SPRINT2D: Adaptive Software for PDEs

M.Berzins R.Fairlie S.V.Pennington J.M.Ware
and
L.E.Scales *

Computational PDEs Unit, School of Computer Studies, The University of Leeds, Leeds LS2 9JT, U.K.

SPRINT2D is a set of software tools for solving both steady and unsteady partial differential equations in two space variables. The software consists of a set of coupled modules for mesh generation, spatial discretization, time integation, nonlinear equations, linear algebra, spatial adaptivity and visualization. The software uses unstructured triangular meshes and adaptive local error control in both space and time. The class of problems solved includes systems of parabolic, elliptic and hyperbolic equations; in the latter case by use of Riemann solver based methods. This paper describes the software and shows how the adaptive techniques may be used to increase the reliability of the solution procedure for a Burgers' equations problem, an electrostatics problem from elastohydrodynamic lubrication, and a challenging gas jet problem.

Categories and Subject Descriptors: G.1.8 [**Numerical Analysis**]: Partial Differential Equations—*method of lines; hyperbolic equations; elliptic equations*; G.4 [**Mathematics of Computing**]: Mathematical Software

General Terms: Algorithms

Additional Key Words and Phrases: Adaptivity, Error control, Finite Volume methods

1. INTRODUCTION

Two important trends in the development of numerical software for partial differential equations are to make numerical methods more applicable to problems defined on quite general geometries through the use of unstructured meshes, and more reliable through the use of adaptive error control e.g. [Adjerid, Flaherty et al. 1992]. The aim of this paper is to describe the SPRINT2D software (Software for PRoblems IN Time in 2 space Dimensions), to show how it uses adaptive unstructured meshes and to illustrate the performance of the software on a number of case studies.

The SPRINT2D package was designed primarily to solve time (in)dependent convection-diffusion-reaction equations on unstructured triangular meshes by using a finite volume method in space coupled to the method of lines for time integration. The starting point for the software was the SPRINT1D code [Berzins et al. 1989] and its extension to hyperbolic equations for the NAG Library [Pennington and

This work was funded by Shell Research and Technology Centre at Thornton, Chester, U.K.
Address: * Shell Research and Technology Centre, P.O. Box 1, Chester CH1 3SH

Berzins 1994]. The need to solve a wide variety of convection dominated problems on quite general geometries prompted an investigation into finite volume methods on triangles [Berzins and Ware 1995]. The SPRINT2D software itself began as part of a joint project [Ware 1993] with Shell Research Ltd., which also led to the Shell TRIFIT [Furzeland et al. 1989] and NAESOL [Scales 1993] packages.

SPRINT2D has been applied to a broad class of convection dominated problems including atmospheric dispersion [Tomlin et al. 1997], combustion [Berzins and Ware 1996], gas dynamics [Fairlie et al. 1997], shallow water equations [Sleigh et al. 1997], as well as more conventional parabolic and elliptic problems. The modular nature of the software and its general data structures allows the addition of different spatial discretisation schemes in order to extend its applicability to an even wider range of problems.

The software will be described as follows. Section 2 of this paper gives an overview of SPRINT2D. Section 3 describes the SPRINT2D spatial discretization method while Section 4 discusses the software engineering approach taken. Section 5 contains a description of the user's driving program and an outline description of the main modules, while Section 6 contains three case studies, including an engineering example from gas jet modelling. Finally Section 7 evaluates the success of the approach taken and points to future directions.

2. AN OVERVIEW OF THE SPRINT2D SOFTWARE

The SPRINT2D software solves time-dependent partial differential equations by using the method of lines to discretize in space thus reducing the PDEs to a system of ODEs (Ordinary Differential Equations) which can then be integrated using existing software packages. This separation of space and time and the use of ODE software makes it possible to combine different spatial and temporal discretization as required.

The primary spatial discretization module in SPRINT2D is a cell-centred finite volume method. In this scheme the PDEs are integrated over an element and the divergence theorem applied to replace the area integral for the fluxes by a line integral around the edge of the element. The flux functions in the PDEs are then used to calculate the numerical fluxes between adjoining elements. Although the finite volume method may use any form of spatial elements, the use of triangular elements allows complex domains to be modelled, and when used in conjunction with spatial adaptivity provides a powerful modelling environment, e.g. [Berzins and Ware 1996]. This is particularly true when temporal local error control and spatial error estimation and control are used.

