that the values y_i and y_{i+1} are sixth-order accurate (which Cash and Singhal [2] tell us is true), then for any $x \in [x_i, x_{i+1}]$

$$S(x) - y(x) = \mathcal{O}(h_i^6).$$

and

$$S^{(j)}(x) - y^{(j)}(x) = \mathcal{O}(h^{6-j}), \quad j = 0, \dots, 5 \,\text{in}\, [x_L, x_R].$$

Furthermore, Cash and Singhal [2] give that the truncation error of the scheme is sixth-order accurate, i.e. $y_i - y(x_i) = \mathcal{O}(h^6)$, and since the function f is Lipschitz it follows both that on each interval

$$\begin{aligned} y_i' - y'(x_i) &\equiv f(x_i, y_i) - f(x_i, y(x_i)) = \mathcal{O}(h_i^6). \\ f(x, S(x)) - f(x, y(x)) &= \mathcal{O}(h^6) \end{aligned}$$

Combining the above

$$\begin{aligned} r(x) &= S'(x) - f(x, S(x))) \\ &= [S'(x) - f(x, S(x))] - [y'(x) - f(x, y(x))] \\ &= [S'(x) - y'(x)] + [f(x, y(x)) - f(x, S(x))] \\ &= S'(x) - y'(x) + \mathcal{O}(h^6). \end{aligned}$$

Finally, since the term $S'(x) - y'(x)$ is $\mathcal{O}(h^5)$, to leading order the residual is equal to the error in the first derivative.

4.3 MIRK6 Interpolant

Cash and Wright [3] present a sixth-order interpolant based upon the MIRK6 data (i.e. the internal mesh points used in the bvp6c scheme), which is defined for any $\omega \in [0, 1]$) by;

$$\begin{aligned} y^{66}(x_i + h_i\omega) &= A_{66}(\omega)y_{i+1} + A_{66}(1-\omega)y_i + h_i \Big[B_{66}(\omega)y_{i+1}' - B_{66}(1-\omega)y_i' + \\ &\quad C_{66}(\omega)\left[y_{i+\frac{3}{4}}' - y_{i+\frac{1}{4}}'\right] + D_{66}(\omega)\overline{y}_{i+\frac{1}{2}}'\Big]. \end{aligned}$$

where

$$\begin{aligned} A_{66}(\omega) &= \omega^2\left(15-50\omega+60\omega^2-24\omega^3\right) \\ B_{66}(\omega) &= \frac{\omega^2}{3}(\omega-1)\left(12\omega^2-14\omega+5\right) \\ C_{66}(\omega) &= -\frac{8}{3}\omega^2(1-\omega)^2 \\ D_{66}(\omega) &= 8\omega^2(\omega-1)^2(2\omega-1) \end{aligned}$$

$$\overline{y}_{i+\frac{1}{2}} = \frac{1}{2}(y_{i+1}+y_i) - \frac{h_i}{24}\{y'_{i+1}-y'_i+4\left[y'_{i+\frac{3}{4}}-y'_{i+\frac{1}{4}}\right]\} \qquad (4.4)$$

When applied to solution data from the Cash-Singhal sixth-order difference scheme with the redefinition of the mid-point value given above, this interpolant is $\mathcal{O}(h_i^6)$ to $S(x)$ across the entire mesh interval.

This interpolant is used to establish the asymptotic behaviour of the residual in the following manner. Let $q(x)$ is be a sixth-order spline interpolant similar to $S(x)$, but corresponding to the true solution $y(x)$. Using the above Cash-Wright interpolant for any $\omega = (x-x_i)/h_i \in [0,1]$ this spline function may be represented locally on each interval by

$$\begin{aligned} q(x_i+\omega h_i) &= A_{66}(\omega)y(x_{i+1}) + A_{66}(1-\omega)y(x_i) + h_i\left[B_{66}(\omega)y'(x_{i+1}) - B_{66}(1-\omega)y'(x_i) + \right. \\ &\quad \left. C_{66}(\omega)\left[y'(x_{i+\frac{3}{4}}) - y'(x_{i+\frac{1}{4}})\right] + D_{66}(\omega)y'(x_{i+\frac{1}{2}})\right]. \end{aligned}$$

This process may be also applied to $S(x)$, hence subtracting the above representation of $q(x)$ and differentiating leads to

$$\begin{aligned} S'(x) - q'(x) &= S'(x_i+\omega h_i) - q'(x_i+\omega h_i) \\ &= \frac{1}{h_i}\left[A'_{66}(w)\left(y_{i+1}-y(x_{i+1})\right) - A'_{66}(1-w)\left(y_i-y(x_i)\right)\right] + \\ &\quad B'_{66}(w)\left(y'_{i+1}-y'(x_{i+1})\right) + B'_{66}(1-w)\left(y'_i-y'(x_i)\right) + \\ &\quad C'_{66}(w)\left(y'_{i+\frac{3}{4}} - y'_{i+\frac{1}{4}} - y'(x_{i+\frac{3}{4}}) + y'(x_{i+\frac{1}{4}})\right) + D'_{66}(w)\left(\overline{y}'_{i+\frac{1}{2}} - y'(x_{i+\frac{1}{2}})\right). \end{aligned}$$

Since B'_{66} and D'_{66} are $\mathcal{O}(1)$ and both the redefined mid-point solution and the mesh point solutions satisfy $y'_i - y'(x_i), \overline{y}'_{i+\frac{1}{2}} - y'(x_{i+\frac{1}{2}}), \overline{y}'_{i+1} - y'(x_{i+1}) = \mathcal{O}(h^6)$ this gives

$$\begin{aligned} S'(x) - q'(x) &= \frac{1}{h_i}\left[A'_{66}(w)\left(y_{i+1} - y(x_{i+1})\right) - A'_{66}(1-w)\left(y_i - y(x_i)\right)\right] \\ & \quad C'_{66}(w)\left(y'_{i+\frac{3}{4}} - y'_{i+\frac{1}{4}} - y'(x_{i+\frac{3}{4}}) + y'(x_{i+\frac{1}{4}})\right) + \mathcal{O}(h^6). \end{aligned}$$

The sixth-order Cash-Singhal formula is satisfied by $y(x)$ with an Local Truncation Error (LTE) $\mathcal{O}(h_i^7)$, i.e. $y_{i+1} - y(x_{i+1}) = y_i - y(x_i) + \mathcal{O}(h^7)$, from which it follows that

$$y_{i+1} - y(x_{i+1}) = y_i - y(x_i) + \mathcal{O}(hi_i^7)$$

Moreover, Cash and Moore [5] show that

$$\begin{aligned} y'_{n+\frac{3}{4}} - y'_{n+\frac{1}{4}} - \left(y'(x_{n+\frac{3}{4}} - y'(x_{n+\frac{1}{4}})\right) &= \frac{3h_i^5}{1024}\left[\frac{1}{5}\frac{\partial f}{\partial y}y^v - \frac{1}{4}\frac{d}{dx}\left(\frac{\partial f}{\partial y}y^{iv}\right)\right] + \mathcal{O}(h_i^6) \\ &\equiv err_{C66}h_i^5 + \mathcal{O}(h_i^6). \end{aligned}$$

So $\quad S'(x) - q'(x) = \frac{1}{h_i}\left[\left(A'_{66}(w) - A'_{66}(1-w)\right)\left(y_i - y(x_i)\right)\right] + C'_{66}(w)err_{C66}h_i^5 + \mathcal{O}(h^6).$

However A' satisfies $A'_{66}(w) - A'_{66}(1-w) = 0$ and therefore

$$S'(x) - q'(x) = C'_{66}(w)err_{C66}h_i^5 + \mathcal{O}(h_i^6).$$

Recall, from the definition of the residual

$$\begin{aligned} r(x) &= S'(x) - f(x, S(x)) \\ &= S'(x) - y'(x) + \mathcal{O}(h_i^6) \\ &= q'(x) - y'(x) + C'_{66}(w)err_{C66}h_i^5 + \mathcal{O}(h_i^6) \end{aligned}$$

Cash and Moore give the interpolation error of $q(x)$ to $y(x)$ as;

$$\frac{h_i^6}{720}w^2(1-w)^2\left(w^2 - w + \frac{9}{32}\right)y^{vi}|_{\xi\in[0,1]}$$

and so differentiating this expression wrt x and substituting to the residual equation gives

$$\begin{aligned} r(x) &= \frac{h_i^5}{3840}w(4w-1)(2w-1)(4w-3)(w-1)y^{vi}(\xi) + h_i^5C'_{66}(w)err_{C66} + \mathcal{O}(h_i^6) \\ &= h_i^5\left[\frac{w}{3840}(4w-1)(2w-1)(4w-3)(w-1)y^{vi}(\xi) - \frac{16w}{3}(2w-1)(w-1)err_{C66}\right] + \mathcal{O}(h_i^6) \end{aligned}$$

So, as expected the behaviour of the residual local to each subinterval and is of order h_i^5 - i.e. to the same order as the error in the first derivative.

In the analysis of the fourth-order interpolant the residual is cubic in ω, third-order accurate and proportional to $y^{(4)}(x_{i+\frac{1}{2}})$. In this instance it is easy to determine an asymptotically accurate maximum of their residual at $\omega = (3 \pm \sqrt{3})/6$ and hence cheaply obtain an estimate for L_∞ on each interval. The more complex form and extra dependance of the new residual computed above howevere mean that the local maxima of the error in each interval is dependant on the problem and hence an L_∞ approximation is not so readily available.

However Kierzenka and Shampine [1] go on to reject the idea of the L_∞ norm in favour of an L_2 norm in bvp4c. That is on each interval they consider

$$\|r(x)\|_i = \left(\int_{x_i}^{x_{i+1}} \|r(x)\|_2^2 dx \right)^{1/2}. \tag{4.5}$$

Obviously such a norm cannot computed be exactly without knowing $r(x)$ explicitly across the $[x_i, x_{i+1}]$ and so numerical quadrature is used to compute this integral to a sufficiently accurate degree.

4.4 Quadrature

In computing the definite integral (4.5), Gaussian quadrature is the natural choice since it is optimal in the sense that an $n-$point Gaussian scheme can fit all polynomials up to a degree $2n - 1$ exactly.

One form of Gaussian quadrature is known as Lobatto quadrature in which two of the n abscissas are fixed at the end points, leaving the other $n - 2$ abscissa free. This is useful since, by construction of the spline interpolant $S(x)$ the residual vanishes at these points and hence gives zero contribution to integral. The restriction of two of the abscissa mean that an n-point Lobatto quadrature scheme is less optimal, solving polynomials of degree $2n-3$ exactly. Thus to cover the tenth-order polynomial $r(x)^2$ a seven-point Lobatto scheme is required. Since the abscissa of Lobatto quadrature are symmetric, it follows that this scheme will contain the mid-point $x_{1+1/2}$ - indeed, the abscissa and weights are as follows;

x_i	0.0	0.08489	0.26558	0.5	0.73442	0.91511	1.0
w_i	0.04762	0.27683	0.43175	0.4876	0.43175	0.27683	0.04762

Table 4.1: Abscissa and Weights of 7-point Lobatto Quadrature

In bvp4c Kierzenka and Shampine manage also to incorporate the contribution from the mid-point by showing $r(x_{i+\frac{1}{2}})$ to be proportional to the collocation function Φ_{i+1}. Since the scheme is designed to take Φ to zero, the value of r at this point then gives an indication of how well the collocation system is being satisfied. Similar analysis was attempted for the quarter-points within sixth-order system, but is prevented by the redefinition of the mid-point in (4.4).

However, evaluating the MIRK6 interpolant at $\omega = 1/2$ shows that

$$\begin{aligned} S(x_{i+1/2}) &= \frac{1}{2}(y_{i+1}+y_i) - \frac{h_i}{24}\{y'_{i+1} - y'_i + 4\left[y'_{i+\frac{3}{4}} - y'_{i+\frac{1}{4}}\right]\} \\ &\equiv \overline{y}_{i+\frac{1}{2}} \end{aligned}$$

Furthermore, taking derivatives of each of the functions from (4.4) at evaluating at $\omega = 1/2$

$$A'(1/2) = B'(1/2) = C'(1/2) = 0, D'(1/2) = 1$$

and hence

$$S'(x_{i+1/2}) = \overline{y}'_{i+1/2} = f(x_{i+1/2}, \overline{y}_{i+1/2}) = f(x_{i+1/2}, S(x_{i+1/2})).$$

The residual at the midpoint is therefore identically zero.

The seven-point lobatto procedure is then used to calcuate the L_2 norm of $r(x)$. Since neither of the end-points nor the mid-point contribute to the integral, only the values of r at the 4 other internal Lobatto abscissas are needed. The MIRK6 interpolation on the current guess is used to find values of $S(x_{lob[i]})$ and $S'(x_{lob[i]})$ which is then subtracted from the evaluation of $f(x, S(x_{lob[i]}))$ to determine $r(x_{lob[i]})$. The product of each of these four points with the associated Lobatto weights is summed and the square-root taken, which gives the L_2 norm of the residual on the interval.

Although as discussed above the quarter-points cannot be used as an indication to the error in the collocation equations, investigation was made as the use of these points (as well as the mid-point) in a modified Gauss-Lobatto quadrature method. Using these three fixed points plus the end points and two further gaussian points it is possible to derive a quadrature scheme capable of solving polynomials of degree 9 exactly. Although to leading order the square of the residual is a polynomial degree 10, it was hoped that the error in such a scheme would be justifiable in removing the need for the two extra RHS function evaluations. However a number of tests showed that the payoff from such an approximation was negligible and the inaccuracy in asymptotic calculation of the residual had damaging affects on the accuracy of the solution. Yet these modified weights and the new abscissa are included here in the hope that future work may over come these difficulties and the iterative routine for their calculation included in appendix (C).

x_i	0.0	0.07182	0.25	0.5	0.75	0.92817	1.0
w_i	0.01825	0.12595	0.22419	0.26320	0.22419	0.12595	0.01825

Table 4.2: Abscissa and Weights of 7-point ‘Lobatto-Gauss-Hale’ Quadrature

4.5 Mesh Selection

As with bvp4c the cornerstone of bvp6c is that both error estimation and mesh selection are based on the residual of the spline function $S(x)$. Once the residual from the approximation on the current mesh has been obtained the algorithm then chooses a new mesh on which it aims to compute a more accurate solution. If on some interval the residual is larger than the user defined tolerance - of which it has been shown that the error in the approximation will also exceed this value - then the algorithm will introduce more points into this interval. Alternatively, if the residual is sufficiently less than this same tolerance over a sequence of intervals then the algorithm will seek to divide these into a smaller set of intervals, reducing the computational costs of the next stage.

Kierzenka and Shampine in [1] state they found it best not to introduce more than two new points into any interval and that has been the experience with bvp6c also. However, since bvp6c uses a higher-order finite difference method it has been observed that for a number of problems it is more efficient to introduce only one new point. Indeed at fourth-order $\|r(s)\|_i$ is $\mathcal{O}(h_i^{7/2})$ whereas at sixth-order it is $\mathcal{O}(h_i^{11/2})$ and so a new point will further reduce the residual. Hence although the default value is kept at two, the user is therefore given an option (a priori) to choose how many points may be added. A similar argument could be used for removing points but it was decided to maintain the existing provisions (although editing the estimate weighting accordingly).

4.6 Output and Devaluation

Since bvp6c uses an adaptive mesh approach the mesh points at which the solution is eventually obtained can bear very little relation to those at which the initial guess was given. With the previous software, if the user wished to find the values of the solution between computed points they would interpolate using the **deval** function.

This option is still available in bvp6c, but whereas bvp4c needed only data from each mesh and mid-point, here the mesh derivatives plus those of the three interior points are needed maintain sixth-order accuracy using the MIRK6 interpolan. This then provides an ultimatum, to store all the derivaives values or to recompute those which are needed when interpolating.

An alternative to both these options is for the user tell the bvp6c software at which points they would like the solution. Since the MIRk6 derivative data is intrinsic to the sixth-order difference scheme, it has already been computed within the software.

As such, if the user is supplying those x values for which the solution is required, or is content with whatever points are chosen by the adaptive mesh then the derivative data is not required. The user is then able to prevent bvp6c from returning this data within the solution structure, which can save considerably on memory allocation. This ability is also useful when it is not important where the solution points lie or when only a small number are required.

Another option for interpolating the solution data from bvp6c is the new **ntrp6c** interpolation helper function. The user calls this function with the function handle for the given ODE problem, an existing solution (which needn't in fact be from bvp6c) for the system and the points at which the solution is required. The function will then determine which mesh intervals from the solution contain the new interpolation points, compute the MIRK6 data points (4.2) and their derivatives, then interpolate at the required values using the MIRK6 interpolant (4.3). Note that ntrp6c can also be used directly with the solution structure returned by bvp6c and also allows problems with parameters, making a very versatile. Examples of usage;

```
xint = linspace(0,1,100);

yint = ntrp6c(ode, xint, sol6)
yint = ntrp6c(ode, xint, sol6.x, sol6.y, [], [], lambda)
yint = ntrp6c(ode, xint, sol4.x, sol4.y, sol4.yp, [], lambda)
```

4.7 User interface

Since the original software is so extensively used within MATLAB, it was important to ensure that the commands used to execute bvp6c are backward compatable with bvp4c and this is indeed the case. Both programs must be called with solinit, which is a structure passed by the bvpinit function. This defines the initial setup of the problem, that is the initial mesh, the initial guess and any known/unknown parameters. In addition the user may also pass the structure 'options' returned by bvpset which may be used to tweak the operation of the software. Options available in the original software were for defining tolerance, analytic jacobians, maximum number of mesh points and the out put of statistics.

bvp6c offers several more options than its predecessor in a manner which can best be described as that of a home DVD recorder: There is a default user interface with a minimum of buttons that overlays a full interface with all of the buttons. The novice user sees only the 'simple' interface. The documentation tells about the basic interface and shows a few model examples whereas advanced documentation describes the full interface.

What new options are available?

1.The software now accepts an initial guess in the form of a matrix of values i.e. a value for each dimension of y at each of the mesh points x_i.

For example

```
x = [0.0000    0.2500    0.5000    0.7500    1.0000]

y = [0.0000    0.2000    0.3000    0.2000    0.0000
     1.0000    0.5000    0.0000   -0.5000   -1.0000]

solinit = bvpinit(x, y, 3.14);
```

2.Maxnewpts - In [1] the authors state that they found it “best not to introduce more than two points at any one time”. Generally this is the case, although in some situations it is best to introduce no more than one. For this reason the default settings for the amount of points introduced has been kept at 2 as in bvp4c, but the user may specify in bvpset a value of their choosing.

```
options = bvpinit('maxnewpts','1.0');
```

3.Xint/SolnPts - Allows the user to pass into bvp6c the set of points for which they require the solution values. See the previous chapter on bvp6c output for more information

```
xint = [0.0000    0.2500    0.5000    0.7500    1.0000]

options = bvpset('Xint',xint,reltol',1e-4,...)
sol = bvp6c(@ode, @bc, solinit, options)
```

4.SlopeOut on/off - suppresses the derivative data if not required. Again see the previous chapter for more details.

```
options = bvpset('SlopeOut','off',reltol',1e-4,...)
sol = bvp6c(@ode, @bc, solinit, options)
```

CHAPTER 5

Testing

Now that the bvp6c software has been implemented into the MATLAB environment its must be tested fully to show it performs favourably against the existing software bvp4c and hence warrant its development. When an analytic solution is available the error of the computed solution can be considered in two ways. Either by taking the points by the solver and computing the true solutions at these x values - referred to as *point errors* - or by evaluating the solution on a fixed mesh and interpolating the results from the numerical solutions at these points - *interpolation errors.* Both are reasonable principals so a decision must be made as to which approach to use. Since the fundamental question is as to the accuracy of the solution computed by the solver rather than the solution obtained by interpolating these points, the predominant consideration here shall be with the point errors.

Even with this, another choice is to how this shall be quantified over the range of the solution. The lower-order method will supply a solution over a larger number of mesh points which will degrade the relevance of an L_2 norm. However, an L_∞ norm may not be representative over the error over the whole mesh. Here an **averaged** L_2 norm is used, that is where the solution is given as a vector $[y_1, y_2, \dots, y_N]$ the error norm is then

$$\left(\frac{1}{N}\sum_{i=1}^{N}[y_i - y(x_i)]^2\right)^{1/2}$$

A further decision was made to consider only the error the first dimension of $\underline{y}$. This is natural if the system (2.1) is the result of an n^{th}-order differential equation, but if the system is the reduction of an existing system of coupled equations this may not be the case. An example of this is the fluid injection example (5.1), where the accuracy of h and θ may be of importance. However, to maintain generality it is assumed that the first dimension of $\underline{y}$ contains the solution of interest.

Consider then the working example (2.12) with an allowable tolerance of $1e-9$. The result of applying both solvers to this problem can be seen below.

```
BVP4c
The solution was obtained on a mesh of 401 points.
The maximum residual is 4.376e-010.
There were 6929 calls to the ODE function.
There were 66 calls to the BC function.
Elapsed time is 1.203000 seconds.

BVP6c
The solution was obtained on a mesh of 31 points.
The maximum residual is 8.200e-010.
There were 1437 calls to the ODE function.
There were 55 calls to the BC function.
Elapsed time is 0.297000 seconds.

Point Error Avg.L2 Norm
      bvp4c:  1.18802e-012
      bvp6c:  6.54455e-012
```

There are a number of things to note from these results. Firstly, the time taken to solve the problem. The sixth-order solver is considerably faster, which is a direct consequence of the far fewer number of mesh points required to obtain the accuracy. Observations can be made too about the accuracy of the solvers. The point errors show that on average the fourth-order solver was more accurate at each point, however on this simple problem the accuracy of both solutions is well below the required tolerance, making this a moot point.

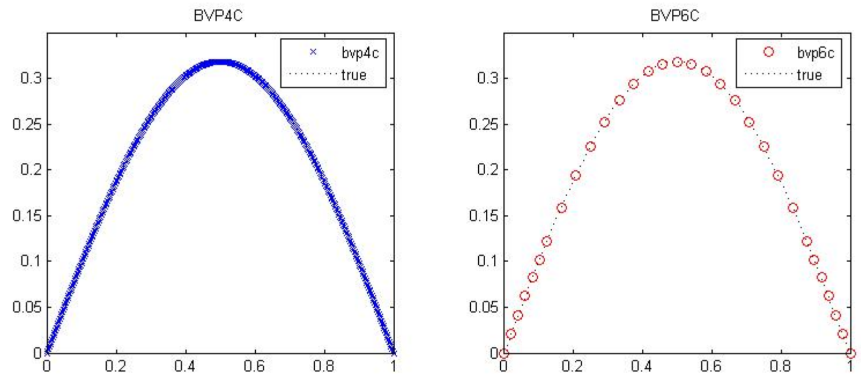


Figure 5.1: Plot of solutions from bvp4c and bvp6c on the working example.

Having demonstrated that the new software behaves appropriately on this trivial problem, it is now time to consider some systems which are a little more taxing.

5.1 Measles, Injections and Shocks

Bvp4c was originally presented with two example ODES to demonstrate how the design and algorithms of the software made it easy to solve a large class of BVPS. Here the same examples to show that the same statements arer equally true for bvp6c.

Measles

The first example is taken from Ascher et al. [6] 1.10 and models the spread of measles via the differential equations

$$\begin{aligned} y_1' &= \mu - \beta(t) y_1 y_3 \\ y_2' &= \beta(t) y_1 y_3 - \frac{y_2}{\lambda} \\ y_3' &= \frac{y_2}{\lambda} - \frac{y_3}{\nu} \end{aligned}$$

where $\beta = 1575(1+\cos(2\pi t))$ and $\mu = 0.02, \lambda = 0.0279$ and $\nu = 0.01$ are given.

The solution should also satisfy the (nonseperated) periodicity conditions $y(0) = y(1)$.

```
function  measles
options = bvpset('Stats','on','reltol',1e-6,'abstol',1e-6);
solinit = bvpinit(linspace(0,1,15),[0.01, 0.01, 0.01]);

sol4 = bvp4c(@odes,@bcs,solinit,options);
sol6 = bvp6c(@odes,@bcs,solinit,options);

function dydx = odes(x,y)
beta = 1575*(1+cos(2*pi*x));
dydx = [    0.02 - beta*y(1)*y(3)
        beta*y(1)*y(3) - y(2)/0.0279
        y(2)/0.0279 - y(3)/0.01     ];

function res = bcs(ya,yb)
res = ya - yb;
```

	bvp4c			bvp6c			
tol	time	mesh	maxres	time	mesh	maxres	error
1e-3	0.1880	20	1.4e-4	0.391	22	9.8e-4	4.07e-4
1e-6	0.5000	130	3.3e-7	0.343	66	3.3e-7	1.25e-4
1e-9	1.3590	385	9.8e-10	0.453	114	1.6e-10	2.29e-8
1e-12	5.4060	2379	1.0e-12	0.953	211	9.4e-13	9.57e-11

Table 5.1: Results from measles example with both bvpc solvers

Both methods perform similarly when only a small degree of accuracy is required but as this is increased bvp6c will solve using much fewer mesh points in a shorter period of time. Quantitatively, for the highest degree of precision, bvp6c uses factor 10 fewer mesh points and solves in one quarter of the time.

Explicit results cannot yet be draw as to the accuracy of the solutions since the analytic solution is not readily available. The 'error' given here is the difference between an interpolation of the results from the two solvers.

Injection

The second example also from Ascher et al. describes flow along a long vertical channel with fluid injection:

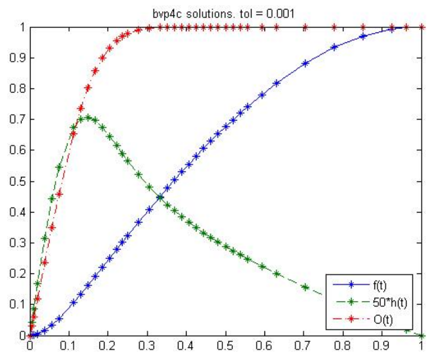
$$\begin{aligned} f''' - R\left[(f')^2 - ff''\right] &= 0 \\ h'' - Rfh' + 1 &= 0 \\ \theta' + Pf\theta &= 0 \end{aligned}$$

where R is a Reynolds number and $P = 0.7R$. The parameter A is unknown and as such there are eight boundary conditions;

$$f(0) = f'(0) = 0,\ f(1) = 1,\ f'(1) = 0,$$

$$h(0) = h(1) = 0,\ \ \theta(0) = 0,\ \ \theta(1) = 1$$

This problem (here taking R=100) is implimented in an almost identical way to that above the following results obtained.



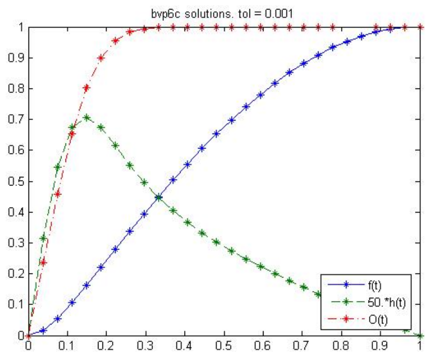


Figure 5.2: Plot of solutions from bvp4c and bvp6c on the Injection problem.

The above graphs of f' shows that the new solution appears the same as that from the existing software and gives an indication of how the mesh points for both methods cluster around points where the solution changes rapidly.

The results from this table agree with that of the previous example. bvp6c performs

	bvp4c			bvp6c			
tol	time	mesh	maxres	time	mesh	maxres	error
1e-3	0.2500	25	5.8e-4	0.234	24	2.6e-5	1.26e-3
1e-6	0.7820	152	9.9e-7	0.313	36	8.9e-7	1.16e-6
1e-9	5.1720	910	9.9e-10	0.813	101	8.3e-10	7.52e-10
1e-12	65.9680	6878	1.0e-12	2.391	346	9.9e-13	8.19e-13

Table 5.2: Results from injection example with both bvpc solvers

comparably with bvp4c for smaller tolerance, but more accurate solutions are computed considerably faster and with much fewer mesh points. The 'error' here again represents the difference between an interpolation of the solutions from the two methods. This difference is to the same order as the specified tolerance, which allows us to be confident that our new method is indeed obtaining the correct solution.