The PDE system solved by this SPRINT2D module is:

$$\boldsymbol{\beta}\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{f^x}}{\partial x} + \frac{\partial \mathbf{f^y}}{\partial y} = \frac{\partial \mathbf{g^x}}{\partial x} + \frac{\partial \mathbf{g^y}}{\partial y} + \mathbf{S}, \tag{1}$$

where all the functions $\boldsymbol{\beta}$, $\mathbf{f^x}$, $\mathbf{f^y}$, $\mathbf{g^x}$, $\mathbf{g^y}$ and $\mathbf{S}$ are allowed to depend on $\mathbf{U}$, $\mathbf{x}$ and t, and $\mathbf{g^x}$, $\mathbf{g^y}$ depend also on $\partial\mathbf{U}/\partial x$ and $\partial\mathbf{U}/\partial y$. For steady problems $\boldsymbol{\beta}$ is set to zero. $\mathbf{f^x}$ and $\mathbf{f^y}$ are the convective fluxes which may give rise to wave-like structures in the solution $\mathbf{U}$, and $\mathbf{g^x}$ and $\mathbf{g^y}$ are the diffusive fluxes. The source term $\mathbf{S}$ can be used to add other processes such as reaction terms including chemical kinetics. The three types of boundary conditions allowed by the package are Dirichlet, Neumann and

flux conditions in which the solution, normal derivatives and fluxes are specified.

The SPRINT2D software has the structure shown in Figure 1 and is implemented

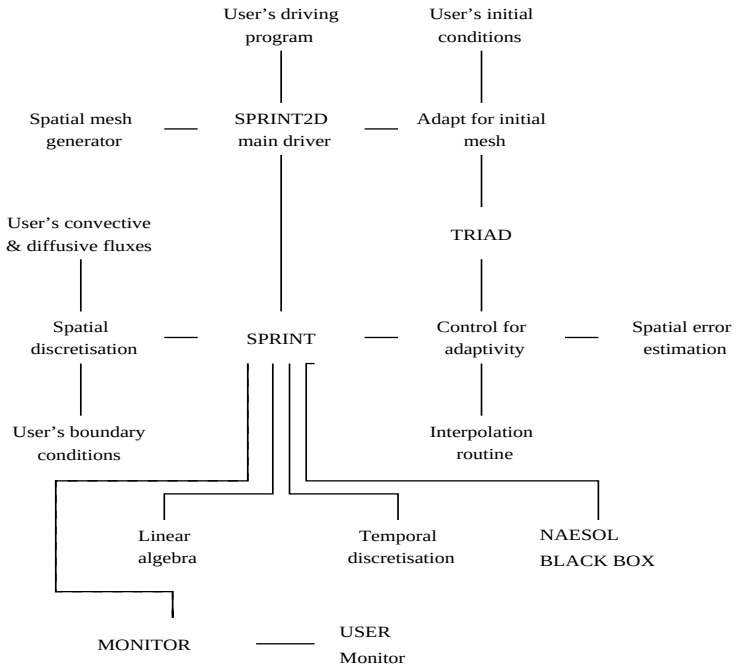

Fig. 1. Outline of SPRINT2D software

on top of two existing numerical packages: SPRINT and NAESOL - all the codes being written in C. After applying spatial discretization to time-dependent problems, the resulting system of ODEs is integrated in time by the SPRINT integrators. The user specifies the time integration module (Backward Differentiation Formula or Theta method) and the linear algebra module (sparse, iterative or operator splitting) to be used. Spatially discretizing steady problems results in a system of non-linear equations which are solved by the non-linear solver package NAESOL [Scales 1993]. The TRIAD package [Ware 1993] provides the routines to perform any spatial adaptivity using h–refinement. The modular nature of the software allows additional solution modules to be added to the package.

An important part of the specification process for solution of two space dimensional PDEs is the definition of the spatial region over which the problem is to be solved. Once this is done, this region can then be meshed to provide a suitable triangulation of the domain for the numerical solution process. SPRINT2D uses unstructured triangular meshes because they can approximate arbitrary domains more easily than quadrilateral based meshes.

The initial meshes used in SPRINT2D are created from a user-supplied geometry description using the TRIFIT [Furzeland et al. 1989] or GEOMPACK [Joe 1991] mesh generators. GEOMPACK constructs the mesh by decomposing the input geometry into simpler polygons and then meshing these polygons. As a

semi-automatic mesh generator, GEOMPACK requires additional information to accomplish this. By supplying this information the user is able to control various aspects of the final mesh such as desired number of triangles, mesh smoothness, and the way in which the geometry is decomposed into simpler polygons.

The user must write a C driving program for SPRINT2D which specifies the PDEs and the solution techniques to be used. The first part of the driving program must include the relevant header files for the SPRINT2D package and modules that are to be used. Various set-up routines specify the techniques and options required. Some of these are essential but others are optional, with default values being used for those parameters and options not set. The driving program also needs to contain the following information: the name of the file containing a specification of the physical domain; relative and absolute tolerances for the adaptivity routines; functions which specify the PDEs and the boundary conditions (depending on the spatial discretization module used); and initial conditions. These functions have to follow a fixed interface in returning values to SPRINT2D. The user may also provide a monitor routine which provides a means of examining the numerical solution at the end of every timestep.