Shockbvp

The final example from the original paper outlining bvp4c also from Asher et al. describes an increasingly severe shock problem through the equation

$$\epsilon y'' + xy' = -\epsilon\pi^2 cos(\pi x) - (\pi x) sin(\pi x)$$

and boundary conditions $y(-1) = -2,\ y(1) = 0$.

This problem can be used to illustrate continuation, vectorization and the benefit of analytic Jacobians. As mentioned throughout, problems with differential equations are often dependant on certain physical parameters. In some cases such as (2.12) these are unknown and must be computed as part of the solution. In others such as this example, the parameter ϵ is defined in advance but still has an dramatic affect on the form of the solution. Consider a problem such as that above with the form $y' = f(x, y; \epsilon)$ for which the solution is desired at a value $\epsilon = \bar{\epsilon}$. Continuation is a process which assumes this problem can be easily solved for some $\epsilon = \epsilon_0$ and that the solution $y(x; \epsilon_{k+1})$ can also be found easily using $y(x; \epsilon_k)$. The problem is then solved for a set of parameters $\{\epsilon_k\}$ such that $\epsilon_k \to \bar{\epsilon}$.

Here the value ϵ describes the width of a boundary layer and the solution becomes progressively difficult to find as this decreases. The problem is solved with with a value $\epsilon = 1e\text{-}2$ from which the solution is used to compute that for $y(x; 1e\text{-}3)$, $(x; 1e\text{-}4)$, $y(x; 1e\text{-}5)$, etc.

	tol=1e-3				tol=1e-6				tol=1e-9
Finalϵ	1e-03	1e-04	1e-05		1e-03	1e-04	1e-05		1e-05
bvp4c	0.32	0.78	2.68		1.68	3.44	6.11		34.45
bvp6c	0.40	1.11	16.53		0.72	2.68	32.89		48.39
vbvp4c	0.34	0.77	2.73		1.67	3.35	6.10		33.65
vbvp6c	0.36	0.93	12.64		0.60	2.15	23.48		31.39
ajbvp4c	0.18	0.41	1.39		0.94	1.85	3.34		18.81
ajbvp6c	0.27	0.51	1.15		0.50	1.29	4.96		16.95
vjbvp4c	0.11	0.24	0.77		0.47	0.91	1.67		9.45
vjbvp6c	0.16	0.28	0.57		0.25	0.60	2.1		6.96

Table 5.3: Time (secs) to solve shockbvp using continuation

The above table, although quite monstrous, contains some results worthy of comment. Following in the steps of the bvp4c analysis in [1] the shockbvp problem is solved using continuation and the times shown are not the intermediate results, but the total time taken to reach a solution for the given value of ϵ. vbvp represents the results from a vectorised version of the problem, ajbvp from using an analytic Jacobian and vjbvp from combining the two. As a general result on both solvers it can be seen that vectorisation has a much smaller affect on the result than that of analytic Jacobians, although it is when used in tandem that the time is most significantly decreased. This is to be expected as the volatility of the solution around the shock layer would oblige the bvp framework to recalculate all of the Jacobians between each mesh point - which is an expensive calculated numerically. Since bvp6c uses the higher-order finite difference method, one would also expect it to perform less efficiently on a shock problem as is clearly the case here. However it has been shown that if analytic Jacobians are available that bvp6c will perform favourably.

5.2 Cash-Wright BVP test suite

bvp6c was tested extensively using the Cash-Wright BVP test suite [4]. This suite is a collection of linear and nonlinear BVP problems designed to test the performance and robustness of a numerical solver.

Since the majority of the problems posed do not have explicit analytic solutions, numerical solutions were computed in MAPLE using the recently developed Capper-Moore twelfth-order MIRK method [15],[16] to a high degree of accuracy on a fine mesh and assumed sufficiently accurate for error analysis.

The Capper-Moore method uses a fixed grid and will give solutions at different mesh points to those of bvp6c with its predecessor bvp4c which are both adaptive solvers. To test the accuracy of these non-fixed points, there are then two possible ways in which to proceed. The first is to use the interpolant associated with each of the two solvers to compute solution values at those fixed points given by the twelfth-order method. This idea is rejected as the higher order of the bvp6c interpolant would give it an advantage during an interpolation - although it can be argued that in practice the user would be interpolating the computed solution and hence be interested in the accuracy of the interpolated. However here interest is mostly with testing the point accuracy of the solver, not that of its interpolant.

Hence instead, the solution given by the high-order method is interpolated at the mesh points given by the two bvp solvers and an average norm of the errors at each mesh point computed. Since the problems from [4] are often used a benchmark for testing ODE solvers, my supervisor Dr. D. Moore requested I write a 'black box' for these problems in MATLAB. The idea being that a user may call blackbox32(P,X), where P is the number of the Cash-Wright problem and X is a vector of the points at which we wish to interpolate the solution. The function would then return a twefth-order accurate interpolation of the Capper-Moore solutions at those requested points.

blackbox32.m

For those of the 32 problems for which an explicit solution is not available, solutions were computed in Maple to a high degree of precision using the Capper-Moore twelfth-order method previously mentioned and the results stored in a CSV data file. The black box program will be called with the problem number we are looking at and those points at which we wish to interpolate. The black box function proceeds as follows.

- If specified problem is linear with known solution then evaluate solution at each given point, output solution and return
- Else, read the data from the stored solution (CWBVP.dat)
- Check to see that the values in X are within the solution range
- Determine which of the intervals $[x_i, x_{i+1}]e$ each element of X is in
- On each interval which contains a point which is to be interpolated, compute the MIRK12 interpolant (note this will require function evaluations and the blackbox function has access to a secondry file containing the ODE functions $\underline{f}(x, \underline{y})$) of each of the problems.

- Interpolate for each point in X and return the computed values

Demonstration
```
function test_blackbox(P,x)
%determine solution at given x using blackbox
y=blackbox32(P,x);

%extract MIRK12 solution for comparison
p=0;
[XL,XR]=xlist(P);
info=csvread('h:\CWBVP_data.txt',0,0,[0,0,0,2]);
while info(1)~=P
    p=p+info(2)+1;
    info=csvread('h:\CWBVP_data.txt',p,0,[p,0,p,2]);
end
yy=csvread('h:\CWBVP_data.txt',p+1,0,[p+1,0,p+info(2),info(3)-1]); %mirk 12 solns
xx=XL:(XR-XL)/(info(3)-1):XR;

%plot blackbox interpolation over MIRK12 solution
figure(1)
plot(x,y(1,:),'*r');hold on
plot(xx,yy(1,:),'-b');hold off
legend('blackbox interpolations', 'MIRK12 solution');
title(['interpolated solutions to C-W BVP suit #',num2str(P)]);
```

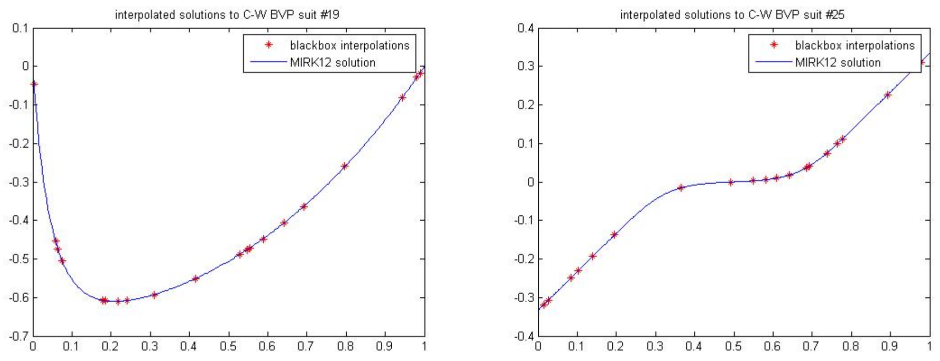


Figure 5.3: Illustrations of the blackbox32 routine on CW19 and CW25

Using this system a highly accurate solution is obtained which is treated as exact for the purpose of error analysis of bvp4c and bvp6c. The next stage is to compute solutions to each of the problems using these two solvers with a range of specified tolerances to compare amongst other properties their accuracy and efficiency. In the testing of these 32 problems a tolerance - for simplicity the same value for relative and absolute - is chosen with an initial starting grid of 33 equally spaced points and an initial guess of a zero vector.

```
function testproblems_solve(problem,tol)
```

```
% % % -----------------SETUP--------------------
E=epslist(problem);
[xl,xr] = xlist(problem);
[ode,bc,iguess] = feval(@problemlist,problem);

options = bvpset('reltol',tol,'abstol',tol);

x=linspace(xl,xr,Nstart);
solinit = bvpinit(x,zeros(dim,33));

% % % ------------bvp4c.m-----------------------
fprintf('\n  BVP4c \n');tic
sol4 = bvp4c(ode,bc,solinit,options,E);
time4= toc;

soln4 = blackbox32(problem, sol4.x);
Y4_L2 = norm(sol4.y(1,:)-soln4(1,:))/sqrt(length(sol4.x));

% % %------------bvp6c.m-----------------------
fprintf('\n  BVP6c \n');tic
sol6 = bvp6c(ode,bc,solinit,options,E);
time6= toc;

soln6 = blackbox32(problem, sol6.x);
Y6_L2 = norm(sol6.y(1,:)-soln6(1,:))/sqrt(length(sol6.x));
```

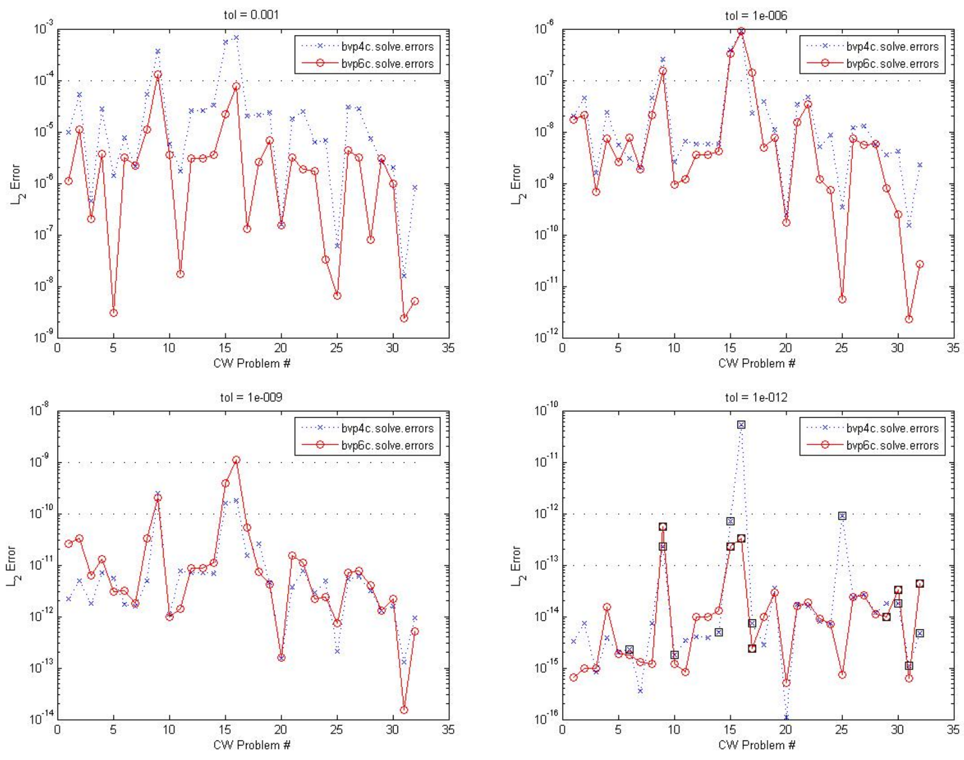


Figure 5.4: L_2 errors from bvpc solvers on CW32 problems with varying tolerance.

Above can be seen the results of applying both methods to the 32 given BVPs as the tolerance levels are decreased simultaneously. For the vast majority of the problems, bvp6c obtains a solution to the specified accuracy in the averaged L_2 norm (also the L_∞ sense) and in those cases which it doesn't, it still outperforms the bvp4c algorithm.

In the final graph a black square indicates the method failed to achieve what it believed was a correct solution (i.e. sufficient reduction of the residual) within the allowable number of mesh points. In fact, bvp4c with a default maximum of $10000/n$ (where n is the size of the ODE system) failed in all but one case and the errors from the true solution remained significantly large (w.r.t. tolerance). Subsequently the default maximum mesh size was increased to $20000/n$ of which the results are given above and nonetheless in some cases bvp4c failed to obtain accuracy.

It can also be seen that there are some problems for which the solvers supply a sufficient accurate result, but for which the residual was not sufficiently reduced. This then implies that in some cases the residual of the bvp framework overestimates the error bound, although this is naturally preferential to an underestimate.

Having seen then that the sixth-order solver compares favourably in terms of accuracy to that of the existing fourth-order, it must still be ascertained that this does not occur at the expense of an improportionate increase in solution time.

It is clear that for the lower tolerance ($1e-3$) the times for each method are comparable, for the 'midrange' tolerances ($1e-6$, $1e-9$) the new software is consistently around twice as fast and for the highest tested tolerance ($1e-12$) bvp6c is up to ten-times faster.

Again problems 16&25 are notable since for these problems bvp4c is in fact the faster

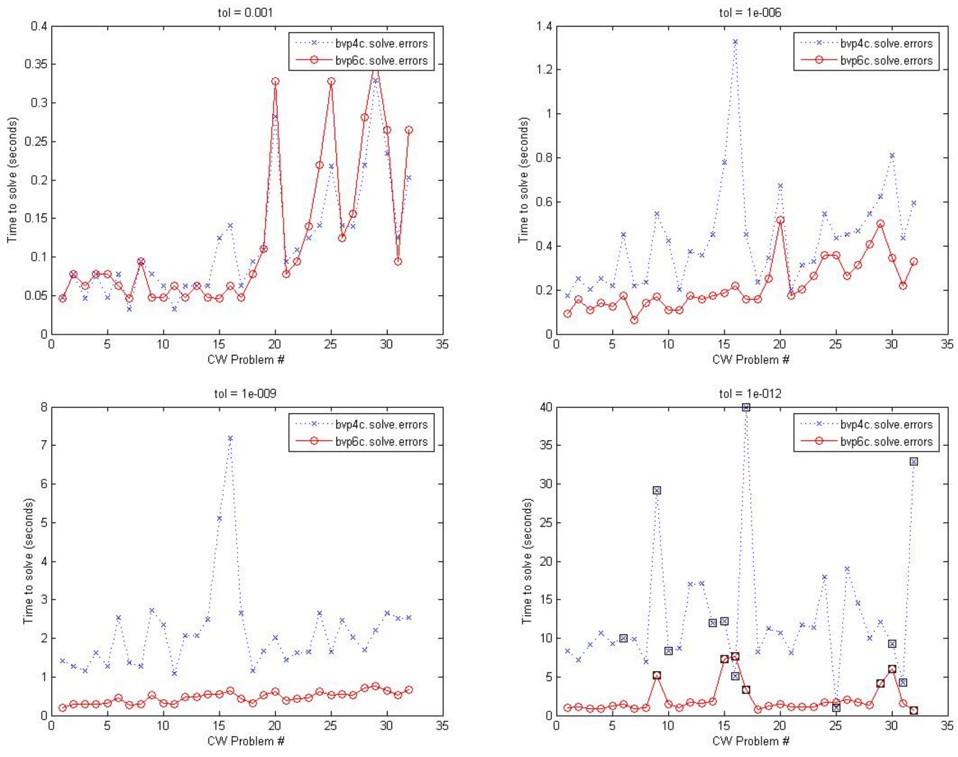


Figure 5.5: solution times bvpc solvers on CW32 problems with varying tolerance.

solver. However this is easily justifiable when noted that the bvp4c method failed to obtain obtained a sufficiently accurate solution before termination, that is bvp4c only appears faster here since it in effect 'gave up' quicker. Further tests have shown that if allowed to run for an equal length of time it will remain less accurate than bvp6. Hence although these 32 problems have given rise to a small number of complications which are still under consideration, the author believes that these results show that bvp6c is indeed the more efficient solver.

CHAPTER 6

Conclusion and Further Work

6.1 Conclusion

It is hoped in view of these examples that the reader has been left convinced that the new software bvp6c is in the least comparable too and in the majority of cases superior to the existing item bvp4c. The aim of this project was introduced using the words *accuracy, efficiency, robustness* & *generality* . Hindsight allows a more in depth perspective of how these terms apply in context and to the extent to which they have been fulfilled.

The **accuracy** between the two solvers is a difficult item to quantify. The design of the bvp framework means that the solver will continue to improve upon its approximations until it arrives at a solution it deems sufficiently accurate. This accuracy is a product of the residual function $r(x)$ and it was shown in (4.2) that locally the bvp6c residual was asymptotically sixth-order to the error from true solution whereas in bvp4c it was fourth-order. However, this type of analysis is only really significant when considering solutions obtained on the same mesh. Since the bvpc framework is based upon an adaptive mesh algorithm which is in turn selected by the residual it is natural that each method will select its own mesh and to obtain comparable reductions in residual the bvp4c solver will require a more dense mesh.

With this in mind, accuracy is then best judged by result. In the extensive testing (particularly of chapter (5.2)) it was shown that whilst not always providing the more accurate of the solutions, bvp6c consistently obtained the accuracy specified by the user. The conclusion for this is that provided both methods give a solution satisfying the required tolerance, a comparison of accuracy between between them is of less importance than that of efficiency.

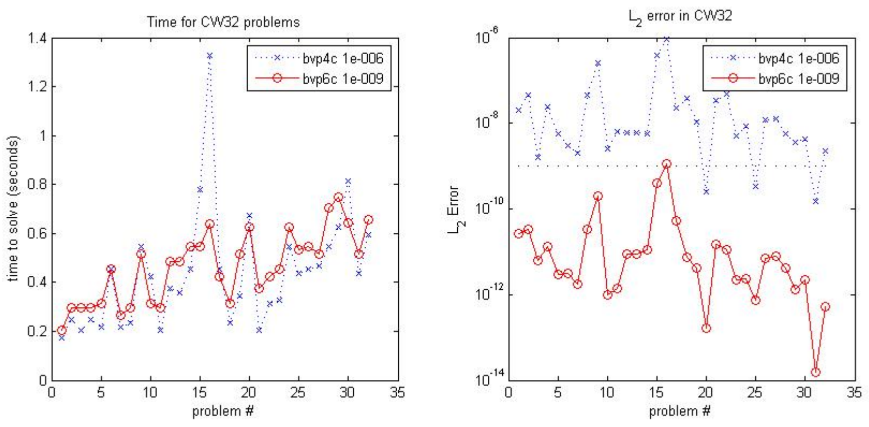


Figure 6.1: solution times and errors for bvpc solvers on CW32 problems

Since the accuracy then has been dismissed as subsidiary to **efficiency** it is important then that bvp6c show a notable increase in this area to justify its development. Indeed, again from those results (5.2) of the Cash-Wright BVP test suite [4] it was shown that bvp6c would give a comparably accurate result in a significantly shorter amount of time.

Furthermore, a comparison of solution times for the bvp4c solver at 1e-3 accuracy to those from bvp6c at 1e-6 accuracy figure(6.1) shows an almost identical time is taken by each.

This demonstrates that in the same time it takes bvp4c to solve each of the solutions to an accuracy of 1e-6, bvp6c can solve the same problem to an accuracy of $1e-9$. A similar result hold between accuracies of 1e-3 and 1e-6 and to an even greater degree for 1e-9 and 1e-12. From these results and others from testing in chapter (5) it can be concluded that indeed the new software is decidedly the most accurate and efficient solver.

As far as the **generality** and **robustness** are concerned the new algorithm was so compatible with the existing framework that any problem solvable through the original has the potential to be solved by the new software. The use of unknown paramaters and non-separated boundary conditions in addition to that of of singular and multi-point boundary problems all continue to be accepted in the sixth-order extension. With an eye to robustness, no problems solvable by bvp4c have yet been found which cause too great a difficulty for the new software. As discussed in (5.1) the shockbvp problem of Ascher et al. [6] gives bvp6c some discomfort for small values of ϵ, but one should expect this of a high-order solver when faced with a severe shock. In situations like these a modest-order solver such as bvp4c should indeed take preference.

Final Conclusion

In light of the discussion above it can be concluded that the bvp6c software achieves those goals outlined at its inception. The main advantages it takes over the existing package are in a significantly reduced runtime and the ability to obtain accurate solutions on a coarse mesh even when a very restrictive tolerance is defined. For these reasons the updated software shall be submitted for review by The Mathworks, inc - producers of the

MATLAB environment for inclusion in future versions of the MATLAB Package. At the time of printing the software was already available on the Mathworks online file exchange and although having been downloaded by over 30 users, was yet to be reviewed.

6.2 Further Work

Singular BVPs and multi-boundary condition support

bvp4c claims to solves a class of singular BVPs (which may also include unknown parameters p) of the form

$$\begin{aligned} y' &= Sy/x + f(x, y, p) \\ g(y(x_L), y(x_R), p) &= 0 \end{aligned}$$

Such problems arise when computing a smooth solution of ODEs that result from PDEs because of cylindrical or spherical symmetry. Currently little testing has been undertaken to verify that the bvp6c extension will maintain this ability, however since the form of the finite difference method has impact on the routine for dealing with the singularity the author feels that this will be the case.

Similarly, bvp4c has the ability to solve multi-point boundary value problems, for which additional conditions apply within the interval $[x_l, x_R]$. Generally these points represent interfaces for which it is natural to divide the interval into a set of smaller regions which may be solved separately. Again, every effort was made during implementation of the new software to maintain this ability and the author sees no reason why this should not be so. However before being released, more extensive testing on this and indeed the previous item should be undertaken.

More Accurate Interpolants

In (4.3) it was shown that the error in the term $[y'_{i+3/4} - y'_{i+1/4}]$ dominate the interpolation of the spline function $S(x)$ and hence affect the asymptotic behavior of the residual $r(x)$. When introducing this interpolant in [5] Cash and Moore describe how the redefinition of a more accurate value for quarter-points can produce a threefold reduction in the error of the interpolation. Such a reduction would increase the accuracy of the residual and hence give a stronger indication as to the accuracy of a computed solution. Although the recalculation of the derivatives at these new points will require additional function evaluations on each interval, it seems worthwhile to investigate the effect of this more accurate interpolant on the bvpc framework.

bvp8c, bvp10c, bvp12c Extensions

In view of the apparent success of the bvp6c extension to the existing MATLAB software, it is tempting to consider a further development to methods using eighth [2], tenth and even twelfth-order [15] difference methods. However, before undertaking this venture one should consider the implications that such high-order schemes would pose.

The Cash-Singhal [2] scheme used in bvp6c required the introduction of two additional interior points in the difference calculations - the corresponding eighth-order scheme requires a total of seven interior points per interval and so although the greater accuracy of the scheme will require fewer mesh points, this would potentially be countered by the expense of more function evaluations.

It was shown also in this work that the Global and Local Jacobian matrices for the sixth-order method takes a much more complex form than that of the fourth-order. As the order is increased these Jacobians becoming significantly more convoluted, hence difficult and expensive to compute. This again means that the increase in accuracy may be unjustifiable due to a decrease in accuracy. It seems as though the law of diminishing returns with respect to increased accuracy at the expense of additional calculation would begin to take affect if an eighth or higher-order finite difference scheme we used.

Finally, although Cash and Moore describe in [5] an eighth-order MIRK8 based interpolant, establishing a workable asymptotic estimate to the behaviour of the residual based upon this interpolation of a sufficiently accurate seems unlikely. Since the residual is the corner-stone of the bvpc framework, being responsible for error estimation and mesh selection, without proper knowledge of proper knowledge of its behavior the accuracy of any obtained solution cannot be guaranteed.

Special Systems: $y''(x) = f(x, y)$ **and** $y'' = f(x, y, y')$**, etc.**

It was mentioned briefly in the introduction to this project that there exist a particular class of ODES which may be expressed in such forms as those given above. Having expressed doubt as to the success of further increasing the order of the solver, the author does however see potential for the enhancement of the bvpc framework to efficiently solve such problems. Indeed, again in [5] Cash and Moore present a sixth-order Hermite-Birkhoff interpolant to fit data produced by a finite difference scheme corresponding to these forms of problems. It remains then only to determine a suitable representation of the residual computable on each interval and justify its correspondence to the accuracy of a solution.

APPENDIX A

bvp6c Research Paper

A Sixth-Order Extension to the MATLAB bvp4c Software of J. Kierzenka and L. Shampine.

N. P. HALE
and
D. R. MOORE
Department of Mathematics,
Imperial College London

Abstract – Kierzenka and Shampine [1] present both theory and implementation of a residual control based BVP solver, commonly known as bvp4c. Presented here is a direct extension to this work, namely development and justification of a sixth-order accurate solver which we have named bvp6c.

Keywords –Collocation, MATLAB, Residual control, BVP4C

A.1 Introduction

Kierzenka and Shampine [1] developed bvp4c to solve a large class of boundary value problems (BVPS) for ordinary differential equations (ODES) in the MATLAB problem solving environment (PSE). Specifically they were concerned with first order systems of ODEs of the form;

$$y' = f(x, y, p), \quad a \le x \le b \tag{A.1}$$

subject to two-point boundary conditions (BCS)

$$g(y(a), y(b), p) = 0 \tag{A.2}$$

where p is a vector of unknown parameters. Their view was that a user solving a BVP of form (A.1.A.2) in MATLAB would be most interested in the graphical representation of a solution and, as such, a modest-order solver such as their MIRK4 based Simpson Method would be appropriate for graphical accuracy.

It is our opinion that whilst a fourth-order solver is acceptable, recent developments mean that a sixth-order solver would would supply not only greater accuracy, but will also perform more efficiently. Thus bvp6c is intended to solve these same problems to a higher degree of accuracy, whilst maintaining the efficiency and robustness of the original software and indeed other valuable properties such as non-separated BCS and unknown parameters.

Although MIRK formulae become more complex as order increases - needing more function evaluations, more interior points and more arithmetic operations at each mesh point - we believe a higher order method will give comparable accuracy on a sufficiently coarser mesh (or alternatively greater accuracy on a comparable mesh) and thus prove more efficient than its lower-order counterpart for majority of examples. Furthermore, Cash and Moore [5] present a sixth-order interpolant fitting the MIRK6 data of Cash and Singhal [2] which we may be used to obtain an accurate graphical interpolation of the solution at little extra cost.

A.2 COLLOCATION METHOD

Although we maintain the assumption that most solutions solved in a PSE rather than in a GSE will be viewed graphically, we must still be able to obtain accurate answers anywhere in the region [a,b]. Some popular methods provide solutions only at mesh points, whereas others provide data everywhere although without uniform accuracy. Here we begin to describe our method which satisfies all these requirements. We assume throughout that the functions $f(x, y, p))$ and $g(x, y, p)$ of (A.1.A.2) are as smooth as necessary. In particular, f is continuous and satisfies a Lipschitz condition in y. For simplicity we suppress the argument p.

The numerical method we use here is in many ways similar to the Simpson method used in bvp4c. Indeed, our method can be viewed as a collocation with a piecewise quintic (whereas bvp4c used cubic) polynomial function $S(x)$ - which we shall refer to as the natural interpolant. $S(x)$ satisfies the boundary conditions on each subinterval $[x_i, x_{i+1}]$ and collocates at the end of the subinterval, the midpoint and the quarter points. It is continuous at the endpoints of each subinterval, which along with collocation then implies that $S(x) \in \mathcal{C}^1[a, b]$. As with bvp4c, the cornerstone of bvp6c is that both error estimation and mesh selection are based on the residual of $S(x)$.