3. SPATIAL TRIANGULAR MESH DISCRETIZATION METHOD

SPRINT2D and its underlying data structures are designed to support a wide variety of discretization methods on triangles. For example a finite element discretization scheme has recently been implemented within the same framework, and the data structures have the connectivity required for cell-vertex (as opposed to cell-centred) finite volume schemes. The primary scheme used at present is a cell-centered finite volume scheme [Berzins and Ware 1995] which uses triangular elements as the control volumes over which the divergence theorem is applied.

In order to illustrate the scheme, consider the scalar form of the PDE system defined by equation (1) with appropriate boundary and initial conditions. The finite volume representation of the solution is formally piecewise constant within each control volume. To allow the construction of high order schemes however the centroid of the triangle is defined as the nodal position and the solution value is associated with that point. In Figure 2 for example, the solution at the centroid of triangle i is U_i, and the solutions at the centroids of the triangles surrounding triangle i are U_j, U_k and U_l. The coordinates of the mesh point at which a solution value, say U_s, is defined are denoted by (x_s, y_s) .

Integration of the scalar form of equation (1) on the ith triangle gives:

$$\int_{A_i} \left[\left(\beta \frac{\partial U}{\partial t} - S \right) + \left(\frac{\partial f^x}{\partial x} + \frac{\partial f^y}{\partial y} \right) - \left(\frac{\partial g^x}{\partial x} + \frac{\partial g^y}{\partial y} \right) \right] d\Omega = 0, \qquad (2)$$

where A_i is the area of triangle i and Ω is the integration variable defined on A_i. The area integral of the first bracketed term in equation (2) is approximated by a one point quadrature rule, the quadrature point being the centroid of triangle i:

$$\int_{A_i} \left(\beta \frac{\partial U}{\partial t} - S \right) d\Omega = A_i \left(\beta_i \frac{\partial U_i}{\partial t} - S_i \right).$$

By using the divergence theorem, the second bracketed term in equation (2) is

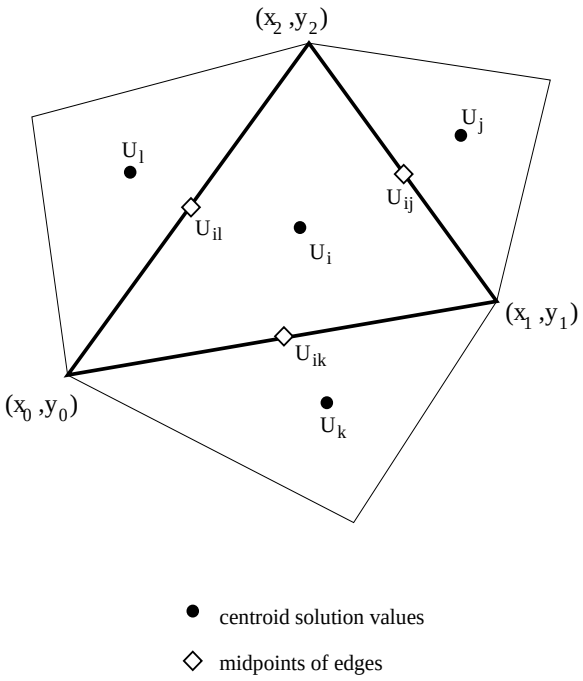

Fig. 2. Example triangle used in discretization description

replaced by a line integral around the triangular element:

$$\int_{A_i} \left(\frac{\partial f^x}{\partial x} + \frac{\partial f^y}{\partial y} \right) d\Omega = \oint_{C_i} (f^x n_x + f^y n_y) ds,$$

where C_i is the circumference of triangle i, (n_x, n_y) is the unit outward normal to the circumference, and s is the integration variable along that circumference. The line integral along each edge is approximated by using the midpoint quadrature rule with the numerical flux being evaluated at the midpoint of the edge. Applying the same treatment to the diffusive fluxes results in the following discrete form of equation (2):

$$\begin{aligned} \beta_i \frac{\partial U_i}{\partial t} - S_i \; = -\frac{1}{A_i} (& f^x_{ik} \Delta y_{0,1} \; - \; f^y_{ik} \Delta x_{0,1} + f^x_{ij} \Delta y_{1,2} \\ & - f^y_{ij} \Delta x_{1,2} \; + \; f^x_{il} \Delta y_{2,0} - f^y_{il} \Delta x_{2,0} \\ & - (g^x_{ik} \Delta y_{0,1} \; - \; g^y_{ik} \Delta x_{0,1} + g^x_{ij} \Delta y_{1,2} \\ & - g^y_{ij} \Delta x_{1,2} \; + \; g^x_{il} \Delta y_{2,0} - g^y_{il} \Delta x_{2,0})), \end{aligned}$$

where $\Delta x_{i,j} = x_j - x_i$, $\Delta y_{i,j} = y_j - y_i$, and f^x_{ij}, f^y_{ij}, g^x_{ij} and g^y_{ij} are the fluxes in the x and y directions evaluated at the midpoint of the triangle edge separating the triangles associated with U_i and U_j.