This formula can be evaluated efficiently using analytical condensation. Specifically, if $y_i = S(x_i) \approx y(x_i)$ and $h_i = x_{i+1} - x_i$, then the formula given by Cash and Singhal is

$$\begin{aligned} y_{i+1} &= y_i + \frac{h_i}{90}\Big[7f(x_i, y_i) + 32f(x_{i+\frac{1}{4}}, y_{i+\frac{1}{4}}) + 12f(x_{i+\frac{1}{2}}, y_{i+\frac{1}{2}}) + ... \\ &\qquad 32\,f(x_{i+\frac{3}{4}}, y_{i+\frac{3}{4}}) + 7f(x_{i+1}, y_{i+1})\Big], \\ &= y_i + \frac{h_i}{90}\Big[7f_i + 32f_{i+\frac{1}{4}} + 12f_{i+\frac{1}{2}} + 32f_{i+\frac{3}{4}} + 7f_{i+1}\Big] \end{aligned}$$

where

$$\begin{aligned} y_{i+\frac{1}{4}} &= \frac{1}{64}\left[54y_i + 10y_{i+1} + h_i\left\{9f_i - 3f_{i+1}\right\}\right], \\ y_{i+\frac{3}{4}} &= \frac{1}{64}\left[10y_i + 54y_{i+1} + h_i\left\{3f_i - 9f_{i+1}\right\}\right], \\ y_{i+\frac{1}{2}} &= \frac{1}{2}\left[y_i + y_{i+1}\right] - h_i\left\{5f_i - 16f_{i+\frac{1}{4}} + 16f_{i+\frac{3}{4}} - 5f_{i+1}\right\} \end{aligned}$$

It is shown [2] that if $h = \max h_i$ and h/h_i is bounded above that this formula is uniformly 6^{th} order accurate at the mesh points and if we enforce the Lipschitz condition on f, it follows that

$$y_i' - y'(x_i) = f(x_i, y_i) - f(x_i, y(x_i)) = O(h^6).$$

and also that the natural interpolant $S(x)$ (such that $S(x_i) = y_i$) satisfies

$$S^{(j)}(x) = y^{(j)}(x) + O(h^{6-j}), \qquad j = 0, ..., 6 \tag{A.3}$$

uniformly for x in $[a, b]$.

A.3 RESIDUAL

Recall $S(x)$ is the natural interpolant of the solution at mesh points and we define the residual of $S(x)$ as

$$\begin{array}{ll} r(x) = S'(x) - f(x, S(x)) & \text{for the ODES and} \\ g(S(a), S(b)) & \text{for the BCS.} \end{array}$$

This Lipschitz condition on f and equation (3) imply that for $a \le x \le b$,

$$f(x, S(x)) = f(x, y(x)) + O(h^6)$$

hence

$$\begin{aligned} r(x) &= (S'(x) - f(x, S(x))) - (y'(x) - f(x, y(x))) \\ &= S'(x) - y'(x) + O(h^6). \end{aligned}$$

We now seek to establish the asymptotic behavior of this residual.

Kierzenka and Shampine [1] use a cubic fourth-order Hermite interpolant to fit $\{y_i, y_i', y_{i+1}, y_{i+1}'\}$, yet Cash and Moore [5] show how that a sixth-order interpolant can be found to fit the data $\{y_i, y_i', y'_{i+\frac{1}{2}}, y'_{i+\frac{3}{4}} - y'_{i+\frac{1}{4}}, y_{i+1}, y'_{i+1}\}$. This information is known to our method, but we must calculate a more accurate estimate of $y_{i+\frac{1}{2}}$ (denoted $\overline{y}_{i+\frac{1}{2}}$ - formula below) and an additional function evaluation to find the new $f_{i+\frac{1}{2}}$, i.e. $f(x, \overline{y}_{i+\frac{1}{2}}) = \overline{f}_{i+\frac{1}{2}}$(or $\overline{y}'_{i+\frac{1}{2}}$).

$$\overline{y}_{i+\frac{1}{2}} = \frac{1}{2}(y_{i+1} + y_i) - \frac{h_i}{24}\{y'_{i+1} - y'_i + 4\left[y'_{i+\frac{3}{4}} - y'_{i+\frac{1}{4}}\right]\}$$

Now, if $q(x)$ is such a polynomial interpolating the true solution $y(x)$ for $x \in [x_i, x_{i+1}]$ and $\omega = (x - x_i)/h_i \in [0, 1]$ it may be written as

$$\begin{aligned} q(x_i + \omega h_i) \quad = \quad & A_{66}(\omega)y(x_{i+1}) + A_{66}(1-\omega)y(x_i) + h_i\left[B_{66}(\omega)y'(x_{i+1}) - B_{66}(1-\omega)y'(x_i) + \right. \\ & \left. C_{66}(\omega)\left[y'(x_{i+\frac{3}{4}}) - y'(x_{i+\frac{1}{4}})\right] + D_{66}(\omega)y'(x_{i+\frac{1}{2}})\right]. \end{aligned}$$

Subtracting this from a similar expression for $S(x)$ and differentiating leads to

$$\begin{aligned} S'(x) - q'(x) &= S'(x_i + \omega h_i) - q'(x_i + \omega h_i) \\ &= \frac{1}{h_i}\left[A'_{66}(w)\left(y_{i+1} - y(x_{i+1})\right) - A'_{66}(1-w)\left(y_i - y(x_i)\right)\right] + \\ & \quad B'_{66}(w)\left(y'_{i+1} - y'(x_{i+1})\right) + B'_{66}(1-w)\left(y'_i - y'(x_i)\right) + \\ & \quad C'_{66}(w)\left(y'_{i+\frac{3}{4}} - y'_{i+\frac{1}{4}} - y'(x_{i+\frac{3}{4}}) + y'(x_{i+\frac{1}{4}})\right) + D'_{66}(w)\left(\overline{y}'_{i+\frac{1}{2}} - y'(x_{i+\frac{1}{2}})\right). \end{aligned}$$

Since B'_{66} and D'_{66} are $O(1)$ and $y'_i - y'(x_i), \overline{y}'_{i+\frac{1}{2}} - y'(x_{i+\frac{1}{2}})$ & $\overline{y}'_{i+1} - y'(x_{i+1}) = O(h^6)$

$$\begin{aligned} S'(x) - q'(x) \quad = \quad & \frac{1}{h_i}\left[A'_{66}(w)\left(y_{i+1} - y(x_{i+1})\right) - A'_{66}(1-w)\left(y_i - y(x_i)\right)\right] \\ & C'_{66}(w)\left(y'_{i+\frac{3}{4}} - y'_{i+\frac{1}{4}} - y'(x_{i+\frac{3}{4}}) + y'(x_{i+\frac{1}{4}})\right) + O(h^6). \end{aligned}$$

We also find that the sixth-order Cash-Singhal formula is satisfied by $y(x)$ with an Local Truncation Error (LTE) $O(h_i^7)$, i.e. $y_{i+1} - y(x_{i+1}) = y_i - y(x_i) + O(h^7)$.
Moreover, Cash and Moore [2] show that

$$\begin{aligned} y'_{n+\frac{3}{4}} - y'_{n+\frac{1}{4}} - \left(y'(x_{n+\frac{3}{4}} - y'(x_{n+\frac{1}{4}})\right) &= \frac{3h_i^5}{1024}\left[\frac{1}{5}\frac{\partial f}{\partial y}y^v - \frac{1}{4}\frac{d}{dx}\left(\frac{\partial f}{\partial y}y^{iv}\right)\right] + O(h_i^6) \\ &= err_{C_{66}}h_i^5 + O(h_i^6). \end{aligned}$$

So $S'(x) - q'(x) = \frac{1}{h_i}\left[\left(A'_{66}(w) - A'_{66}(1-w)\right)(y_i - y(x_i))\right] + C'_{66}(w)err_{C_{66}}h^5 + O(h^6)$.

However, A' satisfies $A'_{66}(w) - A'_{66}(1-w) = 0$ and therefore

$$S'(x) - q'(x) = C'_{66}(w)err_{C_{66}}h^5 + O(h^6).$$

Recall, from the definition of the residual

$$\begin{aligned} r(x) &= S'(x) - f(x, S(x)) \\ &= S'(x) - y'(x) + O(h_6) \\ &= q'(x) - y'(x) + C'_{66}(w)err_{C_{66}}h^5 + O(h_6) \end{aligned}$$

Cash and Moore give the interpolation error of $q(x)$ to $y(x)$ as;

$$\frac{h^6}{720}w^2(1-w)^2\left(w^2 - w + \frac{9}{32}\right)y^{vi}\big|_{\xi\in[0,1]}.$$

Finally we differentiate wrt x and substitute to the residual equation giving

$$\begin{aligned} r(x) &= \frac{h^5}{3840}w(4w-1)(2w-1)(4w-3)(w-1)y^{vi}(\xi) + h^5C'_{66}(w)err_{C_{66}} + O(h^6) \\ &= h^5\left[\frac{1}{3840}w(4w-1)(2w-1)(4w-3)(w-1)y^{vi}(\xi) - \frac{16}{3}w(2w-1)(w-1)err_{C_{66}}\right] + O(h^6). \end{aligned}$$

As expected, this behaviour of the residual local to each subinterval and is of order h^6 at the two end-points and the centre-point. However, our interpolation scheme has left us with a residual of order h^5 dependant on both y^v and y^{iv} at the quarter-points.

Thus to leading order the residual shown above is a polynomial of degree 5, dependant on y^{iv}, y^v and y^{vi}. This extra dependance means the behavior of $r(x)$ is is much more complex and leaves us unable to determine its local extrema, and therefore unable to determine an asymptotically correct estimate of the L_∞ norm.

However, since Kierzenka and Shampine [1] argue against the use of the L_∞ norm, we have little cause for concern and instead concentrate on the L_2 norm. That is, on each subinterval we estimate

$$\|r(x)\|_i = \left(\int_{x_i}^{x_{i+1}} \|r(x)\|_2^2 dx\right)^{1/2}$$

In calculating this L_2 norm we must use a more accurate quadrature method to integrate the now tenth-order polynomial $\|r(x)\|_2^2$. An n-point Lobatto quadrature rule is exact for polynomials of degree $2n-3$, so for an asymptotically correct estimate of our residual we would need at least a 7 point procedure, requiring 4 extra function evaluations per subinterval - 5 if we include the evaluation necessary to calculate the more accurate $\overline{f}_{i+\frac{1}{2}}$.

x_i	0.0	0.08489	0.26558	0.5	0.73442	0.91511	1.0
w_i	0.04762	0.27683	0.43175	0.4876	0.43175	0.27683	0.04762

Table A.1: Abscissa and Weights of 7-point Lobatto Quadrature

A.4 IMPLEMENTATION

From the start bvp6c was intended to be a direct extension to bvp4c and as such implementation would be almost identical and routines would only be changed where necessary to maintain sixth-order accuracy. It follows then that we would wish to maintain the developments of Keirzenka and Shampine in their code, namely the ability to solve problems with unknown parameters and general two-point BCS in addition to not requiring (although allowing) user entered partial derivatives and the vectorization of $f(x, y)$. Fortunatly the extension from bvp4c to bvp6c affects these areas very little and often little alteration was necessary. Here we discuss some of the larger impacts that the higher order extension had on the implementation of the method.

A.4.1 Collocation Equations and Quadrature

The collocation method applied to (A.1.A.2) with a mesh $a = x_0 < x_1 < ... < x_N = b$ is evaluated by solving the algebraic equations

$$\Phi(X, Y) = 0, \tag{A.4}$$

where

$$\begin{aligned}
X &= [x_0, x_1, ..., x_N]^T \\
Y &= [y_0, y_1, ..., y_N, p]^T \\
\Phi_0(X, Y) &= g(y_0, y_N, p) \\
\Phi_{i+1}(X, Y) &= y_{i+1} - y_i - \frac{h_i}{90}\left[7f_i + 32f_{i+\frac{1}{4}} + 12f_{i+\frac{1}{2}} + 32f_{i+\frac{3}{4}} + 7f_{i+1}\right], \quad i{=}0,\ 1,\ ...\,N{-}1
\end{aligned}$$

Suppose we interpolate the given data to find the function values at the midpoint.

$$\begin{aligned}
S(x_{i+\frac{1}{2}}) &= \frac{1}{2}(y_{i+1} + y_i) - \frac{h_i}{24}\{f_{i+1} - f_i + 4\left[f_{i+\frac{3}{4}} - f_{i+\frac{1}{4}}\right]\} = \overline{y}_{i+\frac{1}{2}} \\
S'(x_{i+\frac{1}{2}}) &= \overline{f}_{i+\frac{1}{2}}
\end{aligned}$$

We see that our interpolant is exact for both $\overline{y}$ and $\overline{f}$ (the improved approximations) at the midpoint, and as such $r(x_{i+\frac{1}{2}}) = S'(x_{i+\frac{1}{2}}) - f(x_{i+\frac{1}{2}}, S(x_{i+\frac{1}{2}})) = \overline{f}_{i+\frac{1}{2}} - \overline{f}_{i+\frac{1}{2}} = 0$ needs no calculation. Similarly the residual gives zero contributions at the two end points.

In [1] similar analysis to above the above is used in order to show that the residual at the midpoint in bvp4c gave an approximation to how well the collocation equations (A.4) are satisfied. We would like to use a similar result here for the quarter-points, but since we our interpolant makes use of the more accurate value $\overline{y_{\frac{1}{2}}}$, the result does not carry over.

We therefore choose a 7-point Lobatto quadrature method to calculate the residual of the interval, since this will make use of both of the end and the mid-points of which we know to give a zero contribution. The four remaining points $\{x_L\}$ given by this method must then be interpolated to find $S(x_L)$ and $S'(x_L)$ and the ODE function evaluated to find $f(x_L, S(x_L))$ from which we compute the residual.

A.4.2 Jacobians

We solve (A.4) using a simplified Newton (chord) method and hence require the global Jacobian $\partial\Phi/\partial Y$. The form of the Jacobian here is much more complex than for the Simpson method and much more expensive in terms of computation. One calculates the Jacobian as

$$\begin{aligned}
\frac{\partial\Phi^{[6]}_{i+1}}{\partial y_i} = I + \frac{h}{90} & \left[7\frac{\partial f_i}{\partial y} + 27\frac{\partial f_{i+\frac14}}{\partial y} + 6\frac{\partial f_{i+\frac12}}{\partial y} + 5\frac{\partial f_{i+\frac34}}{\partial y}\right] \quad + \\
\frac{h^2}{360} & \left[\frac{\partial f_{i+\frac12}}{\partial y}\left(27\frac{\partial f_{i+\frac14}}{\partial y} - 5\frac{\partial f_{i+\frac34}}{\partial y}\right) + \left(18\frac{\partial f_{i+\frac14}}{\partial y} - 10\frac{\partial f_{i+\frac12}}{\partial y} + 6\frac{\partial f_{i+\frac34}}{\partial y}\right)\frac{\partial f_i}{\partial y}\right] \\
+\frac{h^3}{240} & \left[\frac{\partial f_{i+\frac12}}{\partial y}\left(3\frac{\partial f_{i+\frac14}}{\partial y} - \frac{\partial f_{i+\frac34}}{\partial y}\right)\frac{\partial f_i}{\partial y}\right]. \\
\frac{\partial\Phi^{[6]}_{i+1}}{\partial y_{i+1}} = -I + \frac{h}{90} & \left[5\frac{\partial f_{i+\frac14}}{\partial y} + 6\frac{\partial f_{i+\frac12}}{\partial y} + 27\frac{\partial f_{i+\frac34}}{\partial y} + 7\frac{\partial f_{i+1}}{\partial y}\right] \quad + \\
\frac{h^2}{360} & \left[\frac{\partial f_{i+\frac12}}{\partial y}\left(5\frac{\partial f_{i+\frac14}}{\partial y} - 27\frac{\partial f_{i+\frac34}}{\partial y}\right) - \left(6\frac{\partial f_{i+\frac14}}{\partial y} - 10\frac{\partial f_{i+\frac12}}{\partial y} + 18\frac{\partial f_{i+\frac34}}{\partial y}\right)\frac{\partial f_{i+1}}{\partial y}\right] \\
-\frac{h^3}{240} & \left[\frac{\partial f_{i+\frac12}}{\partial y}\left(\frac{\partial f_{i+\frac14}}{\partial y} - 3\frac{\partial f_{i+\frac34}}{\partial y}\right)\frac{\partial f_{i+1}}{\partial y}\right]. \\
\left[\frac{\partial\Phi^{[6]}_{i+1}}{\partial y_i}\right]_{p_1,\ldots p_N} = \tfrac{h}{90} & \left[7\left(\frac{\partial f_i}{\partial\lambda} + \frac{\partial f_{i+1}}{\partial\lambda}\right) + 32\left(\frac{\partial f_{i+\frac14}}{\partial\lambda} + \frac{\partial f_{i+\frac34}}{\partial\lambda}\right) + 12\frac{\partial f_{i+\frac12}}{\partial\lambda}\right] \quad + \\
\tfrac{h_2}{180} & \left[\frac{\partial f_{i+\frac14}}{\partial\lambda}\left(9\frac{\partial f_i}{\partial\lambda} - 3\frac{\partial f_{i+1}}{\partial\lambda}\right) + \frac{\partial f_{i+\frac34}}{\partial\lambda}\left(3\frac{\partial f_i}{\partial\lambda} - 9\frac{\partial f_{i+1}}{\partial\lambda}\right)\right. \\
& \left. + \frac{\partial f_{i+\frac12}}{\partial\lambda}\left(16\left(\frac{\partial f_{i+\frac14}}{\partial\lambda} - \frac{\partial f_{i+\frac34}}{\partial\lambda}\right) + 5\left(\frac{\partial f_{i+1}}{\partial\lambda} - \frac{\partial f_i}{\partial\lambda}\right)\right)\right] \quad + \\
\tfrac{h_3}{240} & \left[\frac{\partial f_{i+\frac12}}{\partial\lambda}\frac{\partial f_{i+\frac14}}{\partial\lambda}\left(3\frac{\partial f_i}{\partial\lambda} - \frac{\partial f_{i+1}}{\partial\lambda}\right) + \frac{\partial f_{i+\frac12}}{\partial\lambda}\frac{\partial f_{i+\frac34}}{\partial\lambda}\left(3\frac{\partial f_{i+1}}{\partial\lambda} - \frac{\partial f_i}{\partial\lambda}\right)\right]
\end{aligned}$$

In bvp4c Kierzenka and Shampine justify their decision to approximate the local internal Jacobian $J_{i+\frac12}$ by an average when it is not changing rapidly. Here we have two additional internal Jacobians to calculate ($J_{i+\frac14}$ and $J_{i+\frac34}$), making such an approximation even more justified and demonstrating that although a user supplied Jacobian is not necessary, it could greatly computation speed.

A.4.3 Mesh Selection

We maintain the style of mesh selection used in bvp4c, but since bvp6c is a higher order method, introducing/removing points will have a greater affect on the size of the residual. Mesh points are removed if the residual on an subinterval (and an estimate of the new residual on the coarser mesh) is deemed small enough, or if the residual is too large more points are added. Although having not tested the alternatives extensively, we have followed Kierzenka and Shampine in introducing no more than two mesh points at a time.

A.5 RESULTS / EXAMPLES

We now seek to compare the accuracy an efficiency of our new method with the original version bvp4c. To this end we reconsider some of those problems discussed in [1] as well as introducing a new set of problems known as the Cash-Wright BVP problem suite [4].

For the first family of problems we consider analytic solutions are not available, as such we simply compare results between the two solutions in addition to other important factors, such as solution time, number of mesh points and number of function evaluations. By defining ‘reltol’ and ‘abstol’ equal to some tolerance, we should expect the solution of both methods, and hence their difference, to comply to this tolerance.

A.5.1 measles

The first example is taken from Ascher et al. [6] 1.10 and models the spread of measles via the differential equations

$$\begin{aligned} y_1' &= \mu - \beta(t) y_1 y_3 \\ y_2' &= \beta(t) y_1 y_3 - \frac{y_2}{\lambda} \\ y_3' &= \frac{y_2}{\lambda} - \frac{y_3}{\nu} \end{aligned}$$

where $\beta = 1575(1+\cos(2\pi t))$ and $\mu = 0.02, \lambda = 0.0279$ and $\nu = 0.01$ are given.
The solution should also satisfy the (nonseperated) periodicity conditions $y(0) = y(1)$.

Since, for the end user use of bvp4 and 6c is identical, it is straightforward to adapt the code given in [1] to test both methods and doing so gives results below.

	bvp4c			bvp6c			
tol	time	mesh	maxres	time	mesh	maxres	error
1e-3	0.1880	20	1.4e-4	0.391	22	9.8e-4	4.07e-4
1e-6	0.5000	130	3.3e-7	0.343	66	3.3e-7	1.25e-4
1e-9	1.3590	385	9.8e-10	0.453	114	1.6e-10	2.29e-8
1e-12	5.4060	2379	1.0e-12	0.953	211	9.4e-13	9.57e-11

What we see above shall come to be a consistent result, that both methods perform similarly when only a small degree of accuracy is required, but as this is increased bvp6c will solve using much fewer mesh points in a shorter period of time. Quantitatively, here bvp6c uses factor 10 fewer mesh points and solves in a quarter of the time.

Explicit results cannot yet be draw as to the accuracy of the solutions since the analytic solution is not readily available. The ’error’ given here is merely the difference between an interpolation of the results from the two solvers.

A.5.2 injection

The second example is also taken from [6] 1.4 and describes flow in a long vertical channel with fluid injection:

$$\begin{aligned} f''' - R\left[(f')^2 - ff''\right] &= 0 \\ h'' - Rfh' + 1 &= 0 \\ \theta' + Pf\theta &= 0 \end{aligned}$$

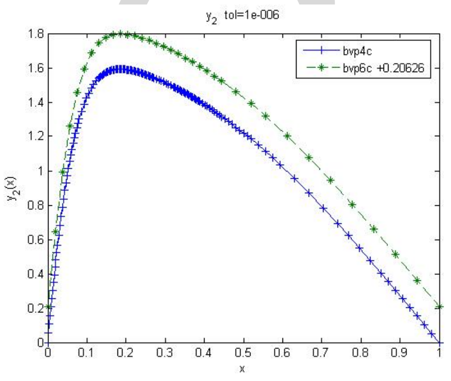
where R is a Reynolds number and $P = 0.7R$. The parameter A is unknown, and as such there are eight boundary conditions;

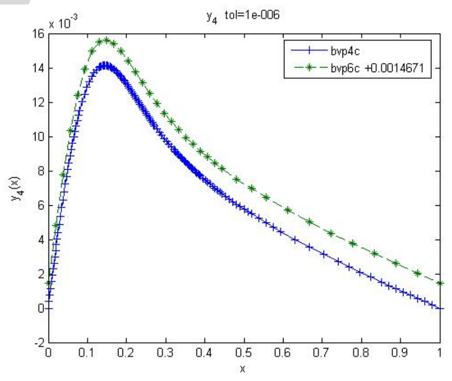
$$\begin{gathered} f(0) = f'(0) = 0,\ f(1) = 1,\ f'(1) = 1, \\ h(0) = h(1) = 0,\ \ \theta(0) = 0,\ \ \theta(1) = 1 \end{gathered}$$

Again, we implement this problem (here taking R=100) and obtain the following results.

	bvp4c			bvp6c			
tol	time	mesh	maxres	time	mesh	maxres	error
1e-3	0.2500	25	5.8e-4	0.234	24	2.6e-5	1.26e-3
1e-6	0.7820	152	9.9e-7	0.313	36	8.9e-7	1.16e-6
1e-9	5.1720	910	9.9e-10	0.813	101	8.3e-10	7.52e-10
1e-12	65.9680	6878	1.0e-12	2.391	346	9.9e-13	8.19e-13

Fortunately we see here that the difference between our two solutions behaves as expected, and for the smaller tolerances the efficiency in the new method is even more pronounced.





We include the above graphs of f' and h (note the solutions have been shifted so as to see results from both methods) to show that solution takes the same form as that given in [1]. This also gives an indication of how the mesh points for both methods cluster around points where the solution changes rapidly, as this is where the calculated residual would be greatest.

A.5.3 shockbvp

This BVP used in Example 9.2 of Ascher et al. [6] to illustrate mesh selection solves

$$\epsilon y'' + xy' = -\epsilon\pi^2 cos(\pi x) - (\pi x) sin(\pi x)$$

with boundary conditions $y(-1) = -2,\ y(1) = 0$.

We use this problem here to demonstrate a number of points, including the affect of shocks, continuation analytic jacobians and vectorisation of the problem.

	tol=1e-3				tol=1e-6				tol=1e-9
Finalϵ	1e-03	1e-04	1e-05		1e-03	1e-04	1e-05		1e-05
bvp4c	0.32	0.78	2.68		1.68	3.44	6.11		34.45
bvp6c	0.40	1.11	16.53		0.72	2.68	32.89		48.39
vbvp4c	0.34	0.77	2.73		1.67	3.35	6.10		33.65
vbvp6c	0.36	0.93	12.64		0.60	2.15	23.48		31.39
ajbvp4c	0.18	0.41	1.39		0.94	1.85	3.34		18.81
ajbvp6c	0.27	0.51	1.15		0.50	1.29	4.96		16.95
vjbvp4c	0.11	0.24	0.77		0.47	0.91	1.67		9.45
vjbvp6c	0.16	0.28	0.57		0.25	0.60	2.1		6.96

Table A.2: Time (secs) to solve shockbvp using continuation

The above table, although quite monstrous, contains some results worthy of comment. Following in the steps of the bvp4c analysis in [1] the shockbvp problem is solved using continuation and the times shown are not the intermediate results, but the total time taken to reach a solution for the given value of ϵ. vbvp represents the results from a vectorised version of the problem, ajbvp from using an analytic Jacobian and vjbvp from combining the two. As a general result on both solvers it can be seen that vectorisation has a much smaller affect on the result than that of analytic Jacobians, although it is when used in tandem that the time is most significantly decreased. This is to be expected as the volatility of the solution around the shock layer would oblige the bvp framework to recalculate all of the Jacobians between each mesh point - which is expensive to calculate numerically. Since bvp6c uses the higher-order finite difference method, one would also expect it to perform less efficiently on a shock problem as is clearly the case here. However it has been shown that if analytic Jacobians are available that bvp6c will perform favourably.

A.6 Cash BVP test suite

bvp6c was tested extensively using the Cash-Wright BVP test suite [4] - a collection of linear and nonlinear BVP problems designed to test the performance and robustness of a numerical solver. Since the majority of the problems posed do not have explicit analytic solutions, numerical solutions were computed using the recently developed Capper-Moore twelfth-order MIRK method [12],[13] to a high degree of accuracy and assumed sufficiently accurate for error analysis.Solutions to each of the 32 problems are computed using the two bvpc solvers with a range of specified tolerances to compare amongst other properties their accuracy and efficiency. For simplicity we choose the same value for relative and absolute is chosen, and an initial guess of zero (where possible) on a grid of 33 equally spaced points. Above can be seen the results of applying both methods to the 32 given

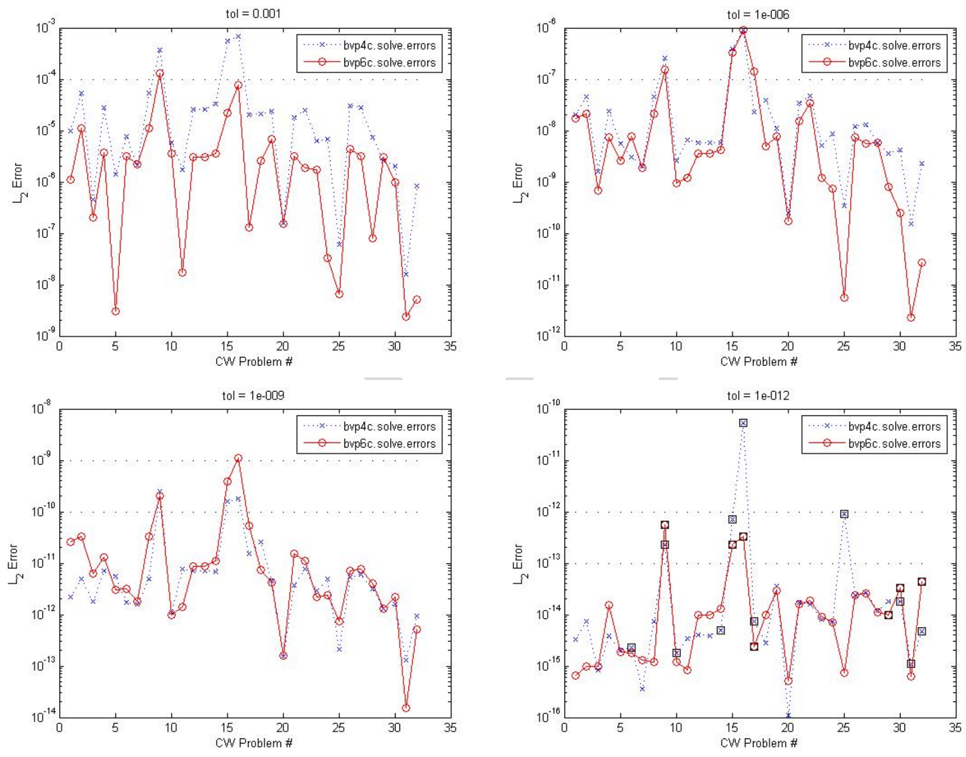


Figure A.1: L_2 errors from bvpc solvers on CW32 problems with varying tolerance.

BVPs as the tolerance levels are decreased simultaneously. To analyse the error we choose to consider those points at which the solvers return solutions rather than an interpolation at fixed values and take an averaged L_2 norm over each of the mesh points, i.e.

$$\left(\frac{1}{N}\sum_{i=1}^{N}\left[y_i - y(x_i)\right]^2\right)^{1/2}$$

For the vast majority of the problems it can be seen that bvp6c obtains a solution to the specified accuracy in the average L_2 norm (N.B. it holds also in the L_∞ sense) and in those cases which it doesn't, it still outperforms the bvp4c algorithm.

APPENDIX B

Sixth-order Jacobians

$$
\begin{aligned}
\frac{\partial \Phi_{i+1}^{[6]}}{\partial y_i} &= I + \frac{h_i}{90}\left[7\left.\frac{\partial f}{\partial y}\right|_{y_i} + 32\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{4}}}\frac{\partial y_{i+\frac{1}{4}}}{\partial y_i} + 12\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{2}}}\frac{\partial y_{i+\frac{1}{2}}}{\partial y_i} + 32\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{3}{4}}}\frac{\partial y_{i+\frac{3}{4}}}{\partial y_i} + 7\left.\frac{\partial f}{\partial y}\right|_{y_i}\frac{\partial y_{i+1}}{\partial y_i}\right] \\
&= I + \frac{h_i}{90}\left[7\left.\frac{\partial f}{\partial y}\right|_{y_i} + \frac{32}{64}\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{4}}}\left(54 + 9h_i\left.\frac{\partial f}{\partial y}\right|_{y_i}\right) + \frac{32}{64}\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{3}{4}}}\left(10 + 3h_i\left.\frac{\partial f}{\partial y}\right|_{y_i}\right) + \right. \\
&\quad \left.\frac{12}{24}\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{2}}}\left(12 - h_i\left[5\left.\frac{\partial f}{\partial y}\right|_{y_i} - 16\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{4}}}\frac{\partial y_{i+\frac{1}{4}}}{\partial y_i} + 16\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{3}{4}}}\frac{\partial y_{i+\frac{3}{4}}}{\partial y_i}\right]\right) + 0\right] \\
&= I + \frac{h_i}{90}\left[7\left.\frac{\partial f}{\partial y}\right|_{y_i} + 27\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{4}}} + 6\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{2}}} + 5\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{3}{4}}}\right] + \\
&\quad \frac{h_i^2}{90}\left[\frac{9}{2}\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{4}}}\left.\frac{\partial f}{\partial y}\right|_{y_i} - \frac{5}{2}\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{2}}}\left.\frac{\partial f}{\partial y}\right|_{y_i} + \frac{27}{4}\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{2}}}\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{4}}} - \frac{5}{4}\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{2}}}\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{3}{4}}} + \frac{3}{2}\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{3}{4}}}\left.\frac{\partial f}{\partial y}\right|_{y_i}\right] + \\
&\quad \frac{h_i^3}{90}\left[\frac{9}{8}\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{2}}}\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{4}}}\left.\frac{\partial f}{\partial y}\right|_{y_i} - \frac{3}{8}\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{2}}}\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{3}{4}}}\left.\frac{\partial f}{\partial y}\right|_{y_i}\right] \\
&= I + \frac{h_i}{90}\left[7\left.\frac{\partial f}{\partial y}\right|_{y_i} + 27\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{4}}} + 6\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{2}}} + 5\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{3}{4}}}\right] + \\
&\quad \frac{h_i^2}{360}\left[\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{2}}}\left(27\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{4}}} - 5\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{3}{4}}}\right) + \left(18\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{4}}} - 10\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{2}}} + 6\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{3}{4}}}\right)\left.\frac{\partial f}{\partial y}\right|_{y_i}\right] \\
&\quad + \frac{h_i^3}{240}\left[\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{2}}}\left(3\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{4}}} - \left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{3}{4}}}\right)\left.\frac{\partial f}{\partial y}\right|_{y_i}\right].
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial \Phi^{[6]}_{i+1}}{\partial y_{i+1}} &= I + \frac{h_i}{90}\left[7\left.\frac{\partial f}{\partial y}\right|_{y_i}\frac{\partial y_i}{\partial y_{i+1}} + 32\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{4}}}\frac{\partial y_{i+\frac{1}{4}}}{\partial y_{i+1}} + 12\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{2}}}\frac{\partial y_{i+\frac{1}{2}}}{\partial y_{i+1}} + 32\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{3}{4}}}\frac{\partial y_{i+\frac{3}{4}}}{\partial y_{i+1}} + 7\left.\frac{\partial f}{\partial y}\right|_{y_{i+1}}\right] \\
&= I + \frac{h_i}{90}\left[0 + \frac{32}{64}\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{4}}}\left(10 - 3h_i\left.\frac{\partial f}{\partial y}\right|_{y_{i+1}}\right) + \frac{32}{64}\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{3}{4}}}\left(54 - 93h_i\left.\frac{\partial f}{\partial y}\right|_{y_{i+1}}\right) + \right. \\
&\quad \left. \frac{12}{24}\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{2}}}\left(12 + h_i\left[5\left.\frac{\partial f}{\partial y}\right|_{y_{i+1}} + 16\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{4}}}\frac{\partial y_{i+\frac{1}{4}}}{\partial y_{i+1}} - 16\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{3}{4}}}\frac{\partial y_{i+\frac{3}{4}}}{\partial y_{i+1}}\right]\right) + 7\left.\frac{\partial f}{\partial y}\right|_{y_{i+1}}\right] \\
&= I + \frac{h_i}{90}\left[5\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{4}}} + 6\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{2}}} + 27\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{3}{4}}} + 7\left.\frac{\partial f}{\partial y}\right|_{y_{i+1}}\right] \quad + \\
&\quad \frac{h_i^2}{360}\left[\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{2}}}\left(5\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{4}}} - 27\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{3}{4}}}\right) + \left(-6\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{4}}} + 10\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{2}}} - 18\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{3}{4}}}\right)\left.\frac{\partial f}{\partial y}\right|_{y_{i+1}}\right] \\
&\quad + \frac{h_i^3}{240}\left[\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{2}}}\left(-\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{4}}} + 3\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{3}{4}}}\right)\left.\frac{\partial f}{\partial y}\right|_{y_{i+1}}\right].
\end{aligned}
$$

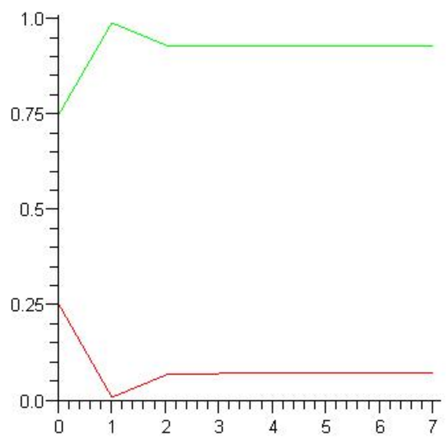
$$
\begin{aligned}
\left[\frac{\partial \Phi^{[6]}_{i+1}}{\partial y_i}\right]_{p_1,\ldots p_N} = \frac{h}{90} \quad & \left[7\left(\left.\frac{\partial f}{\partial \lambda}\right|_{y_i} + \left.\frac{\partial f}{\partial \lambda}\right|_{y_{i+1}}\right) + 32\left(\left.\frac{\partial f}{\partial \lambda}\right|_{y_{i+\frac{1}{4}}} + \left.\frac{\partial f}{\partial \lambda}\right|_{y_{i+\frac{3}{4}}}\right) + 12\left.\frac{\partial f}{\partial \lambda}\right|_{y_{i+\frac{1}{2}}}\right] \quad + \\
\frac{h_2}{180} \quad & \left[\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{4}}}\left(9\left.\frac{\partial f}{\partial \lambda}\right|_{y_i} - 3\left.\frac{\partial f}{\partial \lambda}\right|_{y_{i+1}}\right) + \left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{3}{4}}}\left(3\left.\frac{\partial f}{\partial \lambda}\right|_{y_i} - 9\left.\frac{\partial f}{\partial \lambda}\right|_{y_{i+1}}\right)\right. \\
& \left. + \left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{2}}}\left(16\left(\left.\frac{\partial f}{\partial \lambda}\right|_{y_{i+\frac{1}{4}}} - \left.\frac{\partial f}{\partial \lambda}\right|_{y_{i+\frac{3}{4}}}\right) + 5\left(\left.\frac{\partial f}{\partial \lambda}\right|_{y_{i+1}} - \left.\frac{\partial f}{\partial \lambda}\right|_{y_i}\right)\right)\right] \quad + \\
\frac{h_3}{240} \quad & \left[\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{2}}}\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{4}}}\left(3\left.\frac{\partial f}{\partial \lambda}\right|_{y_i} - \left.\frac{\partial f}{\partial \lambda}\right|_{y_{i+1}}\right) + \left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{1}{2}}}\left.\frac{\partial f}{\partial y}\right|_{y_{i+\frac{3}{4}}}\left(3\left.\frac{\partial f}{\partial \lambda}\right|_{y_{i+1}} - \left.\frac{\partial f}{\partial \lambda}\right|_{y_i}\right)\right]
\end{aligned}
$$

APPENDIX C

Gauss-Hale Quadrature

```
> #ITERATIVE PROCEDURE TO FIND 'GAUSS-HALE' ABSCISSA AND WEIGHTS FOR BVP6c
#NOTE - GENERAL CONVERGENCE IS NOT GUARENTEED!
#SCHEME DESIGNED SPECIFICALLY FOR GUASS-HALE7 WITH 0,0.25,0.5,0.75,1.0.
restart;
Digits:=50:
> #NUMBER OF PREASSIGNED ABSCISSA
m:=5:
#NUMBER OF ABSCISSA TO BE DETERMINED
n:=2:
#TOLERANCE IN SOLUTIONS
tol:=1e-32:
#MAXIMUM ITERATIONS
maxi:=100:
> #PREASSIGNED ABSCISSA
z[1]:=0:
z[2]:=1/4:
z[3]:=1/2:
z[4]:=3/4:
z[5]:=1:
> #INITIAL GUESS FOR NEW NODES
x[1]:=1/4:
x[2]:=3/4:
> Z:=[seq(z[i],i=1..m)]:
Z:=evalf(Z):
X:=[seq(x[i],i=1..n)]:
X=evalf(X):
> Omega(x):=proc(Z,i)
        if i=0 then
                L:=seq(j, j=1..m)
      else
                L:=seq(j, j=1..i-1),seq(j, j=i+1..m)
        end;
        mul( x-Z[k], k=L);
end proc:
w(x):=proc(X,i)
      if i=0 then
```

```
                L:=seq(j, j=1..n)
          else
                L:=seq(j, j=1..i-1),seq(j, j=i+1..n)
            end;
            mul( x-X[k], k=L);
end proc:
psi:=proc(X,i)
      int(Omega(x)(Z,0)*w(x)(X,i)^2*x,x=0..1)/int(Omega(x)(Z,0)*w(x)(X,i)^2,x=0..1);
end proc:
> #ITERATION SETUP
i:=0:   continue:=1:
store:=array(1..n):
for j from 1 to n do   store[j]:=[0,X[j]]: end do:
> #ITERATION TO DETERMINE NEW ABSCISSA
while continue=1 do
        i:=i+1;
        for j from 1 to n do
                Y[j]:=psi(X,j);
        end;
        if abs(X[1]-Y[1])<tol then continue:=0; end;
        if i>maxi then continue:=0; end;
        for j from 1 to n do
                X[j]:=Y[j];
        end;
        for j from 1 to n do
                store[j]:=store[j],[i,X[j]]:
        end do:
end:
> #CALCULATE WEIGHTS OF ASSIGNED NODES
for i from 1 to m do
        a[i]:=int(Omega(x)(Z,i)*w(x)(X,0),x=0..1)/ ...
                        (eval(Omega(x)(Z,i),x=Z[i])*eval(w(x)(X,0),x=Z[i]));
end:
> #CALCULATE WEIGHTS FOR GAUSSIAN NODEs
for i from 1 to n do
        b[i]:=int(Omega(x)(Z,0)*w(x)(X,i)^2,x=0..1)/ ...
                        (eval(Omega(x)(Z,0),x=X[i])*eval(w(x)(X,i),x=X[i])^2);
end:
> #PLOT CONVERGENCE OF ITERATES
plot_data:=[seq([store[j]],j=1..n)]:
plot(plot_data);
```



```
> #PRINT RESULTS
printf("\tx[i] \t\t\t w[i]\nAssigned Points\n");
for i from 1 to m do
        printf("\t%0.16f \t %0.16f\n",Z[i],a[i]);
end;
```

```
printf("Calculated points points\n");
for i from 1 to n do
        printf("\t%0.16f \t %0.16f\n",X[i],b[i]);
end;


        x[i]                        w[i]
Assigned Points
        0.0000000000000000       0.0182539682539683
        0.2500000000000000       0.2241926655719759
        0.5000000000000000       0.2632034632034632
        0.7500000000000000       0.2241926655719759
        1.0000000000000000       0.0182539682539683
Calculated points points
        0.0718255807111624       0.1259516345723242
        0.9281744192888376       0.1259516345723242
> #CREATE SIMPLE TEST POLYNOMIALS
 p[9]:=proc(x)  x^2+x^9; end proc:
 p[10]:=proc(x) x+x^5-x^10; end proc:
> #IMPLEMENT GAUSS-HALE QUADRATURE
Q:=proc(F,x)
        sum(a[l]*F(Z[l]), l=1..m) + sum(b[l]*F(X[l]), l=1..n)
end proc:
> #ERRORS
errP[9]:=int(p[9](x),x=0..1)-evalf(Q(p[9],0)):
printf("Error in Quadrature of x^9:\t%g\n",errP[9]);
errP[10]:=int(p[10](x),x=0..1)-evalf(Q(p[10],0)):
printf("Error in Quadrature of x^10:\t%g\n",errP[10]);

Error in Quadrature of x^9:     0
Error in Quadrature of x^10:    -5.26094e-07
```

APPENDIX D

Code Listings

D.1 bvp6c

```
function sol = bvp6c(ode, bc, solinit, options, varargin)
%BVP6C  Solve boundary value problems for ODEs by collocation.
%       (6th order extension of BVP4C by Kierzenka and Shampine)
%   SOL = BVP6C(ODEFUN,BCFUN,SOLINIT) integrates a system of ordinary
%   differential equations of the form y' = f(x,y) on the interval [a,b],
%   subject to general two-point boundary conditions of the form
%   bc(y(a),y(b)) = 0. ODEFUN is a function of two arguments: a scalar X
%   and a vector Y. ODEFUN(X,Y) must return a column vector representing
%   f(x,y). BCFUN is a function of two vector arguments. BCFUN(YA,YB) must
%   return a column vector representing bc(y(a),y(b)). SOLINIT is a structure
%   with fields named
%       x -- ordered nodes of the initial mesh with
%            SOLINIT.x(1) = a, SOLINIT.x(end) = b
%       y -- initial guess for the solution with SOLINIT.y(:,i)
%            a guess for y(x(i)), the solution at the node SOLINIT.x(i)
%
%   BVP6C produces a solution that is continuous on [a,b] and has a
%   continuous first derivative there. The solution is evaluated at points
%   XINT using the output SOL of BVP6C and the function DEVAL:
%   YINT = DEVAL(SOL,XINT). The output SOL is a structure with
%       SOL.x  -- mesh selected by BVP6C
%       SOL.y  -- approximation to y(x) at the mesh points of SOL.x
%       SOL.solver -- 'bvp6c'
%   If specified in BVPSET, SOL may also contain
%       SOL.yp -- approximation to y'(x) at the mesh points of SOL.x
%       sol.ypmid approximations to y'(x) at interior points of SOL.x
%
%   SOL = BVP6C(ODEFUN,BCFUN,SOLINIT,OPTIONS) solves as above with default
%   parameters replaced by values in OPTIONS, a structure created with the
%   BVPSET function. To reduce the run time greatly, use OPTIONS to supply
%   a function for evaluating the Jacobian and/or vectorize ODEFUN.
%   See BVPSET for details and SHOCKBVP for an example that does both.
%
%   SOL = BVP6C(ODEFUN,BCFUN,SOLINIT,OPTIONS,P1,P2...) passes constant, known
%   parameters P1, P2... to the functions ODEFUN and BCFUN, and to all
%   functions specified in OPTIONS. Use OPTIONS = [] as a place holder if
%   no options are set.
%
%   Some boundary value problems involve a vector of unknown parameters p
%   that must be computed along with y(x):
%       y' = f(x,y,p)
%       0  = bc(y(a),y(b),p)
%   For such problems the field SOLINIT.parameters is used to provide a guess
%   for the unknown parameters. On output the parameters found are returned
%   in the field SOL.parameters. The solution SOL of a problem with one set
%   of parameter values can be used as SOLINIT for another set. Difficult BVPs
%   may be solved by continuation: start with parameter values for which you can
%   get a solution, and use it as a guess for the solution of a problem with
%   parameters closer to the ones you want. Repeat until you solve the BVP
%   for the parameters you want.
%
%   The function BVPINIT forms the guess structure in the most common
%   situations:  SOLINIT = BVPINIT(X,YINIT) forms the guess for an initial mesh X
%   as described for SOLINIT.x and YINIT either a constant vector guess for the
%   solution or a function that evaluates the guess for the solution
%   at any point in [a,b]. If there are unknown parameters,
%   SOLINIT = BVPINIT(X,YINIT,PARAMS) forms the guess with the vector PARAMS of
%   guesses for the unknown parameters.
%
%   BVP6C solves a class of singular BVPs, including problems with
%   unknown parameters p, of the form
%       y' = S*y/x + f(x,y,p)
%       0  = bc(y(0),y(b),p)
%   The interval is required to be [0, b] with b > 0.
%   Often such problems arise when computing a smooth solution of
%   ODEs that result from PDEs because of cylindrical or spherical
%   symmetry. For singular problems the (constant) matrix S is
%   specified as the value of the 'SingularTerm' option of BVPSET,
%   and ODEFUN evaluates only f(x,y,p). The boundary conditions
%   must be consistent with the necessary condition S*y(0) = 0 and
%   the initial guess should satisfy this condition.
%
%   BVP6C can solve multipoint boundary value problems.  For such problems
%   there are boundary conditions at points in [a,b]. Generally these points
%   represent interfaces and provide a natural division of [a,b] into regions.
%   BVP6C enumerates the regions from left to right (from a to b), with indices
%   starting from 1.  In region k, BVP6C evaluates the derivative as
%   YP = ODEFUN(X,Y,K).  In the boundary conditions function,
%   BCFUN(YLEFT,YRIGHT), YLEFT(:,k) is the solution at the 'left' boundary
%   of region k and similarly for YRIGHT(:,k).  When an initial guess is
%   created with BVPINIT(XINIT,YINIT), XINIT must have double entries for
%   each interface point. If YINIT is a function, BVPINIT calls Y = YINIT(X,K)
%   to get an initial guess for the solution at X in region k. In the solution
%   structure SOL returned by BVP6C, SOL.x has double entries for each interface
%   point. The corresponding columns of SOL.y contain the 'left' and 'right'
%   solution at the interface, respectively. See THREEBVP for an example of
%   solving a three-point BVP.
%
%   Example
%         solinit = bvpinit([0 1 2 3 4],[1 0]);
%         sol = bvp6c(@twoode,@twobc,solinit);
%     solve a BVP on the interval [0,4] with differential equations and
%     boundary conditions computed by functions twoode and twobc, respectively.
%     This example uses [0 1 2 3 4] as an initial mesh, and [1 0] as an initial
%     approximation of the solution components at the mesh points.
%         xint = linspace(0,4);
%         yint = deval(sol,xint);
%     evaluate the solution at 100 equally spaced points in [0 4]. The first
%     component of the solution is then plotted with
%         plot(xint,yint(1,:));
%   For more examples see TWOBVP, FSBVP, SHOCKBVP, MAT4BVP, EMDENBVP, THREEBVP.
%
%   See also BVPSET, BVPGET, BVPINIT, DEVAL, @.

%   BVP6C is a finite difference code that implements the Cash-Singhal 6
%   formula. This is a collocation formula and the collocation
%   polynomial provides a C1-continuous solution that is sixth order
%   accurate uniformly in [a,b]. (For multipoint BVPs, the solution is
%   C1-continuous within each region, but continuity is not automatically
%   imposed at the interfaces.) Mesh selection and error control are based
%   on the residual of the continuous solution. Analytical condensation is
%   used when the system of algebraic equations is formed.

%   BVP4C
%    Jacek Kierzenka and Lawrence F. Shampine
%    Copyright 1984-2003 The MathWorks, Inc.
%    $Revision: 1.21.4.8 $  $Date: 2003/12/26 18:08:47 $
%   BVP6C Modification
%    Nick Hale  Imperial College London
%    $Date: 12/06/2006 $


%bvp6c extras
if nargin < 4
    options= [];
end
MaxNewPts = bvpget(options,'MaxNewPts',2);
solnpts = bvpget(options,'Xint',[]);
if isempty(solnpts)
    slopeout = strcmp(bvpget(options,'SlopeOut','on'),'on');
else
    slopeout = 0;
    if strcmp(bvpget(options,'SlopeOut'),'on');
          warning('MATLAB:bvp6c:SolnPtsNoSlopeOut', ...
              ['SOL will not contain slope data \n', ...
              '(solution points were specified in BVPSET)']);
    end
end
```

```
[m,n]=size(solnpts);
if min(m,n) > 1
    error('MATLAB:bvp6c:SolnPtsNotVector',...
        'The required solution points must be a vector (or []).');
end
if m>n
    solnpts=solnpts';
end

% check input parameters
if nargin < 3
  error('MATLAB:bvp6c:NotEnoughInputs', 'Not enough input arguments.')
end

if ~isstruct(solinit)
  error('MATLAB:bvp6c:SolinitNotStruct',...
        'The initial profile must be provided as a structure.')
elseif ~isfield(solinit,'x')
  error('MATLAB:bvp6c:NoXInSolinit',...
        ['The field ''x'' not present in ''' inputname(3) '''.'])
elseif ~isfield(solinit,'y')
  error('MATLAB:bvp6c:NoYInSolinit',...
        ['The field ''y'' not present in ''' inputname(3) '''.'])
end

x = solinit.x(:)';     % row vector
y = solinit.y;

if isempty(x) || (length(x)<2)
  error('MATLAB:bvp6c:SolinitXNotEnoughPts',...
        [''''  inputname(3) '.x'' must contain at least the two end points.'])
else
  N = length(x);      % number of mesh points
end

xdir = sign(x(end)-x(1));
if any(xdir * diff(x) < 0)
  error ('MATLAB:bvp6c:SolinitXNotMonotonic',...
         ['The entries in ''' inputname(3) ...
          '.x'' must increase or decrease.']);
end

% a multi-point BVP?
mbcidx = find(diff(x) == 0);  % locate internal interfaces
ismbvp = ~isempty(mbcidx);

if isempty(y)
  error('MATLAB:bvp6c:SolinitYEmpty',...
        ['No initial guess provided in ''' inputname(3) '.y''.'])
end
if length(y(1,:)) ~= N
  error('MATLAB:bvp6c:SolXSolYSizeMismatch',...
        ['''' inputname(3) '.y'' not consistent with ''' ...
         inputname(3) '.x''.'])
end

n = size(y,1);  % size of the DE system
nN = n*N;

% stats
nODEeval = 0;
nBCeval = 0;

% set the options
if nargin<4
  options = [];
end

% parameters
knownPar = varargin;
unknownPar = isfield(solinit,'parameters');
if unknownPar
```

```
  npar = length(solinit.parameters(:));
  ExtraArgs = [{solinit.parameters(:)} knownPar];
else
  npar = 0;
  ExtraArgs = knownPar;
end
nExtraArgs = length(ExtraArgs);

% Check the argument functions
if ismbvp
  nregions = length(mbcidx) + 1;
  Lidx = [1, mbcidx+1];
  Ridx = [mbcidx, length(x)];
  nBCs = n * nregions + npar;
  testBC = feval(bc,y(:,Lidx),y(:,Ridx),ExtraArgs{:});
else
  nBCs = n + npar;
  testBC = feval(bc,y(:,1),y(:,end),ExtraArgs{:});
end
nBCeval = nBCeval + 1;
if length(testBC) ~= nBCs
  error('MATLAB:bvp6c:BCfunOuputSize',...
        ['The boundary condition function should return a column vector of' ...
        ' length %i'],nBCs);
end
if ismbvp
  region = 1;    % test region 1 only
  testODE = feval(ode,x(1),y(:,1),region,ExtraArgs{:});
else
  testODE = feval(ode,x(1),y(:,1),ExtraArgs{:});
end
nODEeval = nODEeval + 1;
if length(testODE) ~= n
  error('MATLAB:bvp6c:ODEfunOutputSize',...
        ['The derivative function should return a column vector of' ...
        ' length %i'],n);
end

% Get options and set the defaults
rtol = bvpget(options,'RelTol',1e-3);
if (length(rtol) ~= 1) || (rtol<=0)
  error('MATLAB:bvp6c:RelTolNotPos', 'RelTol must be a positive scalar.');
end
if rtol < 100*eps
  rtol = 100*eps;
  warning('MATLAB:bvp6c:RelTolIncrease','RelTol has been increased to %g.',rtol);
end
atol = bvpget(options,'AbsTol',1e-6);
if length(atol) == 1
  atol = atol(ones(n,1));
else
  if length(atol) ~= n
    error('MATLAB:bvp6c:SizeAbsTol',['Solving %s requires a scalar AbsTol, ' ...
           'or a vector AbsTol of length %d.'],funstring(ode),n);
  end
  atol = atol(:);
end
if any(atol<=0)
  error('MATLAB:bvp6c:AbsTolNotPos', 'AbsTol must be positive.');
end

threshold = atol/rtol;

% analytical Jacobians
Fjac = bvpget(options,'FJacobian');
BCjac = bvpget(options,'BCJacobian');
averageJac = isempty(Fjac);

Nmax = bvpget(options,'Nmax',floor(2500/n));
printstats = strcmp(bvpget(options,'Stats','off'),'on');

% vectorized (with respect to x and y)
```

```
xyVectorized = strcmp(bvpget(options,'Vectorized','off'),'on');
if xyVectorized
  vectVars = [1,2]; % input to odenumjac
else
  vectVars = [];
end

% Deal with a singular BVP.
singularBVP = false;
S = bvpget(options,'SingularTerm',[]);
if ~isempty(S)
  if (x(1) ~= 0) || (x(end) <= x(1))
    error('MATLAB:bvp6c:SingBVPInvalidInterval',...
          'Singular BVP must be on interval [0, b] with 0 < b.')
  end
  singularBVP = true;
  % Compute matrix for imposing necessary BC, Sy(0) = 0,
  % and impose on guess for solution.
  PBC = eye(size(S)) - pinv(S)*S;
  y(:,1) = PBC*y(:,1);
  % Compute matrix for proper definition of y'(0).
  PImS = pinv(eye(size(S)) - S);
  % Augment ExtraArgs with information about singular BVP.
  ExtraArgs = [ ExtraArgs {ode Fjac S PImS}];
  ode = @Sode;
  if ~isempty(Fjac)
    if npar > 0
      Fjac = @SFjacpar;
    else
      Fjac = @SFjac;
    end
  end
end

maxNewtIter = 4;
maxProbes = 4;    % weak line search
needGlobJac = true;

% Adjust the warnings.
warnstat(1) = warning('query','MATLAB:singularMatrix');
warnstat(2) = warning('query','MATLAB:nearlySingularMatrix');
warnoff = warnstat;
warnoff(1).state = 'off';
warnoff(2).state = 'off';

done = false;
% THE MAIN LOOP:
while ~done
  if unknownPar
    Y = [y(:);ExtraArgs{1}];
  else
    Y =  y(:);
  end

  [RHS,Xmid,Ymid,yp,Fmid,NF] = colloc_RHS(n,x,Y,ode,bc,npar,xyVectorized, ...
      mbcidx,nExtraArgs,ExtraArgs);
  nODEeval = nODEeval + NF;
  nBCeval = nBCeval + 1;

  for iter=1:maxNewtIter
    if needGlobJac
      % setup and factor the global Jacobian
      [dPHIdy,NF,NBC] = colloc_Jac(n,x,Xmid,Y,Ymid,yp,Fmid,ode,bc,Fjac,BCjac,npar,...
                             vectVars,averageJac,mbcidx,nExtraArgs,ExtraArgs);
      needGlobJac = false;
      nODEeval = nODEeval + NF;
      nBCeval = nBCeval + NBC;
      % explicit row scaling
      wt = max(abs(dPHIdy),[],2);
      if any(wt == 0) || ~all(isfinite(nonzeros(dPHIdy)))
        singJac = true;
      else
```

```
       scalMatrix = spdiags(1 ./ wt,0,nN+npar,nN+npar);
       dPHIdy = scalMatrix * dPHIdy;
       [L,U,P] = lu(dPHIdy);
       singJac = check_singular(dPHIdy,L,U,P,warnstat,warnoff);
     end
     if singJac
       msg = 'Unable to solve the collocation equations -- a singular Jacobian encountered';
       error('MATLAB:bvp6c:SingJac', msg);
     end
     scalMatrix = P * scalMatrix;
   end

   % find the Newton direction
   delY = U\(L\( scalMatrix * RHS ));
   distY = norm(delY);

   % weak line search with an affine stopping criterion
   lambda = 1;
   for probe = 1:maxProbes
     Ynew = Y - lambda*delY;

     if singularBVP     % Impose necessary BC, Sy(0) = 0.
       Ynew(1:n) = PBC*Ynew(1:n);
     end

     if unknownPar
       ExtraArgs{1} = Ynew(nN+1:end);
     end

     [RHS,Xmid,Ymid,yp,Fmid,NF] = colloc_RHS(n,x,Ynew,ode,bc,npar,xyVectorized,mbcidx,nExtraArgs,ExtraA

     nODEeval = nODEeval + NF;
     nBCeval = nBCeval + 1;

     distYnew = norm(U \ (L \ (scalMatrix * RHS)));

     if (distYnew < 0.9*distY)
       break
     else
       lambda = 0.5*lambda;
     end
   end

   needGlobJac = (distYnew > 0.1*distY);

   if distYnew < 0.1*rtol
     break
   else
     Y = Ynew;
   end
 end

y = reshape(Ynew(1:nN),n,N); % yp, ExtraArgs, and RHS are consistent with y

[res,NF,Yip05,Fmid(:,:,2)] = residual(ode,x,y,yp,Fmid,RHS,threshold,xyVectorized,nBCs, ...
                     mbcidx,ExtraArgs);
 nODEeval = nODEeval + NF;

 if max(res) < rtol
   done = true;
 else
   % redistribute the mesh
   [N,x,y,mbcidx] = new_profile(n,x,y,Yip05,yp,Fmid,res,mbcidx,rtol,Nmax,MaxNewPts);
   if N > Nmax
     warning('MATLAB:bvp6c:RelTolNotMet', ...
         [ 'Unable to meet the tolerance without using more than %d '...
           'mesh points. \n The last mesh of %d points and ' ...
           'the solution are available in the output argument. \n ', ...
           'The maximum residual is %g, while requested accuracy is %g.'], ...
           Nmax,length(x),max(res),rtol);
     sol.solver = 'bvp6c';
     sol.maxres = max(res);
     if ~isempty(solnpts)
```

```
          sol.note = 'max residual before interpolation';
          sol.x=solnpts;
          sol.y=ntrp6c(ode,sol.x,x,y,yp,Fmid);
      else
          sol.x = x;
          sol.y = y;
          if slopeout
              sol.yp = yp;
              sol.ypmid = Fmid;
          end
      end
      if unknownPar
        sol.parameters = ExtraArgs{1};
      end
      if ~isempty(knownPar)
        sol.knownpars = knownPar{1};
      end
      sol.fevals = nODEeval;
      sol.meshexceeded= 1;

      if printstats
          fprintf('The solution was obtained on a mesh of %g points.\n',N);
          fprintf('The maximum residual is %10.3e. \n',sol.maxres);
          fprintf('There were %g calls to the ODE function. \n',nODEeval);
          fprintf('There were %g calls to the BC function. \n',nBCeval);
      end

      return
    end

    nN = n*N;

    needGlobJac = true;

  end

end    % while

% Output
sol.solver = 'bvp6c';
if ~isempty(solnpts)
    sol.note = 'max residual before interpolation';
    sol.x=solnpts;
    sol.y=ntrp6c(ode,sol.x,x,y,yp,Fmid);
%     ,ExtraArgs{1},[knownPar{:}]);
else
    sol.x = x;
    sol.y = y;
    if slopeout
      sol.yp = yp;
      sol.ypmid = Fmid;
    end
end
if unknownPar
  sol.parameters = ExtraArgs{1};
end
if ~isempty(knownPar)
    sol.knownpars = [knownPar{:}];
end
sol.fevals = nODEeval;

% Stats
if printstats
  fprintf('The solution was obtained on a mesh of %g points.\n',N);
  fprintf('The maximum residual is %10.3e. \n',max(res));
  fprintf('There were %g calls to the ODE function. \n',nODEeval);
  fprintf('There were %g calls to the BC function. \n',nBCeval);
end

%--------------------------------------------------------------------------

function f = condaux(flag,X,L,U,P)
```

```
%CONDAUX  Auxiliary function for estimation of condition.

switch flag
case 'dim'
    f = max(size(L));
case 'real'
    f = 1;
case 'notransp'
    f = U \ (L \ (P * X));
case 'transp'
    f = P * (L' \ (U' \ X));
end


%-------------------------------------------------------------------------

function singular = check_singular(A,L,U,P,warnstat,warnoff)
%CHECK_SINGULAR  Check A (L*U*P) for 'singularity'; mute certain warnings.

[lastmsg,lastid] = lastwarn('');

warning(warnoff);

Ainv_norm = normest1(@condaux,[],[],L,U,P);

warning(warnstat);

[msg,msgid] = lastwarn;
if isempty(msg)
  lastwarn(lastmsg,lastid);
else
  singular = true;
  for i = 1:length(warnstat)
    if strcmp(msgid,warnstat(i).identifier)
      return;
    end
  end
end

singular = (Ainv_norm*norm(A,inf)*eps > 1e-5);

%--------------------------------------------------------------------------

function [Sx, Spx] = interp_Hermite(w,h,y,yp,yp_ip025,yp_ip05,yp_ip075)
%INTERP_HERMITE  use the 6th order Hermite Interpolant presented by Cash
    %and Wright to find y and y' at abscissas for Quadrature.
N=size(y,2);
diagscal = spdiags(h',0,N-1,N-1);

Sx=   A66(w)*y(:,2:N) + A66(1-w)*y(:,1:N-1)   + ...
    ( B66(w)*yp(:,2:N) - B66(1-w)*yp(:,1:N-1) + ...
      C66(w)*(yp_ip075-yp_ip025) + D66(w)*yp_ip05 )*diagscal;

if nargout >1
    diagscal = spdiags(1./h',0,N-1,N-1);
    Spx=( Ap66(w)*y(:,2:N)  - Ap66(1-w)*y(:,1:N-1) )*diagscal + ...
        ( Bp66(w)*yp(:,2:N) + Bp66(1-w)*yp(:,1:N-1) + ...
          Cp66(w)*(yp_ip075-yp_ip025) + Dp66(w)*yp_ip05 );
end

function coeff = A66(w)
coeff=w^2*(15-50*w+60*w^2-24*w^3);
function coeff = B66(w)
coeff=w^2/3*(w-1)*(12*w^2-14*w+5);
function coeff = C66(w)
coeff=-w^2*8/3*(1-w)^2;
function coeff = D66(w)
coeff=w^2*8*(1-w)^2*(2*w-1);

function coeff = Ap66(w)
coeff=w*(30-150*w+240*w^2-120*w^3);
function coeff = Bp66(w)
```

```
coeff=w*(w*(20*w^2+19)-(104*w^2+10)/3);
function coeff = Cp66(w)
coeff=-16/3*w*(1-3*w+2*w^2);
function coeff = Dp66(w)
coeff=w*(80*w^3-160*w^2+96*w-16);

function [res,nfcn,Y05,Yp05] = residual(Fcn, x, y, yp, Fmid, RHS, threshold, ...
                                xyVectorized, nBCs, mbcidx, ExtraArgs)
%RESIDUAL  Compute L2-norm of the residual using 7-point Lobatto quadrature.

% multi-point BVP support
ismbvp = ~isempty(mbcidx);
nregions = length(mbcidx) + 1;
Lidx = [1, mbcidx+1];
Ridx = [mbcidx, length(x)];
if ismbvp
  FcnArgs = {0,ExtraArgs{:}};  % pass region idx
else
  FcnArgs = ExtraArgs;
end

lob(2)= 0.0848880518607165;
lobw(2)=0.276826047361566;
lob(3)= 0.265575603264643;
lobw(3)=0.431745381209863;

lob(5)= 0.734424396735357;
lobw(5)=lobw(3);
lob(6)= 0.9151119481392835;
lobw(6)=lobw(2);

[n,N] = size(y);

Yp05=zeros(n,N-1);    %output more accurate yp_ip05
Y05=zeros(n,N-1);    %output more accurate yp_ip05

res = [];
nfcn = 0;

NewtRes = zeros(n,N-1);
% Do not populate the interface intervals for multi-point BVPs.
intidx = setdiff(1:N-1,mbcidx);
NewtRes(:,intidx) = reshape(RHS(nBCs+1:end),n,[]);

for region = 1:nregions
  if ismbvp
    FcnArgs{1} = region;    % Pass the region index to the ODE function.
  end

  xidx = Lidx(region):Ridx(region);    % mesh point index
  Nreg = length(xidx);
  xreg = x(xidx);
  yreg = y(:,xidx);
  ypreg = yp(:,xidx);
  hreg = diff(xreg);

  iidx = xidx(1:end-1);   % mesh interval index
  Nint = length(iidx);
  thresh = threshold(:,ones(1,Nint));

  yp_ip025=Fmid(:,iidx,1);
  yp_ip075=Fmid(:,iidx,3);

  diagscal = spdiags(hreg',0,Nint,Nint);
  xip05 = 0.5*(xreg(iidx) + xreg(iidx+1));
  %more accurate estimate of y_ip05 than used in Cash-Singhal
  Yip05 = 0.5*(yreg(:,iidx+1)+ yreg(:,iidx)) - ...
      (ypreg(:,iidx+1)-yp(:,iidx)+4*(yp_ip075-yp_ip025))*diagscal/24;
  if xyVectorized
    Yp_ip05 = feval(Fcn,xip05,Yip05,FcnArgs{:});
    nfcn = nfcn + 1;
  else % not vectorized
```

```
    Yp_ip05 = zeros(n,Nint);
    for i = 1:Nint
      Yp_ip05(:,i) = feval(Fcn,xip05(i),Yip05(:,i),FcnArgs{:});
    end
    nfcn = nfcn + Nint;
  end
  Y05(:,iidx)=Yip05;
  Yp05(:,iidx)=Yp_ip05;

  res_reg=zeros(1,Nint);

  %Sum contributions from other points
  for j=[2,3,5,6]
      xLob = xreg(1:Nreg-1) + lob(j)*hreg;
      [yLob,ypLob] = interp_Hermite(lob(j),hreg,yreg,ypreg,yp_ip025,Yp_ip05,yp_ip075);
      if xyVectorized
        fLob = feval(Fcn,xLob,yLob,FcnArgs{:});
        nfcn = nfcn + 1;
      else
        fLob = zeros(n,Nint);
        for i = 1:Nint
          fLob(:,i) = feval(Fcn,xLob(i),yLob(:,i),FcnArgs{:});
        end
        nfcn = nfcn + Nint;
      end
      temp = (ypLob - fLob) ./ max(abs(fLob),thresh);
      res_reg = res_reg + lobw(j)*dot(temp,temp,1);
  end

  % scaling
  res_reg = sqrt( abs(hreg/2) .* res_reg);

  res(iidx) = res_reg;
end
% %--------------------------------------------------------------------------

function [NN,xx,yy,mbcidxnew] = new_profile(n,x,y,y05,F,Fmid,res,mbcidx,rtol,Nmax,MaxNewPts)
%NEW_PROFILE  Redistribute mesh points and approximate the solution.

% multi-point BVP support
nregions = length(mbcidx) + 1;
Lidx = [1, mbcidx+1];
Ridx = [mbcidx, length(x)];

mbcidxnew = [];

xx = [];
yy = [];
NN = 0;

for region = 1:nregions

  xidx = Lidx(region):Ridx(region);  % mesh point index
  xreg = x(xidx);
  yreg = y(:,xidx);
  Freg = F(:,xidx);
  hreg = diff(xreg);
  Nreg = length(xidx);

  iidx = xidx(1:end-1);    % mesh interval index
  resreg = res(iidx);

  F025reg = Fmid(:,iidx,1);
  F05reg  = Fmid(:,iidx,2);
  F075reg = Fmid(:,iidx,3);

  i1 = find(resreg > rtol);
  i2 = find(resreg(i1) > 100*rtol);
  NNmax = Nreg + length(i1) + length(i2);
  xxreg = zeros(1,NNmax);
  yyreg = zeros(n,NNmax);
  last_int = Nreg - 1;
  xxreg(1) = xreg(1);
  yyreg(:,1) = yreg(:,1);
  NNreg = 1;

  i = 1;
  while i <= last_int
    if resreg(i) > rtol     % introduce points
      if resreg(i) > 100*rtol
        Ni = MaxNewPts;
        hi = hreg(i) / (Ni+1);
        j = 1:Ni;
        xxreg(NNreg+j) = xxreg(NNreg) + j*hi;
        for j=1:Ni
            yyreg(:,NNreg+j) = interp_Hermite(j/(Ni+1),hreg(i),yreg(:,i:i+1),...
                      Freg(:,i:i+1),F025reg(:,i),F05reg(:,i),F075reg(:,i));
        end
      else
        Ni = 1;
        xxreg(NNreg+1) = xxreg(NNreg) + hreg(i)/2;
        yyreg(:,NNreg+1) = y05(:,i);
      end
      NNreg = NNreg + Ni;
    else                 % try removing points
      if (i <= last_int-2) && (max(resreg(i+1:i+2)) < rtol)
        hnew = (hreg(i)+hreg(i+1)+hreg(i+2))/2;
        C1 = resreg(i)/(hreg(i)/hnew)^(11/2);
        C2 = resreg(i+1)/(hreg(i+1)/hnew)^(11/2);
        C3 = resreg(i+2)/(hreg(i+2)/hnew)^(11/2);
        pred_res = max([C1,C2,C3]);

        if pred_res < 0.5 * rtol   % replace 3 intervals with 2
          xxreg(NNreg+1) = xxreg(NNreg) + hnew;
          yyreg(:,NNreg+1) = interp_Hermite(0.5,hreg(i),yreg(:,(i+1):(i+2)),...
                      Freg(:,(i+1):(i+2)),F025reg(:,i+1),F05reg(:,i+1),F075reg(:,i+1));
          NNreg = NNreg + 1;
          i = i + 2;
        end
      end
    end
    NNreg = NNreg + 1;
    xxreg(NNreg) = xreg(i+1);   % preserve the next mesh point
    yyreg(:,NNreg) = yreg(:,i+1);
    i = i + 1;
  end

  NN = NN + NNreg;
  if (NN > Nmax)
    % return the previous solution
    xx = x;
    yy = y;
    mbcidxnew = mbcidx;
    break
  else
    xx = [xx, xxreg(1:NNreg)];
    yy = [yy, yyreg(:,1:NNreg)];
    if region < nregions    % possible only for multipoint BVPs
      mbcidxnew = [mbcidxnew, NN];
    end
  end
end

%--------------------------------------------------------------------------

function [Phi,Xmid,Ymid,F,Fmid,nfcn] = colloc_RHS(n, x, Y, Fcn, Gbc, npar, ...
                                        xyVectorized, mbcidx, nExtraArgs, ExtraArgs)
%COLLOC_RHS  Evaluate the system of collocation equations Phi(Y).
%   The derivative approximated at the midpoints and returned in Fmid is
%   used to estimate the residual.

% multi-point BVP support
ismbvp = ~isempty(mbcidx);
```

```
nregions = length(mbcidx) + 1;
Lidx = [1, mbcidx+1];
Ridx = [mbcidx, length(x)];
if ismbvp
  FcnArgs = {0,ExtraArgs{:}};    % Pass the region index to the ODE function.
else
  FcnArgs = ExtraArgs;
end

y = reshape(Y(1:end-npar),n,[]);

[n,N] = size(y);
nBCs = n*nregions + npar;

F = zeros(n,N);
Fmid = zeros(n,N-1,3);    % include interface intervals
Phi = zeros(nBCs+n*(N-nregions),1);    % exclude interface intervals

% Boundary conditions
% Do not pass info about singular BVPs in ExtraArgs to BC function.
Phi(1:nBCs) = feval(Gbc,y(:,Lidx),y(:,Ridx),ExtraArgs{1:nExtraArgs});
phiptr = nBCs;    % active region of Phi

for region = 1:nregions
  if ismbvp
    FcnArgs{1} = region;
  end
  xidx = Lidx(region):Ridx(region);   % mesh point index
  Nreg = length(xidx);
  xreg = x(xidx);
  yreg = y(:,xidx);

  iidx = xidx(1:end-1);   % mesh interval index
  Nint = length(iidx);

  % derivative at the mesh points
  if xyVectorized
    Freg = feval(Fcn,xreg,yreg,FcnArgs{:});
    nfcn = 1;
  else
    Freg = zeros(n,Nreg);
    for i = 1:Nreg
      Freg(:,i) = feval(Fcn,xreg(i),yreg(:,i),FcnArgs{:});
    end
    nfcn = Nreg;
  end

  %mesh point data
  h = diff(xreg);
  H = spdiags(h(:),0,Nint,Nint);
  xi = xreg(1:end-1);
  yi = yreg(:,1:end-1);
  xip1 = xreg(2:end);
  yip1 = yreg(:,2:end);
  Fi = Freg(:,1:end-1);
  Fip1 = Freg(:,2:end);

  %interior points & derivative
  xip025 = 0.25*(3*xi + xip1);
  xip075 = 0.25*(xi + 3*xip1);
  yip025 = (54*yi + 10*yip1 + (9*Fi - 3*Fip1)*H)/64;
  yip075 = (10*yi + 54*yip1 + (3*Fi - 9*Fip1)*H)/64;
  if xyVectorized
    Fip025 = feval(Fcn,xip025,yip025,FcnArgs{:});
    Fip075 = feval(Fcn,xip075,yip075,FcnArgs{:});
    nfcn = nfcn + 3;
  else % not vectorized
    Fip025 = zeros(n,Nint);
    Fip075 = zeros(n,Nint);
    for i = 1:Nint
      Fip025(:,i) = feval(Fcn,xip025(i),yip025(:,i),FcnArgs{:});
      Fip075(:,i) = feval(Fcn,xip075(i),yip075(:,i),FcnArgs{:});
    end
    nfcn = nfcn + 2*Nint;
  end

  %mid points & derivative
  xip05 = 0.5*(xi + xip1);
  yip05 = 0.5*(yi + yip1) - (5*Fi - 16*Fip025 + 16*Fip075 - 5*Fip1)*(H/24);
  if xyVectorized
    Fip05 = feval(Fcn,xip05,yip05,FcnArgs{:});
    nfcn = nfcn + 1;
  else % not vectorized
    Fip05 = zeros(n,Nint);
    for i = 1:Nint
      Fip05(:,i) = feval(Fcn,xip05(i),yip05(:,i),FcnArgs{:});
    end
    nfcn = nfcn + Nint;
  end

  % the Cash-Singhal formula
  Phireg = yip1 - yi - (7*Fi + 32*Fip025 + 12*Fip05 + 32*Fip075 + 7*Fip1)*(H/90);

  % output assembly
  Phi(phiptr+1:phiptr+n*Nint) = Phireg(:);
  phiptr = phiptr + n*Nint;

  Xmid(:,iidx,1) = xip025;
  Xmid(:,iidx,2) = xip05;
  Xmid(:,iidx,3) = xip075;

  Ymid(:,iidx,1) = yip025;
  Ymid(:,iidx,2) = yip05;
  Ymid(:,iidx,3) = yip075;

  F(:,xidx) = Freg;
  Fmid(:,iidx,1) = Fip025;
  Fmid(:,iidx,2) = Fip05;
  Fmid(:,iidx,3) = Fip075;

end


%---------------------------------------------------------------------------

function [dPHIdy,nfcn,nbc] = colloc_Jac(n, x, Xmid, Y, Ymid, F, Fmid, ode, bc, ...
                                        Fjac, BCjac, npar, vectVars, averageJac,...
                                        mbcidx, nExtraArgs, ExtraArgs)
%COLLOC_JAC  Form the global Jacobian of collocation eqns.

% multi-point BVP support
ismbvp = ~isempty(mbcidx);
nregions = length(mbcidx) + 1;
Lidx = [1, mbcidx+1];
Ridx = [mbcidx, length(x)];
if ismbvp
  FcnArgs = {0,ExtraArgs{:}};  % pass region idx
else
  FcnArgs = ExtraArgs;
end

N = length(x);
nN = n*N;
In = eye(n);

nfcn = 0;
nbc = 0;

y = reshape(Y(1:nN),n,N);

% BC points
ya = y(:,Lidx);
yb = y(:,Ridx);
```

```
if isempty(Fjac)  % use numerical approx
  threshval = 1e-6;
  Joptions.diffvar = 2;  % dF(x,y)/dy
  Joptions.vectvars = vectVars;
  Joptions.thresh = threshval(ones(n,1));
  if npar > 0   % unknown parameters
    if ismbvp
      dPoptions.diffvar = 4;  % dF(x,y,region,p)/dp
    else
      dPoptions.diffvar = 3;  % dF(x,y,p)/dp
    end
    dPoptions.vectvars = vectVars;
    dPoptions.thresh = threshval(ones(npar,1));
    dPoptions.fac = [];
  end
end

% Collocation equations
nBCs = n*nregions + npar;

rows = nBCs+1:nBCs+n;   % define the action area
cols = 1:n;             % in the global Jacobian

if npar == 0   % no unknown parameters -----------------------------

  dPHIdy = spalloc(nN,nN,2*n*nN);  % sparse storage

  if isempty(BCjac)   % use numerical approx
    [dGdya,dGdyb,nbc] = BCnumjac(bc,ya,yb,n,npar,nExtraArgs,ExtraArgs);
  elseif isa(BCjac,'cell')     % Constant partial derivatives {dGdya,dGdyb}
    dGdya = BCjac{1};
    dGdyb = BCjac{2};
  else  % use analytical Jacobian
    [dGdya,dGdyb] = feval(BCjac,ya,yb,ExtraArgs{1:nExtraArgs});
  end

  % Collocation equations
  for region = 1:nregions

    % Left BC
    dPHIdy(1:nBCs,cols) = dGdya(:,(region-1)*n+(1:n));

    if  ismbvp
      FcnArgs{1} = region; % Pass the region index to the ODE function.
    end

    xidx = Lidx(region):Ridx(region);    % mesh point index
    xreg = x(xidx);
    yreg = y(:,xidx);
    Freg = F(:,xidx);
    hreg = diff(xreg);

    iidx = xidx(1:end-1);    % mesh interval index
    Nint = length(iidx);

    [X1qtrreg, Xmidreg, X3qtrreg] = midptreg(iidx,Xmid);
    [Y1qtrreg, Ymidreg, Y3qtrreg] = midptreg(iidx,Ymid);
    [F1qtrreg, Fmidreg, F3qtrreg] = midptreg(iidx,Fmid);

    % Collocation equations
    if isempty(Fjac)  % use numerical approx

      Joptions.fac = [];
      [Ji,Joptions.fac,ignored,nFcalls] = ...
          odenumjac(ode,{xreg(1),yreg(:,1),FcnArgs{:}},Freg(:,1),Joptions);
      nfcn = nfcn+nFcalls;
      nrmJi = norm(Ji,1);

      for i = 1:Nint
        hi = hreg(i);
        % the right mesh point
        xip1 = xreg(i+1);
        yip1 = yreg(:,i+1);
        Fip1 = Freg(:,i+1);
        [Jip1,Joptions.fac,ignored,nFcalls] = ...
            odenumjac(ode,{xip1,yip1,FcnArgs{:}},Fip1,Joptions);
        nfcn = nfcn + nFcalls;
        nrmJip1 = norm(Jip1,1);

        %the interior points
        if averageJac && (norm(Jip1 - Ji,1) <= 0.125*(nrmJi + nrmJip1))
          Jip025 = 0.25*(3*Ji + Jip1);
          Jip05 = 0.5*(Ji + Jip1);
          Jip075 = 0.25*(Ji + 3*Jip1);
        else
          [xip025, xip05, xip075] = midpti(i,X1qtrreg, Xmidreg, X3qtrreg);
          [yip025, yip05, yip075] = midpti(i,Y1qtrreg, Ymidreg, Y3qtrreg);
          [Fip025, Fip05, Fip075] = midpti(i,F1qtrreg, Fmidreg, F3qtrreg);

          [Jip025,Joptions.fac,ignored,nFcalls025] = ...
              odenumjac(ode,{xip025,yip025,FcnArgs{:}},Fip025,Joptions);
          [Jip05,Joptions.fac,ignored,nFcalls05] = ...
              odenumjac(ode,{xip05,yip05,FcnArgs{:}},Fip05,Joptions);
          [Jip075,Joptions.fac,ignored,nFcalls075] = ...
              odenumjac(ode,{xip075,yip075,FcnArgs{:}},Fip075,Joptions);
          nfcn = nfcn + nFcalls025 + nFcalls05 + nFcalls075;
        end

        Jip05Jip025=Jip05*Jip025;
        Jip05Jip075=Jip05*Jip075;
        % assembly
        dPHIdy(rows,cols)=calc_dPHYdy1(hi,In,Ji,Jip025,Jip05,Jip075,Jip05Jip025,Jip05Jip075);
        cols = cols + n;
        dPHIdy(rows,cols)=calc_dPHYdy2(hi,In,Jip1,Jip025,Jip05,Jip075,Jip05Jip025,Jip05Jip075);
        rows = rows + n;   % next equation

        Ji = Jip1;
        nrmJi = nrmJip1;
      end

    elseif isa(Fjac,'numeric') % constant Jacobian
      J = Fjac(:,(region-1)*n+(1:n));
      J2 = J*J;
      J3 = J2*J;
      for i = 1:Nint
        hhreg(i) = hreg(i)*hreg(i);
        hhhreg(i) = hhreg(i)*hreg(i);
        h2J   = hreg(i)/2*J;
        h10J2 = hhhreg(i)/10*J2;
        h120J3 = hhhreg(i)/120*J3;
        dPHIdy(rows,cols) = - In - (h2J+h10J2+h120J3);
        cols = cols + n;
        dPHIdy(rows,cols) =   In - (h2J-h10J2+h120J3);
        rows = rows + n;   % next equation
      end

    else % use analytical Jacobian

      Ji = feval(Fjac,xreg(:,1),yreg(:,1),FcnArgs{:});

      for i = 1:Nint
        hi = hreg(i);
        % the right mesh point
        xip1 = xreg(i+1);
        yip1 = yreg(:,i+1);
        Jip1 = feval(Fjac,xip1,yip1,FcnArgs{:});

        %the interior points
        [xip025, xip05, xip075] = midpti(i,X1qtrreg, Xmidreg, X3qtrreg);
        [yip025, yip05, yip075] = midpti(i,Y1qtrreg, Ymidreg, Y3qtrreg);

        Jip025 = feval(Fjac,xip025,yip025,FcnArgs{:});
```

```
        Jip05 = feval(Fjac,xip05,yip05,FcnArgs{:});
        Jip075 = feval(Fjac,xip075,yip075,FcnArgs{:});

        Jip05Jip025=Jip05*Jip025;
        Jip05Jip075=Jip05*Jip075;
        % assembly
        dPHIdy(rows,cols)=calc_dPHYdy1(hi,In,Ji,Jip025,Jip05,Jip075,Jip05Jip025,Jip05Jip075);
        cols = cols + n;
        dPHIdy(rows,cols)=calc_dPHYdy2(hi,In,Jip1,Jip025,Jip05,Jip075,Jip05Jip025,Jip05Jip075);
        rows = rows + n;   % next equation

        Ji = Jip1;
      end
    end

    % Right BC
    dPHIdy(1:nBCs,cols) = dGdyb(:,(region-1)*n+(1:n));

    cols = cols + n;

  end % regions

else  % there are unknown parameters --------------

  dPHIdy = spalloc(nN+npar,nN+npar,(nN+npar)*(2*n+npar));  % sparse storage
  last_cols = zeros(nN+npar,npar);   % accumulator

  if isempty(BCjac)   % use numerical approx
    [dGdya,dGdyb,nbc,dGdpar] = BCnumjac(bc,ya,yb,n,npar,nExtraArgs,ExtraArgs);
  elseif isa(BCjac,'cell')     % Constant partial derivatives {dGdya,dGdyb}
    dGdya  = BCjac{1};
    dGdyb  = BCjac{2};
    dGdpar = BCjac{3};
  else  % use analytical Jacobian
    [dGdya,dGdyb,dGdpar] = feval(BCjac,ya,yb,ExtraArgs{1:nExtraArgs});
  end
  last_cols(1:nBCs,:) = dGdpar;

  % Collocation equations
  for region = 1:nregions

    % Left BC
    dPHIdy(1:nBCs,cols) = dGdya(:,(region-1)*n+(1:n));

    if ismbvp
      FcnArgs{1} = region;
    end

    xidx = Lidx(region):Ridx(region);
    xreg = x(xidx);
    yreg = y(:,xidx);
    Freg = F(:,xidx);
    hreg = diff(xreg);

    iidx = xidx(1:end-1);    % mesh interval index
    Nint = length(iidx);

    [X1qtrreg, Xmidreg, X3qtrreg] = midptreg(iidx,Xmid);
    [Y1qtrreg, Ymidreg, Y3qtrreg] = midptreg(iidx,Ymid);
    [F1qtrreg, Fmidreg, F3qtrreg] = midptreg(iidx,Fmid);

    % Collocation equations
    if isempty(Fjac)  % use numerical approx

      Joptions.fac = [];
      dPoptions.fac = [];
      [Ji,Joptions.fac,dFdpar_i,dPoptions.fac,nFcalls] = ...
          Fnumjac(ode,{xreg(1),yreg(:,1),FcnArgs{:}},Freg(:,1),...
                  Joptions,dPoptions);
      nfcn = nfcn+nFcalls;
      nrmJi = norm(Ji,1);
      nrmdFdpar_i = norm(dFdpar_i,1);
      for i = 1:Nint
        hi = hreg(i);
        % the right mesh point
        xip1 = xreg(i+1);
        yip1 = yreg(:,i+1);
        Fip1 = Freg(:,i+1);
        [Jip1,Joptions.fac,dFdpar_ip1,dPoptions.fac,nFcalls] = ...
            Fnumjac(ode,{xip1,yip1,FcnArgs{:}},Fip1,Joptions,dPoptions);
        nfcn = nfcn + nFcalls;
        nrmJip1 = norm(Jip1,1);
        nrmdFdpar_ip1 = norm(dFdpar_ip1,1);

        %the interior points
        if averageJac && (norm(Jip1 - Ji,1) <= 0.125*(nrmJi + nrmJip1)) && ...
              (norm(dFdpar_ip1 - dFdpar_i,1) <= 0.125*(nrmdFdpar_i + nrmdFdpar_ip1))
          Jip025 = 0.25*(3*Ji + Jip1);
          Jip05 = 0.5*(Ji + Jip1);
          Jip075 = 0.25*(Ji + 3*Jip1);

          dFdpar_ip025 = 0.25*(3*dFdpar_i + dFdpar_ip1);
          dFdpar_ip05 = 0.5*(dFdpar_i + dFdpar_ip1);
          dFdpar_ip075 = 0.25*(dFdpar_i + 3*dFdpar_ip1);
        else
          [xip025, xip05, xip075] = midpti(i,X1qtrreg, Xmidreg, X3qtrreg);
          [yip025, yip05, yip075] = midpti(i,Y1qtrreg, Ymidreg, Y3qtrreg);
          [Fip025, Fip05, Fip075] = midpti(i,F1qtrreg, Fmidreg, F3qtrreg);

          [Jip025,Joptions.fac,dFdpar_ip025,dPoptions.fac,nFcalls025] = ...
              Fnumjac(ode,{xip025,yip025,FcnArgs{:}},Fip025,Joptions,dPoptions);
          [Jip05,Joptions.fac,dFdpar_ip05,dPoptions.fac,nFcalls05] = ...
              Fnumjac(ode,{xip05,yip05,FcnArgs{:}},Fip05,Joptions,dPoptions);
          [Jip075,Joptions.fac,dFdpar_ip075,dPoptions.fac,nFcalls075] = ...
              Fnumjac(ode,{xip075,yip075,FcnArgs{:}},Fip075,Joptions,dPoptions);

          nfcn = nfcn + nFcalls025 + nFcalls05 + nFcalls075;
        end

        Jip05Jip025=Jip05*Jip025;
        Jip05Jip075=Jip05*Jip075;
        % assembly
        dPHIdy(rows,cols)=calc_dPHYdy1(hi,In,Ji,Jip025,Jip05,Jip075,Jip05Jip025,Jip05Jip075);
        cols = cols + n;
        dPHIdy(rows,cols)=calc_dPHYdy2(hi,In,Jip1,Jip025,Jip05,Jip075,Jip05Jip025,Jip05Jip075);

        last_cols(rows,:) = - ( ...
            hi/90*( 7*(dFdpar_i+dFdpar_ip1)+32*(dFdpar_ip025+dFdpar_ip075)+12*dFdpar_ip05 )     + ...
            hi*hi/180*( Jip025*(9*dFdpar_i-3*dFdpar_ip1) + Jip075*(3*dFdpar_i-9*dFdpar_ip1) + ...
                        Jip05*( 16*(dFdpar_ip025-dFdpar_ip075)+5*(dFdpar_ip1-dFdpar_i) )  )     + ...
            hi*hi*hi/240*( Jip05Jip025*(3*dFdpar_i-dFdpar_ip1)+Jip05Jip075*(3*dFdpar_ip1-dFdpar_i) )  ..
                               );
        rows = rows + n;   % next equation

        Ji = Jip1;
        nrmJi = nrmJip1;
        dFdpar_i = dFdpar_ip1;
        nrmdFdpar_i = nrmdFdpar_ip1;
      end

    elseif isa(Fjac,'cell') % Constant partial derivatives {dFdY,dFdp}

      J = Fjac{1}(:,(region-1)*n+(1:n));
      dFdp = Fjac{2}(:,(region-1)*npar+(1:npar));
      J2 = J*J;
      J3 = J2*J;
      for i = 1:Nint
        hhreg(i) = hreg(i)*hreg(i);
        hhhreg(i) = hhreg(i)*hreg(i);
        h2J   = hreg(i)/2*J;
        h10J2 = hhhreg(i)/10*J2;
        h120J3 = hhhreg(i)/120*J3;
        h3_60J2 = hhhreg(i)/60*J2;
```

```
        dPHIdy(rows,cols) = - In - (h2J+h10J2+h120J3);
        cols = cols + n;
        dPHIdy(rows,cols) =   In - (h2J-h10J2+h120J3);

        last_cols(rows,:) = - hreg(i)*dFdp - h3_60J2*DFdp;
        rows = rows+n;   % next equation
      end

    else % use analytical Jacobian

      [Ji,dFdpar_i] = feval(Fjac,xreg(1),yreg(:,1),FcnArgs{:});

      for i = 1:Nint
        hi = hreg(i);
        % the right mesh point
        xip1 = xreg(i+1);
        yip1 = yreg(:,i+1);
        [Jip1, dFdpar_ip1] = feval(Fjac,xip1,yip1,FcnArgs{:});

        %the interior points
        [xip025, xip05, xip075] = midpti(i,X1qtrreg, Xmidreg, X3qtrreg);
        [yip025, yip05, yip075] = midpti(i,Y1qtrreg, Ymidreg, Y3qtrreg);

        [Jip025, dFdpar_ip025] = feval(Fjac,xip025,yip025,FcnArgs{:});
        [Jip05, dFdpar_ip05] = feval(Fjac,xip05,yip05,FcnArgs{:});
        [Jip075, dFdpar_ip075] = feval(Fjac,xip075,yip075,FcnArgs{:});

        % assembly
        dPHIdy(rows,cols)=calc_dPHYdy1(hi,In,Ji,Jip025,Jip05,Jip075,Jip05Jip025,Jip05Jip075);
        cols = cols + n;
        dPHIdy(rows,cols)=calc_dPHYdy2(hi,In,Jip1,Jip025,Jip05,Jip075,Jip05Jip025,Jip05Jip075);

        last_cols(rows,:) = - ( ...
            hi/90*( 7*(dFdpar_i+dFdpar_ip1)+32*(dFdpar_ip025+dFdpar_ip075)+12*dFdpar_ip05 )     + ...
            hi*hi/180*( Jip025*(9*dFdpar_i-3*dFdpar_ip1) + Jip075*(3*dFdpar_i-9*dFdpar_ip1) + ...
                        Jip05*( 16*(dFdpar_ip025-dFdpar_ip075)+5*(dFdpar_ip1-dFdpar_i) )  )     + ...
            hi*hi*hi/240*( Jip05Jip025*(3*dFdpar_i-dFdpar_ip1)+Jip05Jip075*(3*dFdpar_ip1-dFdpar_i) )  ..
                           );
        rows = rows + n;   % next equation

        Ji = Jip1;
        dFdpar_i = dFdpar_ip1;
      end
    end

    % Right BC
    dPHIdy(1:nBCs,cols) = dGdyb(:,(region-1)*n+(1:n));

    cols = cols + n;
  end

  dPHIdy(:,end-npar+1:end) = last_cols;  % accumulated
end

%--------------------------------------------------------------------------

function dPHIdy=calc_dPHYdy1(hi,In,Ji,Jip025,Jip05,Jip075,Jip05Jip025,Jip05Jip075)

     dPHIdy   = - In -             ( ...
            hi/90*       ( 7*Ji+27*Jip025+6*Jip05+5*Jip075   ) + ...
            hi*hi/360*   ( 27*Jip05Jip025-5*Jip05Jip075)       + ...
            ( hi*hi/360*(18*Jip025-10*Jip05+6*Jip075) + ...
              hi*hi*hi/240*(3*Jip05Jip025-Jip05Jip075) )*Ji      ...
                                   );
%--------------------------------------------------------------------------

function dPHIdy=calc_dPHYdy2(hi,In,Jip1,Jip025,Jip05,Jip075,Jip05Jip025,Jip05Jip075)

      dPHIdy   = In -              (...
            hi/90*       ( 5*Jip025+6*Jip05+27*Jip075+7*Jip1  ) + ...
            hi*hi/360*   ( 5*Jip05Jip025-27*Jip05Jip075 )       - ...
            ( hi*hi/360*(6*Jip025-10*Jip05+18*Jip075) + ...
            hi*hi*hi/240*(Jip05Jip025-3*Jip05Jip075)  )*Jip1   ...
                                     );
%--------------------------------------------------------------------------

function res = bcaux(Ya,Yb,n,bcfun,varargin)
  ya = reshape(Ya,n,[]);
  yb = reshape(Yb,n,[]);
  res = feval(bcfun,ya,yb,varargin{:});

%---------------------------------------------------------------------------

function [dBCdya,dBCdyb,nCalls,dBCdpar] = BCnumjac(bc,ya,yb,n,npar, ...
                                                  nExtraArgs,ExtraArgs)
%BCNUMJAC  Numerically compute dBC/dya, dBC/dyb, and dBC/dpar, if needed.

% Do not pass info about singular BVPs in ExtraArgs to BC function.
bcArgs = {ya(:),yb(:),n,bc,ExtraArgs{1:nExtraArgs}};
dBCoptions.thresh = repmat(1e-6,length(ya(:)),1);
dBCoptions.fac = [];
dBCoptions.vectvars = []; % BC functions not vectorized

bcVal  = bcaux(bcArgs{:});
nCalls = 1;

dBCoptions.diffvar = 1;
[dBCdya,ignored,ignored1,nbc] = odenumjac(@bcaux,bcArgs,bcVal,dBCoptions);
nCalls = nCalls + nbc;
dBCoptions.diffvar = 2;
[dBCdyb,ignored,ignored1,nbc] = odenumjac(@bcaux,bcArgs,bcVal,dBCoptions);
nCalls = nCalls + nbc;
if npar > 0
  bcArgs = {ya,yb,ExtraArgs{1:nExtraArgs}};
  dBCoptions.thresh = repmat(1e-6,npar,1);
  dBCoptions.diffvar = 3;
  [dBCdpar,ignored,ignored1,nbc] = odenumjac(bc,bcArgs,bcVal,dBCoptions);
  nCalls = nCalls + nbc;
end

%---------------------------------------------------------------------------

function [dFdy,dFdy_fac,dFdp,dFdp_fac,nFcalls] = Fnumjac(ode,odeArgs,odeVal,...
                                                    Joptions,dPoptions)
%FNUMJAC  Numerically compute dF/dy and dF/dpar.

[dFdy,dFdy_fac,ignored,dFdy_nfcn] = odenumjac(ode,odeArgs,odeVal,Joptions);

[dFdp,dFdp_fac,ignored,dFdp_nfcn] = odenumjac(ode,odeArgs,odeVal,dPoptions);

nFcalls = dFdy_nfcn + dFdp_nfcn;

%---------------------------------------------------------------------------

function Fval = Sode(x,y,varargin)
%SODE    Evaluate the derivative function for singular BVPs.
%   Information for singular BVP starts at varargin{end-3}.
odefun = varargin{end-3};
Fval = feval(odefun,x,y,varargin{1:end-4});

% singular point, PImS = varargin{end};
idx = find(x == 0);
if ~isempty(idx)
  Fval(:,idx) = varargin{end}*Fval(:,idx);
end

% regular points, S = varargin{end-1}
idx = find(x ~= 0);
if ~isempty(idx)
  Fval(:,idx) = Fval(:,idx) + (varargin{end-1}*y(:,idx)) * ...
                              spdiags(1./x(idx)',0,length(idx),length(idx));
end
```

```
%---------------------------------------------------------------------------

function dFdy = SFjac(x,y,varargin)
%SFJAC   Numerically compute dF/dy for singular BVPs.
%   Information for singular BVP starts at varargin{end-3}.
Fjac = varargin{end-2};
if isa(Fjac,'numeric')
  n = size(Fjac,1);
  cols = 1:n;
  % Is this a multipoint BVP?
  nreg = size(Fjac,2)/n;   % rectangular Jacobian?
  if nreg > 1
    region = varargin{1};
    cols = n*(region-1)+1 : n*region;
  end
  dFdy = Fjac(:,cols);
else
  dFdy = feval(Fjac,x,y,varargin{1:end-4});
end
if x > 0
  % S = varargin{end-1}
  dFdy = dFdy + varargin{end-1}/x;
else
  % PImS = varargin{end};
  dFdy = varargin{end}*dFdy;
end

%---------------------------------------------------------------------------

function [dFdy, dFdpar] = SFjacpar(x,y,varargin)
%SFJACPAR  Numerically compute dF/dy and dF/dpar for singular BVPs.
%   Information for singular BVP starts at varargin{end-3}.
Fjac = varargin{end-2};
if isa(Fjac,'cell')
  n_y = size(Fjac{1},1);
  % Is this a multipoint BVP?
  nreg = size(Fjac{1},2)/n;   % rectangular Jacobian?
  n_p = size(Fjac{2},2)/nreg; % number of parameters
  cols_y = 1:n_y;
  cols_p = 1:n_p;
  % Is this a multipoint BVP?
  if nreg > 1
    region = varargin{1};
    cols_y = n_y*(region-1)+1 : n_y*region;
    cols_p = n_p*(region-1)+1 : n_p*region;
  end
  dFdy   = Fjac{1}(:,cols_y);
  dFdpar = Fjac{2}(:,cols_p);
else
  [dFdy, dFdpar] = feval(Fjac,x,y,varargin{1:end-4});
end
if x > 0
  % S = varargin{end-1}
  dFdy = dFdy + varargin{end-1}/x;
else
  % PImS = varargin{end};
  dFdy = varargin{end}*dFdy;
  dFdpar = varargin{end}*dFdpar;
end


%--------------------------------------------------------------------------

function [a025, a05, a075] = midptreg(iidx,mid)
a025 = mid(:,iidx,1);
a05 = mid(:,iidx,2);
a075 = mid(:,iidx,3);

function [a025, a05, a075] = midpti(i, b025, b05, b075)
a025 = b025(:,i);
a05 = b05(:,i);
```

```
a075 = b075(:,i);
```

D.2 bvpinit, bvpset, bvpget

```
function solinit = bvpinit(x,v,parameters,varargin)
%BVPINIT  Form the initial guess for BVP4C.
%   SOLINIT = BVPINIT(X,YINIT) forms the initial guess for BVP4C in common
%   circumstances. The boundary value problem (BVP) is to be solved on [a,b].
%   The vector X specifies a and b as X(1) = a and X(end) = b. It is also
%   a guess for an appropriate mesh. BVP4C will adapt this mesh to the solution,
%   so often a guess like X = linspace(a,b,10) will suffice, but in difficult
%   cases, mesh points should be placed where the solution changes rapidly.
%
%   The entries of X must be ordered. For two-point BVPs, the entries of X
%   must be distinct, so if a < b, then X(1) < X(2) < ... < X(end), and
%   similarly for a > b. For multipoint BVPs there are boundary conditions
%   at points in [a,b]. Generally, these points represent interfaces and
%   provide a natural division of [a,b] into regions. BVPINIT enumerates
%   the regions from left to right (from a to b), with indices starting
%   from 1. You can specify interfaces by double entries in the initial
%   mesh X. BVPINIT interprets oneentry as the right end point of region k
%   and the other as the left end point of region k+1. THREEBVP exemplifies
%   this for a three-point BVP.
%
%   YINIT provides a guess for the solution. It must be possible to evaluate
%   the differential equations and boundary conditions for this guess.
%   YINIT can be either a vector or a function:
%
%   vector:  YINIT(i) is a constant guess for the i-th component Y(i,:) of
%            the solution at all the mesh points in X.
%
%   function:  YINIT is a function of a scalar x. For example, use
%              solinit = bvpinit(x,@yfun) if for any x in [a,b], yfun(x)
%              returns a guess for the solution y(x). For multipoint BVPs,
%              BVPINIT calls Y = YINIT(X,K) to get an initial guess for the
%              solution at x in region k.
%
%   SOLINIT = BVPINIT(X,YINIT,PARAMETERS) indicates that the BVP involves
%   unknown parameters. A guess must be provided for all parameters in the
%   vector PARAMETERS.
%
%   SOLINIT = BVPINIT(X,YINIT,PARAMETERS,P1,P2...) passes the additional
%   known parameters P1,P2,... to the guess function as YINIT(X,P1,P2...) or
%   YINIT(X,K,P1,P2) for multipoint BVPs. Known parameters P1,P2,... can be
%   used only when YINIT is a function. When there are no unknown parameters,
%   use SOLINIT = BVPINIT(X,YINIT,[],P1,P2...).
%
%   SOLINIT = BVPINIT(SOL,[ANEW BNEW]) forms an initial guess on the interval
%   [ANEW,BNEW] from a solution SOL on an interval [a,b]. The new interval
%   must be bigger, so either ANEW <= a < b <= BNEW or ANEW >= a > b >= BNEW.
%   The solution SOL is extrapolated to the new interval. If present, the
%   PARAMETERS from SOL are used in SOLINIT. To supply a different guess for
%   unknown parameters use SOLINIT = BVPINIT(SOL,[ANEW BNEW],PARAMETERS).
%   Note, this has not been extended for bvp6c (i.e. the new computed
%   initial guess from a 6th of bvp6c solution will be only 4th order).
%
%   See also BVPGET, BVPSET, BVP4C, BVP6C, DEVAL, NTRP6C, @.

%   Jacek Kierzenka and Lawrence F. Shampine
%   Copyright 1984-2003 The MathWorks, Inc.
%   $Revision: 1.11.4.2 $  $Date: 2003/05/19 11:15:02 $
%   BVP6C Modification
%    Nick Hale  Imperial College London
%    $Date: 12/06/2006 $
% Extend existing solution?
if isstruct(x)
  if nargin < 2 || length(v) < 2
    error('MATLAB:bvpinit:NoSolInterval',...
          'Did not specify [ANEW BNEW] in BVPINIT(SOL,[ANEW BNEW]).')
  elseif nargin < 3
    parameters = [];
  end
  solinit = bvpxtrp(x,v,parameters);
  return;
end

% Create a guess structure.
N = length(x);
if x(1) == x(N)
  error('MATLAB:bvpinit:XSameEndPts',...
        'The entries of x must satisfy a = x(1) ~= x(end) = b.')
elseif x(1) < x(N)
  if any(diff(x) < 0)
    error('MATLAB:bvpinit:IncreasingXNotMonotonic',...
          'The entries of x must satisfy a = x(1) < x(2) < ... < x(end) = b.')
  end
else  % x(1) > x(N)
  if any(diff(x) > 0)
    error('MATLAB:bvpinit:DecreasingXNotMonotonic',...
          'The entries of x must satisfy a = x(1) > x(2) > ... > x(end) = b.')
  end
end

if nargin>2
  params = parameters;
else
  params = [];
end

extraArgs = varargin;

mbcidx = find(diff(x) == 0);  % locate internal interfaces
ismbvp = ~isempty(mbcidx);
if ismbvp
  Lidx = [1, mbcidx+1];
  Ridx = [mbcidx, length(x)];
end

if isnumeric(v)
  [m,n] = size(v);
  if n==1 && m==1        %must have at least two BPts and two functions
    error('MATLAB:bvpinit:SolGuessNotVector',...
          'The guess for the solution must return a vector.')
  elseif n==1 || m==1
    L=length(v);
    yinit=zeros(L,length(x));
    for i=1:L
        yinit(i,:)=v(i)*ones(1,length(x));
    end
  elseif m == length(x)
    yinit=v';
  elseif n == length(x)
    yinit=v;
  else
     error('MATLAB:bvpinit:SolGuessNotVector',...
          'The guess for the solution must return a vector.')
  end
else
  %checking
  if ismbvp
    w = feval(v,x(1),1,extraArgs{:});   % check region 1, only.
  else
    w = feval(v,x(1),extraArgs{:});
  end
  [m,n] = size(w);
  if m~=1 && n~=1
```

```
    error('MATLAB:bvpinit:SolGuessNotVector',...
          'The guess for the solution must return a vector.')
  end
  %assigning
  if ismbvp
    for region = 1:nregions
      for i = Lidx(region):Ridx(region)
        yinit(:,i) = feval(v,x(i),region,extraArgs{:});
      end
    end
  else
    yinit(:,1) = w(:);
    for i = 2:N
      yinit(:,i) = feval(v,x(i),extraArgs{:});
    end
  end
end

solinit.x = x(:)';  % row vector
solinit.y = yinit;
if ~isempty(params)
  solinit.parameters = params;
end

%---------------------------------------------------------------------------

function solxtrp = bvpxtrp(sol,v,parameters)
% Extend a solution SOL on [sol.x(1),sol.x(end)]
% to a guess for [v(1),v(end)] by extrapolation.

a = sol.x(1);
b = sol.x(end);

anew = v(1);
bnew = v(end);

if (a < b && (anew > a || bnew < b)) || ...
   (a > b && (anew < a || bnew > b))

  msg = sprintf(['The call BVPINIT(SOL,[ANEW BNEW]) must have\n',...
                 'ANEW <= SOL.x(1) < SOL.x(end) <= BNEW  or \n',...
                 'ANEW >= SOL.x(1) > SOL.x(end) >= BNEW. \n']);
  error('MATLAB:bvpinit:bvpxtrp:SolInterval', msg);

end

solxtrp.x = sol.x;
solxtrp.y = sol.y;
if abs(anew - a) <= 100*eps*max(abs(anew),abs(a))
    solxtrp.x(1) = anew;
else
    S = Hermite(sol.x(1),sol.y(:,1),sol.yp(:,1),...
                sol.x(2),sol.y(:,2),sol.yp(:,2),anew);
    solxtrp.x = [anew solxtrp.x];
    solxtrp.y = [S solxtrp.y];
end
if abs(bnew - b) <= 100*eps*max(abs(bnew),abs(b))
    solxtrp.x(end) = bnew;
else
    S = Hermite(sol.x(end-1),sol.y(:,end-1),sol.yp(:,end-1),...
                sol.x(end),sol.y(:,end),sol.yp(:,end),bnew);
    solxtrp.x = [solxtrp.x bnew];
    solxtrp.y = [solxtrp.y S];
end

if ~isempty(parameters)
  solxtrp.parameters = parameters;
elseif isfield(sol,'parameters')
  solxtrp.parameters = sol.parameters;
end

%---------------------------------------------------------------------------
```

```
function S = Hermite(x1,y1,yp1,x2,y2,yp2,xi)
% Evaluate cubic Hermite interpolant at xi.
h = x2 - x1;
s = (xi - x1)/h;
A1 = (s - 1)^2 * (1 + 2*s);
A2 = (3 - 2*s)*s^2;
B1 = h*s*(s - 1)^2;
B2 = h*s^2 *(s - 1);
S = A1*y1 + A2*y2 + B1*yp1 + B2*yp2;
```

```
function options = bvpset(varargin)
%BVPSET  Create/alter BVP OPTIONS structure.
%   OPTIONS = BVPSET('NAME1',VALUE1,'NAME2',VALUE2,...) creates an integrator
%   options structure OPTIONS in which the named properties have the
%   specified values. Any unspecified properties have default values. It is
%   sufficient to type only the leading characters that uniquely identify the
%   property. Case is ignored for property names.
%
%   OPTIONS = BVPSET(OLDOPTS,'NAME1',VALUE1,...) alters an existing options
%   structure OLDOPTS.
%
%   OPTIONS = BVPSET(OLDOPTS,NEWOPTS) combines an existing options structure
%   OLDOPTS with a new options structure NEWOPTS. Any new properties overwrite
%   corresponding old properties.
%
%   BVPSET with no input arguments displays all property names and their
%   possible values.
%
%BVPSET PROPERTIES
%
%RelTol - Relative tolerance for the residual [ positive scalar {1e-3} ]
%   This scalar applies to all components of the residual vector, and
%   defaults to 1e-3 (0.1% accuracy). The computed solution S(x) is the exact
%   solution of S'(x) = F(x,S(x)) + res(x). On each subinterval of the mesh,
%   component i of the residual satisfies
%          norm( res(i) / max( [abs(F(i)) , AbsTol(i)/RelTol] ) ) <= RelTol.
%
%AbsTol - Absolute tolerance for the residual [ positive scalar or vector {1e-6} ]
%   A scalar tolerance applies to all components of the residual vector.
%   Elements of a vector of tolerances apply to corresponding components of
%   the residual vector. AbsTol defaults to 1e-6. See RelTol.
%
%SingularTerm - Singular term of singular BVPs [ matrix ]
%   Set to the constant matrix S for equations of the form y' = S*y/x + f(x,y,p).
%
%FJacobian - Analytical partial derivatives of ODEFUN
%          [ function | matrix | cell array ]
%   For example, when solving y' = f(x,y), set this property to @FJAC if
%   DFDY = FJAC(X,Y) evaluates the Jacobian of f with respect to y.
%   If the problem involves unknown parameters, [DFDY,DFDP] = FJAC(X,Y,P)
%   must also return the partial derivative of f with respect to p.
%   For problems with constant partial derivatives, set this property to
%   the value of DFDY or to a cell array {DFDY,DFDP}.
%
%BCJacobian - Analytical partial derivatives of BCFUN
%           [ function | cell array ]
%   For example, for boundary conditions bc(ya,yb) = 0, set this property to
%   @BCJAC if [DBCDYA,DBCDYB] = BCJAC(YA,YB) evaluates the partial
%   derivatives of bc with respect to ya and to yb. If the problem involves
%   unknown parameters, [DBCDYA,DBCDYB,DBCDP] = BCJAC(YA,YB,P) must also
%   return the partial derivative of bc with respect to p.
%   For problems with constant partial derivatives, set this
%   property to a cell array {DBCDYA,DBCDYB} or {DBCDYA,DBCDYB,DBCDP}.
%
%Nmax - Maximum number of mesh points allowed [positive integer {floor(1000/n)}]
%
%Stats - Display computational cost statistics  [ on | {off} ]
%
%Vectorized - Vectorized ODE function  [ on | {off} ]
%   Set this property 'on' if the derivative function
%   ODEFUN([x1 x2 ...],[y1 y2 ...]) returns [ODEFUN(x1,y1) ODEFUN(x2,y2) ...].
%   When parameters are present, the derivative function
%   ODEFUN([x1 x2 ...],[y1 y2 ...],p) should return
%   [ODEFUN(x1,y1,p) ODEFUN(x2,y2,p) ...].
%
%MaxNewPts - Maximum number of new points introduced during mesh refinement.
%   [BVP6C ONLY]
%
%SlopeOut - Output derivative data for mesh and internal points. [ {on} | off ]
%   [BVP6C ONLY]
%   Storing these values will allow cheaper computation if necessary to
%   interpolate the bvp6c solution at given mesh points (and is necessary
%   if DEVAL is to be used). The values are output by default in SOL.YP and
%   SOL.YPMID but should be turned off to save on storage if not necessary.
%   See 'SolPts' for an alternative.
%
%XInt - Points within range of ODE system for which a solution is required.
%   When the solution is required at a set of feasible points, they
%   are entered here and the solution interpolated within bvp6c.
%   This alternative approach to using DEVAL saves on both computation and
%   storage since values required for interpolation are already known.
%
%   See also BVPGET, BVPINIT, BVP4C, BVP6C, DEVAL, NTRP6C.

%   Jacek Kierzenka and Lawrence F. Shampine
%   Copyright 1984-2003 The MathWorks, Inc.
%   $Revision: 1.10.4.2 $  $Date: 2003/10/21 11:55:36 $
%   BVP6C Modification
%    Nick Hale  Imperial College London
%    $Date: 12/06/2006 $

% Print out possible values of properties.
if (nargin == 0) && (nargout == 0)
  fprintf('          AbsTol: [ positive scalar or vector {1e-6} ]\n');
  fprintf('          RelTol: [ positive scalar {1e-3} ]\n');
  fprintf('    SingularTerm: [ matrix ]\n');
  fprintf('       FJacobian: [ function ]\n');
  fprintf('      BCJacobian: [ function ]\n');
  fprintf('           Stats: [ on | {off} ]\n');
  fprintf('            Nmax: [ nonnegative integer {floor(1000/n)} ]\n');
  fprintf('      Vectorized: [ on | {off} ]\n');
  fprintf('       MaxNewPts: [ nonnegative integer {2} ]\n');
  fprintf('        SlopeOut: [ {on} | off ]\n');
  fprintf('            XInt: [ vector ]\n');
  fprintf('\n');
  return;
end

Names = [
    'AbsTol      '
    'RelTol      '
    'SingularTerm'
    'FJacobian   '
    'BCJacobian  '
    'Stats       '
    'Nmax        '
    'Vectorized  '
    'MaxNewPts   '
    'SlopeOut    '
    'XInt        '
    ];

m = size(Names,1);
names = lower(Names);

% Combine all leading options structures o1, o2, ... in odeset(o1,o2,...).
options = [];
i = 1;
while i <= nargin
  arg = varargin{i};
  if ischar(arg)                         % arg is an option name
    break;
  end
  if ~isempty(arg)                      % [] is a valid options argument
    if ~isa(arg,'struct')
      error('MATLAB:bvpset:NoPropNameOrStruct',...
            ['Expected argument %d to be a string property name '...
                     'or an options structure\ncreated with BVPSET.'], i);
    end
    if isempty(options)
      options = arg;
    else
      for j = 1:m
        val = arg.(deblank(Names(j,:)));
        if ~isequal(val,[])             % empty strings '' do overwrite
          options.(deblank(Names(j,:))) = val;
        end
      end
    end
  end
  i = i + 1;
end
if isempty(options)
  for j = 1:m
    options.(deblank(Names(j,:))) = [];
  end
end

% A finite state machine to parse name-value pairs.
if rem(nargin-i+1,2) ~= 0
  error('MATLAB:bvpset:ArgNameValueMismatch',...
        'Arguments must occur in name-value pairs.');
end
expectval = 0;                      % start expecting a name, not a value
while i <= nargin
  arg = varargin{i};
  if ~expectval
    if ~ischar(arg)
      error('MATLAB:bvpset:InvalidPropName',...
        'Expected argument %d to be a string property name.', i);
    end
    lowArg = lower(arg);
    j = strmatch(lowArg,names);
    if isempty(j)                       % if no matches
      error('MATLAB:bvpset:InvalidPropName',...
            'Unrecognized property name ''%s''.', arg);
    elseif length(j) > 1                % if more than one match
      % Check for any exact matches (in case any names are subsets of others)
      k = strmatch(lowArg,names,'exact');
      if length(k) == 1
        j = k;
      else
        msg = sprintf('Ambiguous property name ''%s'' ', arg);
        msg = [msg '(' deblank(Names(j(1),:))];
        for k = j(2:length(j))'
          msg = [msg ', ' deblank(Names(k,:))];
        end
        msg = sprintf('%s).', msg);
        error('MATLAB:bvpset:AmbiguousPropName', msg);
      end
    end
    expectval = true;                      % we expect a value next
  else
    options.(deblank(Names(j,:))) = arg;
    expectval = false;
  end
  i = i + 1;
end

if expectval
  error('MATLAB:bvpset:NoValueForProp',...
        'Expected value for property ''%s''.', arg);
end
```

```
function o = bvpget(options,name,default)
%BVPGET  Get BVP OPTIONS parameters.
%   VAL = BVPGET(OPTIONS,'NAME') extracts the value of the named property
%   from integrator options structure OPTIONS, returning an empty matrix if
%   the property value is not specified in OPTIONS. It is sufficient to type
%   only the leading characters that uniquely identify the property. Case is
%   ignored for property names. [] is a valid OPTIONS argument.
%
%   VAL = BVPGET(OPTIONS,'NAME',DEFAULT) extracts the named property as
%   above, but returns VAL = DEFAULT if the named property is not specified
%   in OPTIONS. For example
%
%       val = bvpget(opts,'RelTol',1e-4);
%
%   returns val = 1e-4 if the RelTol property is not specified in opts.
%
%   See also BVPSET, BVPINIT, BVP4C, DEVAL.

%   Jacek Kierzenka and Lawrence F. Shampine
%   Copyright 1984-2003 The MathWorks, Inc.
%   $Revision: 1.11.4.2 $  $Date: 2003/10/21 11:55:35 $
%   BVP6C Modification
%    Nick Hale  Imperial College London
%    $Date: 12/06/2006 $

if nargin < 2
  error('MATLAB:bvpget:NotEnoughInputs', 'Not enough input arguments.');
end
if nargin < 3
  default = [];
end

if ~isempty(options) && ~isa(options,'struct')
  error('MATLAB:bvpget:OptsNotStruct',...
        'First argument must be an options structure created with BVPSET.');
end

if isempty(options)
  o = default;
  return;
end

Names = [
    'AbsTol      '
    'RelTol      '
    'SingularTerm'
    'FJacobian   '
    'BCJacobian  '
    'Stats       '
    'Nmax        '
    'Vectorized  '
    'MaxNewPts   '
    'SlopeOut    '
    'XInt        '
    ];

names = lower(Names);

lowName = lower(name);
j = strmatch(lowName,names);
if isempty(j)               % if no matches
  error('MATLAB:bvpget:InvalidPropName',...
        ['Unrecognized property name ''%s''.  ' ...
         'See BVPSET for possibilities.'], name);
elseif length(j) > 1            % if more than one match
  % Check for any exact matches (in case any names are subsets of others)
  k = strmatch(lowName,names,'exact');
  if length(k) == 1
    j = k;
  else
    msg = sprintf('Ambiguous property name ''%s'' ', name);
    msg = [msg '(' deblank(Names(j(1),:))];
    for k = j(2:length(j))'
      msg = [msg ', ' deblank(Names(k,:))];
    end
    msg = sprintf('%s).', msg);
    error('MATLAB:bvpget:AmbiguousPropName', msg);
  end
end

if any(strmatch(fieldnames(options),deblank(Names(j,:))))
  o = options.(deblank(Names(j,:)));
  if isempty(o)
    o = default;
  end
else
  o = default;
end
```

D.3 deval, ntrp6c

```
function [Sxint,Spxint] = deval(sol,xint,idx)
%DEVAL  Evaluate the solution of a differential equation problem.
%   SXINT = DEVAL(SOL,XINT) evaluates the solution of a differential equation
%   problem at all the entries of the vector XINT. SOL is a structure returned
%   by an initial value problem solver (ODE45, ODE23, ODE113, ODE15S, ODE23S,
%   ODE23T, ODE23TB, ODE15I), the boundary value problem solver (BVP4C,BVP6C), or
%   the solver for delay differential equations (DDE23). The elements of XINT
%   must be in the interval [SOL.x(1) SOL.x(end)]. For each I, SXINT(:,I) is
%   the solution corresponding to XINT(I).
%
%   SXINT = DEVAL(SOL,XINT,IDX) evaluates as above but returns only
%   the solution components with indices listed in IDX.
%
%   SXINT = DEVAL(XINT,SOL) and SXINT = DEVAL(XINT,SOL,IDX) are also acceptable.
%
%   [SXINT,SPXINT] = DEVAL(...) evaluates as above but returns also the value
%   of the first derivative of the polynomial interpolating the solution.
%
%   For multipoint boundary value problems, the solution obtained with BVP4C
%   might be discontinuous at the interfaces. For an interface point XC, DEVAL
%   returns the average of the limits from the left and right of XC. To get
%   the limit values, set the XINT argument of DEVAL to be slightly smaller or
%   slightly larger than XC.
%
%   Class support for inputs SOL and XINT:
%     float: double, single
%
%   See also
%       ODE solvers:  ODE45, ODE23, ODE113, ODE15S,
%                     ODE23S, ODE23T, ODE23TB, ODE15I
%       DDE solver:   DDE23
%       BVP solver:   BVP4C, BVP6C

%   Jacek Kierzenka and Lawrence F. Shampine
%   Copyright 1984-2004 The MathWorks, Inc.
%   BVP6C Modification
%    Nick Hale  Imperial College London
%    $Date: 12/06/2006 $

if ~isa(sol,'struct')
  % Try  DEVAL(XINT,SOL)
  temp = sol;
  sol  = xint;
  xint = temp;
end
```

```
try
  t = sol.x;
  y = sol.y;
catch
  error('MATLAB:deval:SolNotFromDiffEqSolver',...
        '%s must be a structure returned by a differential equation solver.',...
        inputname(1));
end

if nargin < 3
  idx = 1:size(y,1);  % return all solution components
else
  if any(idx < 0) || any(idx > size(y,1))
    error('MATLAB:deval:IDXInvalidSolComp',...
          'Incorrect solution component requested in %s.',inputname(3));
  end
end
idx = idx(:);

if isfield(sol,'solver')
  solver = sol.solver;
else
  msg = sprintf('Missing ''solver'' field in the structure %s.',inputname(1));
  if isfield(sol,'yp')
    warning('MATLAB:deval:MissingSolverField',['%s\n         DEVAL will ' ...
            'treat %s as an output of the MATLAB R12 version of BVP4C.'], ...
            msg, inputname(1));
    solver = 'bvp4c';
  else
    error('MATLAB:deval:NoSolverInStruct', msg);
  end
end

if ~ismember(solver,{'ode113','ode15i','ode15s','ode23','ode23s','ode23t',...
                     'ode23tb','ode45','bvp4c','dde23','bvp6c'})
  error('MATLAB:deval:InvalidSolver',...
        'Unrecognized solver name ''%s'' in %s.solver.',solver,inputname(1));
end

% If necessary, convert sol.idata to MATLAB R14 format.
if ~isfield(sol,'extdata') && ismember(solver,{'ode113','ode15s','ode23','ode45'})
  sol.idata = convert_idata(solver,sol.idata);
end

% Determine the dominant data type
dataType = superiorfloat(sol.x,xint);

Spxint_requested = (nargout > 1);   % Evaluate the first derivative?

n = length(idx);
Nxint = length(xint);
Sxint = zeros(n,Nxint,dataType);
if Spxint_requested
  Spxint = zeros(n,Nxint,dataType);
end

% Make tint a row vector and if necessary,
% sort it to match the order of t.
tint = xint(:).';
tdir = sign(t(end) - t(1));
had2sort = any(tdir*diff(tint) < 0);
if had2sort
  [tint,tint_order] = sort(tdir*tint);
  tint = tdir*tint;
end

% Using the sorted version of tint, test for illegal values.
if (tdir*(tint(1) - t(1)) < 0) || (tdir*(tint(end) - t(end)) > 0)
  error('MATLAB:deval:SolOutsideInterval',...
        ['Attempting to evaluate the solution outside the interval\n'...
         '[%e, %e] where it is defined.\n'],t(1),t(end));
end
```

```
% Select appropriate interpolating function.
switch solver
 case 'ode113'
  interpfcn = @ntrp113;
 case 'ode15i'
  interpfcn = @ntrp15i;
 case 'ode15s'
  interpfcn = @ntrp15s;
 case 'ode23'
  interpfcn = @ntrp23;
 case 'ode23s'
  interpfcn = @ntrp23s;
 case 'ode23t'
  interpfcn = @ntrp23t;
 case 'ode23tb'
  interpfcn = @ntrp23tb;
 case 'ode45'
  interpfcn = @ntrp45;
 case {'bvp4c','dde23'}
  interpfcn = @ntrp3h;
 case {'bvp6c'}
  if ~isfield(sol,'yp') || ~isfield(sol,'ypmid')
        error('MATLAB:deval:nsufficientData',...
        'bvp6c deval requires the option BVPSET(''SLOPEOUT'',''ON'')');
  end
  interpfcn = @ntrp6h;
end

evaluated = 0;
bottom = 1;
while evaluated < Nxint

  % Find right-open subinterval [t(bottom), t(bottom+1))
  % containing the next entry of tint.
  Index = find( tdir*(tint(evaluated+1) - t(bottom:end)) >= 0 );
  bottom = bottom - 1 + Index(end);

  % Is it [t(end), t(end)]?
  at_tend = (t(bottom) == t(end));

  if at_tend
    % Use (t(bottom-1) t(bottom)] to interpolate y(t(end)) and yp(t(end)).
    index = find(tint(evaluated+1:end) == t(bottom));
    bottom = bottom - 1;
  else
    % Interpolate inside [t(bottom) t(bottom+1)).
    index = find( tdir*(tint(evaluated+1:end) - t(bottom+1)) < 0 );
  end

  switch solver
   case 'ode113'
    interpdata = { sol.idata.klastvec(bottom+1) ...
                   sol.idata.phi3d(:,:,bottom+1) ...
                   sol.idata.psi2d(:,bottom+1) };

   case 'ode15i'
    k = sol.idata.kvec(bottom+1);
    interpdata = { sol.x(bottom:-1:bottom-k+1) ...
                   sol.y(:,bottom:-1:bottom-k+1) };

   case 'ode15s'
    interpdata = { t(bottom+1)-t(bottom) ...
                   sol.idata.dif3d(:,:,bottom+1) ...
                   sol.idata.kvec(bottom+1) };

   case {'ode23','ode45'}
    interpdata = { t(bottom+1)-t(bottom)...
                   sol.idata.f3d(:,:,bottom+1) };

   case 'ode23s'
    interpdata = { t(bottom+1)-t(bottom) ...
                   sol.idata.k1(:,bottom+1) ...
```

```
                  sol.idata.k2(:,bottom+1) };

   case 'ode23t'
    interpdata = { t(bottom+1)-t(bottom) ...
                   sol.idata.z(:,bottom+1) ...
                   sol.idata.znew(:,bottom+1) };

   case 'ode23tb'
    interpdata = { sol.idata.t2(bottom+1) ...
                   sol.idata.y2(:,bottom+1) };

   case {'bvp4c','dde23'}
    interpdata = { sol.yp(:,bottom) ...
                   sol.yp(:,bottom+1) };
   case {'bvp6c'}
    interpdata = { sol.yp(:,bottom) sol.yp(:,bottom+1) ...
        sol.ypmid(:,bottom,:)};

  end

  % Evaluate the interpolant at all points from [t(bottom), t(bottom+1)).
  if Spxint_requested
    [yint,ypint] = feval(interpfcn,tint(evaluated+index),t(bottom),y(:,bottom),...
                         t(bottom+1),y(:,bottom+1),interpdata{:});
  else
    yint = feval(interpfcn,tint(evaluated+index),t(bottom),y(:,bottom),...
                 t(bottom+1),y(:,bottom+1),interpdata{:});
  end

  if at_tend
    bottom = bottom + 1;
  end

  % Purify the solution at t(bottom).
  index1 = find(tint(evaluated+index) == t(bottom));
  if ~isempty(index1)
    yint(idx,index1) = repmat(y(idx,bottom),1,length(index1));
  end

  % Accumulate the output.
  Sxint(:,evaluated+index) = yint(idx,:);
  if Spxint_requested
    Spxint(:,evaluated+index) = ypint(idx,:);
  end
  evaluated = evaluated + length(index);
end

% For multipoint BVPs, check if solution requested at interface points
multipointBVP = (isequal(solver,'bvp4c') || isequal(solver,'bvp6c')) && any(diff(t)==0);
if multipointBVP
  idxDiscontPoints = find(diff(t)==0);
  for i = idxDiscontPoints
    % Check whether solution requested at the interface
    idxIntPoints = find(tint == t(i));
    n = length(idxIntPoints);
    if n > 0
      % Average the solution across the interface
      nl = '\n         ';
      warning('MATLAB:deval:NonuniqueSolution',...
              ['At the interface XC = %g the solution might not be continuous.'...
               nl 'DEVAL returns the average of the limits from the left and'...
               nl 'from the right of the interface. To approximate the limit'...
               nl 'values, call DEVAL for XC-EPS(XC) or XC+EPS(XC).\n'],t(i));

      Sxint(:,idxIntPoints) = repmat((sol.y(idx,i)+sol.y(idx,i+1))/2,1,n);
      if Spxint_requested
        Spxint(:,idxIntPoints) = repmat((sol.yp(idx,i)+sol.yp(idx,i+1))/2,1,n);
      end
    end
  end
end
```

```
if had2sort     % Restore the order of tint in the output.
  Sxint(:,tint_order) = Sxint;
  if Spxint_requested
    Spxint(:,tint_order) = Spxint;
  end
end

% --------------------------------------------------------------------------

function idataOut = convert_idata(solver,idataIn)
% Covert an old sol.idata to the MATLAB R14 format

idataOut = idataIn;
switch solver
 case 'ode113'
  idataOut.phi3d = shiftdim(idataIn.phi3d,1);
 case 'ode15s'
  idataOut.dif3d = shiftdim(idataIn.dif3d,1);
 case {'ode23','ode45'}
  idataOut.f3d = shiftdim(idataIn.f3d,1);
end
```

```
function [Yint FEVALS]=ntrp6c(f,Xint,x,y,yp,Fmid,varargin)
%NTRP6C  New interpolation helper function for BVP6C.
%   YINT = NTRP6C(F,XINT,SOL) interpolates the bvp6c solution SOL
%   of ode system F to give solution values at new points XINT.
%   Note, it is not necessary to pass F if the slope values SOL.YP
%   and SOL.YPMID are stored in the structure SOL.
%
%   The function may be called without using bvp6c SOL structure using
%   YINT = NTRP6C(F,XINT,X,Y) where X and Y are some obtained solution.
%   Note that if the ODE system contains parameters if F=F(X,Y,P1,P2...)
%   then user should call YINT = NTRP6C(F,XINT,X,Y,[],[],P1,P2,..).
%
%   [YINT, FEVALS] = NTRP6C(F,...) will return also the number of
%   times function F was evaluated in computing new solutions YINT.
%
%   Note, this function is designed specifically for use via DEVAL.
%
%   See also BVP6C, DEVAL.

%   Nick Hale

expars=[];
if nargin>6  expars=varargin; end
if nargin<5  yp=[]; Fmid=[];   end
if nargin<4
    if isstruct(x)
        y=x.y;
        if isfield(x,'yp')    yp=x.yp;      else yp=[];   end
        if isfield(x,'ypmid') Fmid=x.ypmid; else Fmid=[]; end
        if isfield(x,'parameters') expars=[x.parameters]; end
        if isfield(x,'knownpars')  expars=[expars x.knownpars]; end;
        x=x.x;
    else
        error('MATLAB:ntrp6c:NotEnoughInputs',...
            'Insufficient input arguements\n');
    end
    expars=num2cell(expars);
end
if (isempty(yp)|isempty(Fmid))&~isa(f, 'function_handle')
    error('MATLAB:ntrp6c:FNotFunctionHandle',...
        'F must be a function if slope data is not specified.');
end

global FEVALS
FEVALS =0;

XL=x(1);
XR=x(end);
N=length(Xint);

if all(XL<=Xint) + all(Xint<=XR)<2    %check that XL<X<XR
    error('MATLAB:ntrp6c:XIntOutsideRange',...
        'X outside interval range [%f,%f]\n',XL,XR);
    return
end

% if all(sort(X)~=X) error('Please order the X[i]'); end

strt=1;
if Xint(1)==XL                   %check if Xint[1]==XL
    I(1)=0; w(1)=0; strt=2;
end
for i=strt:N
    I(i)=0;
    if Xint(i)==XL                   %error if Xint[i]==XL for i!=1
        error('MATLAB:ntrp6c:XLConflict',...
            'Xint[i]=XL for i!=1. Please order the Xint[i]');
        return;
    end
    while Xint(i)-x(I(i)+1)>0      %find interval containing each X[i]
        I(i)=I(i)+1;
    end
    w(i)=(Xint(i)-x(I(i)))./(x(I(i)+1)-x(I(i)));
end

J=1;K=1;
if I(1)==0                      %if X[1]=XL
    Yint(:,1)=y(:,1);
    J=2;K=2;
end
while J<=N
    while I(K)-I(J)==0          %find X's in same interval
        K=K+1;
        if K==N+1 break; end
    end
    if w(K-1)==1                %if X[J]=x[j] then use exact y(:,j)
        Yint(:,(K-1))=y(:,I(J)+1);
        if J<=K-2
            Yint(:,J:(K-2))=interp(f,x,y,yp,Fmid,expars,w(J:(K-2))',I(J));
        end
    else
        Yint(:,J:(K-1))=interp(f,x,y,yp,Fmid,expars,w(J:(K-1))',I(J));
    end
    J=K;
end

%----------------------------Interpolate---------------------------------------
function Y=interp(f,x,y,yp,Fmid,expars,w,int)
global FEVALS
XL=x(int);  XR=x(int+1);
H=XR-XL;
YL=y(:,int);YR=y(:,int+1);
if isempty(yp)                  %evaluate function f at mesh points
    FL=f(XL,YL,expars{:});
    FR=f(XR,YR,expars{:});
    FEVALS=FEVALS+2;
else                            %mesh slopes are known
    FL=yp(:,int);
    FR=yp(:,int+1);
end
if isempty(Fmid)                %evaluate function f at internal points
    y025 = (54*YL + 10*YR + (9*FL - 3*FR)*H)/64;
    y075 = (10*YL + 54*YR + (3*FL - 9*FR)*H)/64;
    F025 = f(0.25*(3*XL + XR),y025,expars{:});
    F075 = f(0.25*(XL + 3*XR),y075,expars{:});
    y05 = 0.5*(YL + YR) - H*(FR-FL+4*(F075-F025))/24;
    F05 = f(0.5*(XL + XR),y05,expars{:});
    FEVALS=FEVALS+3;
else                            %interior slopes are known
    F025 = Fmid(:,int,1);
    F05 =  Fmid(:,int,2);
    F075 = Fmid(:,int,3);
end

% interpolate at w
for i=1:length(w)
    wi=w(i);
    Y(:,i)=A66(wi)*YR + A66(1-wi)*YL+      ...
       (XR-XL)*( B66(wi)*FR - B66(1-wi)*FL + ...
      C66(wi)*(F075-F025) + D66(wi)*F05 );
end

%----------------------------Interpolation functions----------------------------------------
function ans = A66(w)
ans=w.^2.*(15-50*w+60*w.^2-24*w.^3);
function ans = B66(w)
ans=w.^2/3.*(w-1).*(12*w.^2-14*w+5);
function ans = C66(w)
ans=-w.^2*8/3.*(1-w).^2;
function ans = D66(w)
ans=w.^2.*8*(1-w).^2.*(2*w-1);
```

D.4 blackbox32

```
function Y=blackbox32(P,X)
%BLACKBOX32
%  Interplate very accurate solutions from MIRK12 for CW32

%   Nick Hale
%       Imperial College London
%       June 2006

if nargin==0 P=19; end
X=rand(10,1)

coeffs;     %compute constant coefficients

if P*(33-P)<0
    error('Problem outside permited range: 0-32\n');
end
[XL,XR]=xlist(P);
E=epslist(P);
N=length(X);

% exist analytic solution?
switch P
    case {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,20}
        Y=soln(X,E,P);
        return;
end;

p=0;
info=csvread('h:\my_data.txt',0,0,[0,0,0,2]);
while info(1)~=P
    p=p+info(2)+1;
    info=csvread('h:\my_data.txt',p,0,[p,0,p,2]);
end
y=csvread('h:\my_data.txt',p+1,0,[p+1,0,p+info(2),info(3)-1]); %mirk 12 solns
x=XL:(XR-XL)/(info(3)-1):XR;

if all(XL<=X) + all(X<=XR)<2    %check that XL<X<XR
    error('X outside interval range [%f,%f]\n',XL,XR);
end

strt=1;
if X(1)==XL                     %check if X[1]==XL
    I(1)=0; w(1)=0; strt=2;
end
for i=strt:N
    I(i)=0;
    if X(i)==XL                  %error if X[i]==XL for i!=1
        error('X[i]=XL for i!=1. Please order the X[i]');
    end
    while X(i)-x(I(i)+1)>0      %find interval containing each X[i]
        I(i)=I(i)+1;
    end
    w(i)=(X(i)-x(I(i)))./(x(I(i)+1)-x(I(i)));
end

J=1;K=1;
if I(J)==0                      %if X[J]=XL
    Y(:,J)=y(:,1);
    J=2;K=2;
end
while J<=N
    while I(K)-I(J)==0          %find X's in same interval
        K=K+1;
        if K==N+1 break; end
    end

    if w(K-1)==1                %if X[J]=x[j] then use exact y(:,j)
        Y(:,(K-1))=y(:,I(J)+1);
        if J<=K-2
            Y(:,J:(K-2))=interp(x,y,E,P,w(J:(K-2))',I(J));
        end
    else
        Y(:,J:(K-1))=interp(x,y,E,P,w(J:(K-1))',I(J));
    end
    J=K;
end

% %-------------------------------------------------------- %delete this
X;
Y;


%------------------------------------------------------------

function Y=interp(x,y,E,P,w,int)
XL=x(int);
XR=x(int+1);
YL=y(:,int);
YR=y(:,int+1);
FL=f(XL,YL,E,P);
FR=f(XR,YR,E,P);

%find interpolation points
[ypa7p,yma7p,ypa8p,yma8p,ypa9p,yma9p]=interppts(XL,XR,YL,YR,FL,FR,E,P);
%interpolate at w
for i=1:length(w)
    wi=w(i);
    Awi=A(wi);
    Y(:,i)=Awi*YR+(1-Awi)*YL+     ...
       (XR-XL)*( B(wi)*FR-B(1-wi).*FL          ...
                +C(wi)*ypa7p-C(1-wi)*yma7p    ...
                +D(wi)*ypa8p-D(1-wi)*yma8p    ...
                +EE(wi)*ypa9p-EE(1-wi)*yma9p);
end



function [ypa7p,yma7p,ypa8p,yma8p,ypa9p,yma9p]=interppts(XL,XR,YL,YR,FL,FR,E,P)
%--------------------------------COEFFS-----------------------------------
global r3 r5 r11 r15 r33 r1311 r1653 r2185 r4054 r22857 r24795 r5294425981
global a1 a2 a3 a4 a5 a6 a7 a8 a9
global a1m a1p a2m a2p
global b1p b1m b2p b2m b3
global c1p c1m c2p c2m c3p c3m
global d1p d1m d2p d2m d3p d3m
global e1 e2 e3
global f1p f1m f2p f2m f3 f4p f4m f5
global g1p g1m g2p g2m g3 g4p g4m g5
global rr1 rr2 rr3 rr4
global s1p s1m s2p s2m s3p s3m s4p s4m
global t1p t1m t2p t2m t3p t3m t4p t4m
global u1p u1m u2p u2m u3 u4 u5p u5m u6m u6p
%------------------------------------------------------------------------
h=XR-XL;
XLh=XL+0.5*h;

ypa1p=f(XLh+h*a1,a1p*YL+a1m*YR-h*(a2p*FL+a2m*FR),E,P);
yma1p=f(XLh-h*a1,a1m*YL+a1p*YR+h*(a2m*FL+a2p*FR),E,P);

ypa2p=f(XLh+h*a2,b1p*YL+b1m*YR-h*(b2p*FL+b2m*FR+b3*(yma1p-ypa1p)),E,P);
yma2p=f(XLh-h*a2,b1m*YL+b1p*YR+h*(b2m*FL+b2p*FR-b3*(yma1p-ypa1p)),E,P);

ypa3p=f(XLh+h*a3,c1p*YL+c1m*YR-h*(c2p*FL+c2m*FR+c3p*yma2p+c3m*ypa2p),E,P);
yma3p=f(XLh-h*a3,c1m*YL+c1p*YR+h*(c2m*FL+c2p*FR+c3m*yma2p+c3p*ypa2p),E,P);

ypa4p=f(XLh+h*a4,d1p*YL+d1m*YR-h*(d2p*FL+d2m*FR+d3p*yma3p+d3m*ypa3p),E,P);
yma4p=f(XLh-h*a4,d1m*YL+d1p*YR+h*(d2m*FL+d2p*FR+d3m*yma3p+d3p*ypa3p),E,P);

yp05p=f(XLh,0.5*(YL+YR)+h*(e1*(FL-FR)+e2*(yma3p-ypa3p)+e3*(yma4p-ypa4p)),E,P);

ypa5p=f(XLh+h*a5,f1p*YL+f1m*YR-h*(f2p*FL+f2m*FR+f3*(ypa3p-yma3p)+f4p*yma4p+f4m*ypa4p+f5*yp05p),E,P);
```

```
yma5p=f(XLh-h*a5,f1m*YL+f1p*YR+h*(f2m*FL+f2p*FR-f3*(ypa3p-yma3p)+f4m*yma4p+f4p*ypa4p+f5*yp05p),E,P);

ypa6p=f(XLh+h*a6,g1p*YL+g1m*YR-h*(g2p*FL+g2m*FR+g3*(ypa3p-yma3p)+g4p*yma4p+g4m*ypa4p+g5*yp05p),E,P);
yma6p=f(XLh-h*a6,g1m*YL+g1p*YR+h*(g2m*FL+g2p*FR-g3*(ypa3p-yma3p)+g4m*yma4p+g4p*ypa4p+g5*yp05p),E,P);

yp05p=f(XLh,0.5*(YL+YR)+h*(rr1*(FL-FR)+rr2*(yma4p-ypa4p)+rr3*(yma5p-ypa5p)+rr4*(yma6p-ypa6p)),E,P);

ypa7p=f(XLh+h*a7,s1p*YL+s1m*YR-h*(s2p*FL+s2m*FR+s3p*yma5p+s3m*ypa5p+s4p*yma6p+s4m*ypa6p),E,P);
yma7p=f(XLh-h*a7,s1m*YL+s1p*YR+h*(s2m*FL+s2p*FR+s3m*yma5p+s3p*ypa5p+s4m*yma6p+s4p*ypa6p),E,P);

ypa8p=f(XLh+h*a8,t1p*YL+t1m*YR-h*(t2p*FL+t2m*FR+t3p*yma5p+t3m*ypa5p+t4p*yma6p+t4m*ypa6p),E,P);
yma8p=f(XLh-h*a8,t1m*YL+t1p*YR+h*(t2m*FL+t2p*FR+t3m*yma5p+t3p*ypa5p+t4m*yma6p+t4p*ypa6p),E,P);

ypa9p=f(XLh+h*a9,u1p*YL+u1m*YR-h*(u2p*FL+u2m*FR+u3*(ypa5p-yma5p)+u4*(ypa6p-yma6p)+ ...
    u5p*yma7p+u5m*ypa7p+u6p*yma8p+u6m*ypa8p),E,P);
yma9p=f(XLh-h*a9,u1m*YL+u1p*YR+h*(u2m*FL+u2p*FR-u3*(ypa5p-yma5p)-u4*(ypa6p-yma6p)+ ...
    u5m*yma7p+u5p*ypa7p+u6m*yma8p+u6p*ypa8p),E,P);

%----------------------------Interpolation functions---------------------------------------

function coeffs = A(w)
    coeffs = -w.^2.*(-210+2590*w-14070*w.^2+41664*w.^3-71610*w.^4+71280*w.^5-38115*w.^6+8470*w.^7);
function coeffs = B(w)
    coeffs = 1/24*w.^2.*(w-1).*(8833*w.^6-30371*w.^5+42097*w.^4-30008*w.^3+11620*w.^2-2354*w+207);
function coeffs = C(w)
    global r3 r11 r33 a8
    coeffs = 1/95568*( 1445466*w.^5+(91476*a8*r11-176418*a8*r33-3613665)*w.^4  ...
                  +(-182952*a8*r11+352836*a8*r33+8712*r3+3441240)*w.^3    ...
                  +(-13068*r3-265650*a8*r33-1548195+157212*a8*r11)*w.^2   ...
                  +(89232*a8*r33+334158-65736*a8*r11+8316*r3)*w            ...
                  -29502-1980*r3+11880*a8*r11-12672*a8*r33).*(w-1).^2.*w.^2*(148+r3);
function coeffs = D(w)
    global r3 a8
    coeffs = 1/95568*w.^2.*(w-1).^2.*(r3-148).* ...
                (-1445466*w.^5 ...
                 +(3613665+1221858*a8-431244*a8*r3)*w.^4 ...
                 +(-2443716*a8+862488*a8*r3+8712*r3-3441240)*w.^3 ...
                 +(-13068*r3-590964*a8*r3+1548195+1762002*a8)*w.^2 ...
                 +(159720*a8*r3-334158+8316*r3-540144*a8)*w ...
                 +29502-15048*a8*r3-1980*r3+66528*a8);
function coeffs = EE(w)
    global r33
    coeffs = 121/48*w.^2.*(w-1).^2.*(-242*w.^5+(33*r33+605)*w.^4-(66*r33+528)*w.^3 ...
                                 +(49*r33+187)*w.^2-(16*r33+22)*w+2*r33);

%-----------------------------------------------------------------------------------------

function dydx = f(x,y,E,n)
[ode,ignored,ignored2] = feval(@problemlist,n);
dydx = feval(ode,x,y,E);

function ssoln = soln(x,E,n)
[ignored,ignored2,sol] = feval(@problemlist,n);
ssoln = feval(sol,x,E);

%-----------------------calculate constant coeffs-----------------------------------------
function coeffs
global r3 r5 r11 r15 r33 r1311 r1653 r2185 r4054 r22857 r24795 r5294425981
r3=sqrt(3.0);
r5=sqrt(5.0);
r11=sqrt(11.0);
r15=r3*r5;
r33=r11*r3;
r1311=sqrt(1311.0);
r1653=sqrt(1653.0);
r2185=sqrt(2185.0);
r4054=sqrt(4054.0);
r22857=sqrt(22857.0);
r24795=sqrt(24795.0);
r5294425981=sqrt(5294425981.0);

global a1 a2 a3 a4 a5 a6 a7 a8 a9
```

```
a1=1/4.0;
a2=r5294425981/212914.0;
a3=r2185/114.0;
a4=r1653/114.0;
a5=sqrt(5445/(15+2*r15))/66.0;
a6=sqrt(495+66*r15)/66.0;
a7=sqrt(297-132*r3)/66.0;
a8=sqrt(297+132*r3)/66.0;
a9=r33/22.0;

global a1m a1p a2m a2p
a1m = 27/32.0;
a1p = 5/32.0;
a2m = 9/64.0;
a2p = -3/64.0;

global b1p b1m b2p b2m b3
b1p = 1/2.0-134819/22666185698.0*r5294425981;
b1m = 1/2.0+134819/22666185698.0*r5294425981;
b2p = 14181/22666185698.0*r5294425981-973398021/22666185698.0;
b2m = 14181/22666185698.0*r5294425981+973398021/22666185698.0;
b3  = -536268696/11333092849.0;

global c1p c1m c2p c2m c3p c3m
c1p = 1/2.0-154085/13678632.0*r2185;
c1m = 1/2.0+154085/13678632.0*r2185;
c2p = 71731079/122511587616.0*r2185-618856/21824559.0;
c2m = 71731079/122511587616.0*r2185+618856/21824559.0;
c3p = -274547/1085400792747.0*r5294425981+1538286841/2327720164704.0*r2185;
c3m = 274547/1085400792747.0*r5294425981+1538286841/2327720164704.0*r2185;

global d1p d1m d2p d2m d3p d3m
d1p = 1/2.0-8053/656298.0*r1653;
d1m = 1/2.0+8053/656298.0*r1653;
d2p = 1001/2625192.0*r1653-7/456.0;
d2m = 1001/2625192.0*r1653+7/456.0;
d3p = -21/17480.0*r2185+21/15352.0*r1653;
d3m = +21/17480.0*r2185+21/15352.0*r1653;

global e1 e2 e3
e1 = 827/12544.0;
e2 = -1539/288512.0*r2185;
e3 = 57/6272.0*r1653;

global f1p f1m f2p f2m f3 f4p f4m f5
f1p = (15972-5*a5*(13302+301*r15))/31944.0;
f1m = (15972+5*a5*(13302+301*r15))/31944.0;
f2p = -(3191/149072+2225/1565256*r15)+(40085/447216+1979/894432*r15)*a5;
f2m = 3191/149072+2225/1565256*r15+(40085/447216+1979/894432*r15)*a5;
f3  = 171/24000592*r1311*(7*r15-124);
f4p = -(437/308792*r1653+6745/15130808*r24795)+(1543275/4323088+267501/8646176*r15)*a5;
f4m = 437/308792*r1653+6745/15130808*r24795+(1543275/4323088+267501/8646176*r15)*a5;
f5  = 2*a5*(994-101*r15)/10527;

global g1p g1m g2p g2m g3 g4p g4m g5
g1p = 1/2-11085*a6/5324+1505/31944*r15*a6;
g1m = (15972+5*a6*(13302-301*r15))/31944.0;
g2p = -3191/149072+2225/1565256*r15+(40085/447216-1979/894432*r15)*a6;
g2m = 3191/149072-2225/1565256*r15+(40085/447216-1979/894432*r15)*a6;
g3  = 171/24000592*r1311*(7*r15+124);
g4p = -437/308792*r1653+6745/15130808*r24795+(1543275/4323088-267501/8646176*r15)*a6;
g4m = 437/308792*r1653-6745/15130808*r24795+(1543275/4323088-267501/8646176*r15)*a6;
g5  = 2*a6*(994+101*r15)/10527;

global rr1 rr2 rr3 rr4
rr1 = -121/7168;
rr2 = -1083/207872*r1653;
rr3 = 3*a5*(7*r15+124)/512;
rr4 = -3*a6*(7*r15-124)/512;

global s1p s1m s2p s2m s3p s3m s4p s4m
```

```
s1p = 1/43454004.0*(3627+874*r3)*(7254-1748*r3-29927*a7);
s1m = 1/43454004.0*(3627+874*r3)*(7254-1748*r3+29927*a7);
s2p = 1/618552.0*(171+34*r3)*(213*a7+12*r3-52);
s2m = 1/618552.0*(171+34*r3)*(213*a7-12*r3+52);
s3p = 1/19021200.0*(3340+9*r15+362*r5+1400*r3)*(1965*a7-56*r15*a5-1360*a5 ...
        -32*r5*a5+240*r3*a5);
s3m = 1/19021200.0*(3340+9*r15+362*r5+1400*r3)*(1965*a7+56*r15*a5+1360*a5 ...
        +32*r5*a5-240*r3*a5);
s4p = 1/19021200.0*(3340+1400*r3-362*r5-9*r15)*(1965*a7-1360*a6+240*r3*a6 ...
        +32*r5*a6+56*r15*a6);
s4m = 1/19021200.0*(3340+1400*r3-362*r5-9*r15)*(1965*a7+1360*a6-240*r3*a6 ...
        -32*r5*a6-56*r15*a6);

global t1p t1m t2p t2m t3p t3m t4p t4m
t1p = 1/43454004.0*(874*r3-3627)*(-7254+29927*a8-1748*r3);
t1m = -1/43454004.0*(874*r3-3627)*(29927*a8+7254+1748*r3);
t2p = -1/618552.0*(-171+34*r3)*(-52-12*r3+213*a8);
t2m = -1/618552.0*(-171+34*r3)*(52+12*r3+213*a8);
t3p = 1/19021200.0*(-1400*r3+9*r5*r3+3340-362*r5)*(1965*a8+32*r5*a5 ...
        -240*r3*a5-1360*a5-56*a5*r15);
t3m = 1/19021200.0*(-1400*r3+9*r15+3340-362*r5)*(1965*a8+1360*a5 ...
        -32*r5*a5+240*r3*a5+56*a5*r15);
t4p = -1/19021200.0*(-3340+9*r5*r3+1400*r3-362*r5)*(-1360*a6 ...
        +1965*a8-240*r3*a6+56*r15*a6-32*r5*a6);
t4m = -1/19021200.0*(-3340+9*r5*r3+1400*r3-362*r5)*(240*r3*a6 ...
        +32*r5*a6+1360*a6+1965*a8-56*r15*a6);

global u1p u1m u2p u2m u3 u4 u5p u5m u6m u6p
u1m = -51/2662.0*r11*r3+1/2.0;
u1p = 51/2662.0*r11*r3+1/2.0;
u2m = 83/6655.0-1/1331.0*r11*r3;
u2p = -83/6655.0-1/1331.0*r11*r3;
u3 = 4/33275.0*(1416+37*r15)*a5;
u4 = 4/33275.0*r11*(122*r15+207)*a5;
u5m = 1/945010.0*r11*(-171+34*r3)*(120*r3+165+142*a8);
u5p = 1/945010.0*r11*(-171+34*r3)*(120*r3+165-142*a8);
u6m = 1/10395110.0*(-378+377*r3)*(15*r11*r3-420*r11+1562*a8);
u6p = 1/10395110.0*(-378+377*r3)*(-1562*a8+15*r33-420*r11);
```

D.5 examples

```
function measles
%MEASLES
%   This example is from U. Ascher, R. Mattheij, and R. Russell,
%   Numerical Solution of Boundary Value Problems for Ordinary Differential
%   Equations, SIAM, Philadelphia, PA, 1988 which models the spread of
%   measles in a population.
%
%   The example is used here to compare performance of bvp4c and bvp6c
%
%   See also BVP4C, BVPSET, BVPGET, BVPINIT, DEVAL, @.

%   Jacek Kierzenka and Lawrence F. Shampine
%   Copyright 1984-2002 The MathWorks, Inc.
%   $Revision: 1.10 $  $Date: 2002/04/15 03:35:16 $
%   BVP6C Modification
%    Nick Hale  Imperial College london
%    $Date: 12/06/2006 $
tol=1e-3;
fprintf('MEASLES\n abstol = %1.1g\n reltol = %1.1g \n\n',tol,tol)

options = bvpset('Stats','on','reltol',tol,'abstol',tol);
solinit = bvpinit(linspace(0,1,15),[0.01, 0.01, 0.01]);

fprintf('  BVP4c \n')
tic;
sol4 = bvp4c(@odes,@bcs,solinit,options);
toc
```

```
fprintf('\n  BVP6c \n')
tic;
sol6 = bvp6c(@odes,@bcs,solinit,options);
toc

figure(1)
plot(sol4.x,sol4.y(1,:)-0.07,'-',sol4.x,sol4.y(2,:),'--', ...
    sol4.x,sol4.y(3,:),'-.');
title(['bvp4c solutions. tol = ', num2str(tol)]);
legend('y_1(t) - 0.07 ','y_2(t)','y_3(t)',4);

figure(2)
plot(sol6.x,sol6.y(1,:)-0.07,'-',sol6.x,sol6.y(2,:),'--', ...
    sol6.x,sol6.y(3,:),'-.');
title(['bvp6c solutions. tol = ', num2str(tol)]);
legend('y_1(t) - 0.07 ','y_2(t)','y_3(t)',4);

% Difference in Solutions
x = linspace(0,1);
y4 = deval(sol4,x);
y6 = deval(sol6,x);

err=y4-y6;
for i=1:3
    Linf(i)=norm(err(i,:),inf);
end

fprintf('L_inf Difference in solutions of interpolation\n');
fprintf('\ty[1] = %g\n\ty[2] = %g\n\ty[3] = %g\n',Linf(1),Linf(2),Linf(3));


% --------------------------------------------------------------------------
function dydx = odes(x,y)
beta = 1575*(1+cos(2*pi*x));
dydx = [    0.02 - beta*y(1)*y(3)
        beta*y(1)*y(3) - y(2)/0.0279
        y(2)/0.0279 - y(3)/0.01     ];



% --------------------------------------------------------------------------

function res = bcs(ya,yb)
res = ya - yb;
```

```
function injection
%INJECTION
%   This example is also from U. Ascher, R. Mattheij, and R. Russell,
%   Numerical Solution of Boundary Value Problems for Ordinary Differential
%   Equations, SIAM, Philadelphia, PA, 198. here it models the injection of
%   a fluid along a long vertical channel.
%
%   The example is used here to compare performance of bvp4c and bvp6c
%
%   See also BVP4C, BVPSET, BVPGET, BVPINIT, DEVAL, @.

%   Jacek Kierzenka and Lawrence F. Shampine
%   Copyright 1984-2002 The MathWorks, Inc.
%   $Revision: 1.10 $  $Date: 2002/04/15 03:35:16 $
%   BVP6C Modification
%    Nick Hale  Imperial College london
%    $Date: 12/06/2006 $
tol=1e-3;
fprintf('INJECTION\n abstol = %1.1g\n reltol = %1.1g \n\n',tol,tol)

options = bvpset('stats','on','abstol',tol,'reltol',tol);
solinit = bvpinit(linspace(0,1,10),ones(7,1),1);

fprintf('  BVP4c \n')
tic;
sol4 = bvp4c(@odes,@bcs,solinit,options);
toc

fprintf('\n  BVP6c \n')
tic;
sol6 = bvp6c(@odes,@bcs,solinit,options);
toc

figure(1)
plot(sol4.x,sol4.y(1,:),'-*',sol4.x,50.0*sol4.y(4,:),'--*', ...
    sol4.x,sol4.y(6,:),'-.*');
title(['bvp4c solutions. tol = ', num2str(tol)]);
legend('f(t)','50*h(t)','O(t)',4);
axis([0 1 0 1]);

figure(2)
plot(sol6.x,sol6.y(1,:),'-*',sol6.x,50.0*sol6.y(4,:),'--*', ...
    sol6.x,sol6.y(6,:),'-.*');
title(['bvp6c solutions. tol = ', num2str(tol)]);
legend('f(t)','50.*h(t)','O(t)',4);
axis([0 1 0 1]);

% Difference in Solutions
x = linspace(0,1);
y4 = deval(sol4,x);
y6 = deval(sol6,x);

err=y4-y6;
for i=[1,4,6]
    Linf(i)=norm(err(i,:),inf);
end

fprintf('L_inf Difference in solutions of interpolation\n');
fprintf('\ty[1] = %g\n\ty[2] = %g\n\ty[3] = %g\n',Linf(1),Linf(4),Linf(6));

% --------------------------------------------------------------------------

function dydx = odes(x,y,A)
dydx = [y(2);
        y(3);
        100*(y(2)^2-y(1)*y(3)-A);
        y(5);
        -100*y(1)*y(5)-1;
        y(7);
        -70*y(1)*y(7)];
```

```
% ------------------------------------------------------------------------

function res = bcs(ya,yb,A)
res = [ ya(1);
        ya(2);
        yb(1)-1;
        yb(2);
        ya(4);
        yb(4);
        ya(6);
        yb(6)-1 ];
```

```
function shockbvp6
%SHOCKBVP  The solution has a shock layer near x = 0
%   This is an example used in U. Ascher, R. Mattheij, and R. Russell,
%   Numerical Solution of Boundary Value Problems for Ordinary Differential
%   Equations, SIAM, Philadelphia, PA, 1995,  to illustrate the mesh
%   selection strategy of COLSYS.
%
%   For 0 < e << 1, the solution of
%
%       e*y'' + x*y' = -e*pi^2*cos(pi*x) - pi*x*sin(pi*x)
%
%   on the interval [-1,1] with boundary conditions y(-1) = -2 and y(1) = 0
%   has a rapid transition layer at x = 0.
%
%   This example illustrates how a numerically difficult problem (e = 1e-4)
%   can be solved successfully using continuation. For this problem,
%   analytical partial derivatives are easy to derive and the solver benefits
%   from using them.
%
%   See also BVP4C, BVPSET, BVPGET, BVPINIT, DEVAL, @.

%   Jacek Kierzenka and Lawrence F. Shampine
%   Copyright 1984-2002 The MathWorks, Inc.
%   $Revision: 1.10 $  $Date: 2002/04/15 03:35:16 $
%   BVP6C Modification
%    Nick Hale  Imperial College London
%    $Date: 12/06/2006 $

% The differential equations written as a first order system and the
% boundary conditions are coded in shockODE and shockBC, respectively. Their
% partial derivatives are coded in shockJac and shockBCJac and passed to the
% solver via the options. The option 'Vectorized' instructs the solver that
% the differential equation function has been vectorized, i.e.
% shockODE([x1 x2 ...],[y1 y2 ...],e) returns [shockODE(x1,y1,e) shockODE(x2,y2,e) ...].
% Such coding improves the solver performance.

options = bvpset('FJacobian',@shockJac,'BCJacobian',@shockBCJac,'Vectorized','on');
options = bvpset(options,'reltol',1e-6,'abstol',1e-6);

% A guess for the initial mesh and the solution
sol = bvpinit([-1 -0.5 0 0.5 1],[1 0]);

% Solution for e = 1e-2, 1e-3, 1e-4 obtained using continuation.
e = 0.1;
tic
for i=2:5
  e = e/10;
  % e is the problem parameter. bvp4c passes its value to all the functions
  % shockODE, shockBC, shockJac and shockBCJac.
  sol = bvp6c(@shockODE,@shockBC,sol,options,e);
end
toc


% A guess for the initial mesh and the solution
sol = bvpinit([-1 -0.5 0 0.5 1],[1 0]);

% Solution for e = 1e-2, 1e-3, 1e-4 obtained using continuation.
e = 0.1;
tic
for i=2:5
  e = e/10;
  % e is the problem parameter. bvp4c passes its value to all the functions
  % shockODE, shockBC, shockJac and shockBCJac.
  sol = bvp6d(@shockODE,@shockBC,sol,options,e);
end
toc

% The final solution
figure;
plot(sol.x,sol.y(1,:));
```

```
axis([-1 1 -2.2 2.2]);
title(['There is a shock at x = 0 when \epsilon =' sprintf('%.e',e) '.']);
xlabel('x');
ylabel('solution y');

% --------------------------------------------------------------------------

function dydx = shockODE(x,y,e)
%SHOCKODE  Evaluate the ODE function (vectorized)
pix = pi*x;
dydx = [                 y(2,:)
         -x/e.*y(2,:) - pi^2*cos(pix) - pix/e.*sin(pix) ];

% --------------------------------------------------------------------------

function res = shockBC(ya,yb,e)
%SHOCKBC  Evaluate the residual in the boundary conditions
res = [ ya(1)+2
         yb(1)  ];

% --------------------------------------------------------------------------

function jac = shockJac(x,y,e)
%SHOCKJAC  Evaluate the Jacobian of the ODE function
jac = [ 0   1
        0 -x/e ];

% --------------------------------------------------------------------------

function [dBCdya,dBCdyb] = shockBCJac(ya,yb,e)
%SHOCKBCJAC  Evaluate the partial derivatives of the boundary conditions
dBCdya = [ 1 0
           0 0 ];

dBCdyb = [ 0 0
           1 0 ];
```

Bibliography

[1] J.Kierzenka and L.F.Shampine. *A BVP Solver Based on Residual Control and the MATLAB PSE.* ACM Transactions on Mathematical Software, Vol. 27, N0.3, 299-316. (2001).

[2] J.R.Cash and A.Singhal. *High-Order Methods for the Numerical Solution of Two-Point Boundary Value Problems.* BIT 22, 184-199. (1982).

[3] J.R.Cash and M.H.Wright. *A deferred correction method for nonlinear two-point buondary value problems. Implementation and numerical evaluation.* SIAM J. Sci. Statistical Computing 12, 971-989. (1991).

[4] J.R.Cash. *32 Test Problems for Analysis of Two-Point Boundary Value Problem Solvers.* online http://www.ma.ic.ac.uk/ jcash/BVP_software/problems.ps

[5] J.R.Cash and D.R.Moore. *High-Order Interpolants for Solutions of Two-Point Boundary Value Problems Using MIRK Methods.* Computers and Mathematics with Applications, 48, 1749-1763. (2004).

[6] U.M.Ascher, R.M.R.Mattheij and R.D.Russell. *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations.* Prentice Hall, New Jersey. (1988).

[7] H.B.Keller. *Numerical Solution of Two-Point Boundary Value Problems*, SIAM Regional Conference Series in Applied Mathematics. Bristol, J.W.Arrowsmith Ltd (1976).

[8] B.Wendroff. *Theoretical Numerical Analysis.* Academic Press, New york. (1966).

[9] J.Kierzenka, L.F.Shampine and M.W.Reichelt. *Solving Boundary Value Problems for Ordinary Differential Equations in MATLAB with bvp4c.* The Mathworks, Inc, 3 Apple Hill Drive, Natick, MA01760. (2000).

[10] J.D.Lambert. *Numerical methods for ordinary differential systems: the initial value problem* John Wiley & Sons, Inc. New York, NY, USA. (1991).

[11] L.F.Shampine and M.W.Reichelt. *The MATLAB ODE suite.* SIAM J. Sci. Computing 18, 1-22. (1997).

[12] L.F.Shampine. *Interpolation for Runge-Kutta methods.* SIAM J. Numerical Analysis 22, 1014-1027, (1985).

[13] http://en.wikipedia.org/wiki/Picard-Lindel%C3%B6f_theorem

[14] L.F.Shampine, I.Gladwell and S.Thompson. *Solving ODEs with MATLAB.* Cambridge University Press. (2003).

[15] S.D.Capper and D.R.Moore. *On High Order MIRK Schemes and Hermite-Birkhoff Interpolants*, Accepted for publication in JNIAM. (2006).

[16] S.D.Capper and D.R.Moore. *NewNRK a Fortran 90 Code for solving Two-Point Boundary Value Problems*, online http://www.ma.ic.ac.uk/sdc99/bvp (2006).

[17] S.D.Capper J.R.Cash and D.R.Moore. *Lobatto-Obrechkoff MIRK Formulae for 2nd Order Two-Point Boundary Value Problems*, ICNAAM 2005 Extended Abstracts, 1-9. (2005). (2006)

[18] J.R.Cash, M.P.Garcia and D.R.Moore *Mono-implicit Runge-Kutta formulae for the numerical solution of second order nonlinear two-point boundary value problems*, Journal of Computational and Applied Mathematics, Vol 143, 275-289. (2002).