The convective fluxes f^x_{ij} and f^y_{ij} are evaluated by using approximate Riemann solvers (see later) denoted by f^x_{Rm} and f^y_{Rm} respectively. At the midpoint of the edge between triangles i and j, one-dimensional Riemann problems are solved in the

Cartesian directions with the *left* solution value U^l_{ij} being defined as that internal to triangle i, and the *right* solution value U^r_{ij} being defined as that external to triangle i:

$$\begin{aligned}\beta_i\frac{\partial U_i}{\partial t} - S_i \;= -\frac{1}{A_i}\Big(\; & f^x_{Rm}(U^l_{ik},U^r_{ik})\Delta y_{0,1} - f^y_{Rm}(U^l_{ik},U^r_{ik})\Delta x_{0,1} + \\ & f^x_{Rm}(U^l_{ij},U^r_{ij})\Delta y_{1,2} - f^y_{Rm}(U^l_{ij},U^r_{ij})\Delta x_{1,2} + \\ & f^x_{Rm}(U^l_{il},U^r_{il})\Delta y_{2,0} - f^y_{Rm}(U^l_{il},U^r_{il})\Delta x_{2,0} - \\ & (g^x(U^c_{ik},(U_{ik})_x)\Delta y_{0,1} - g^y(U^c_{ik},(U_{ik})_y)\Delta x_{0,1} + \\ & g^x(U^c_{ij},(U_{ij})_x)\Delta y_{1,2} - g^y(U^c_{ij},(U_{ij})_y)\Delta x_{1,2} + \\ & g^x(U^c_{il},(U_{il})_x)\Delta y_{2,0} - g^y(U^c_{il},(U_{il})_y)\Delta x_{2,0}\;)),\end{aligned} \tag{3}$$

where U^c_{ij} is a solution at the midpoint of the edge computed using the values on both sides (see [Berzins and Ware 1995]) and $(U_{ij})_x$ is the derivative with respect to x at the midpoint.

A standard first-order scheme uses the piecewise constant solution on either side of the edge as the upwind values, e.g.

$$U^l_{ij} = U_i, \;\; U^r_{ij} = U_j\,.$$

Although this scheme results in numerical solutions with no undershoots or overshoots, the amount of numerical diffusion introduced is often not acceptable. For this reason a second-order variant of this method is implemented [Berzins and Ware 1995]. This method uses limited linear interpolants to construct the left and right values at each edge mid-point. This scheme is nonlinear even for a linear problem and so falls into the class of schemes discussed in [Venkatakrishnan 1995].

3.1 Example Advection Diffusion Problem

Consider for example the two-dimensional advection-diffusion equation:

$$\frac{\partial U}{\partial t} + a\frac{\partial U}{\partial x} + b\frac{\partial U}{\partial y} = c\frac{\partial^2 U}{\partial x^2} + c\frac{\partial^2 U}{\partial y^2}$$

where a and b are positive constants for example. The discrete form (see equation (3)) is given by

$$\begin{aligned}\frac{\partial U_i}{\partial t} = -\frac{1}{A_i}\Big(\; & aU^l_{ik}\Delta y_{0,1} - bU^r_{ik}\Delta x_{0,1} + \\ & aU^l_{ij}\Delta y_{1,2} - bU^r_{ij}\Delta x_{1,2} + \\ & aU^r_{il}\Delta y_{2,0} - bU^l_{il}\Delta x_{2,0} - \\ & (c(U_{ik})_x\Delta y_{0,1} - c(U_{ik})_y\Delta x_{0,1} + \\ & c(U_{ij})_x\Delta y_{1,2} - c(U_{ij})_y\Delta x_{1,2} + \\ & c(U_{il})_x\Delta y_{2,0} - c(U_{il})_y\Delta x_{2,0}\;)),\end{aligned} \tag{4}$$

assuming that the triangle is aligned to the characteristic directions in the manner shown in Figure 2, and given that the solution to the Riemann problem is the product of the upwind value and either a or b.

3.2 Riemann Solvers

For convection dominated PDEs, correct specification of the flux and careful space discretization are essential to avoid unphysical oscillations in the discrete solution. A standard approach to ensure a stable solution is to place more emphasis on the

information coming from the direction of the flow (the upwind values) in discretizing the advective parts of the PDE. For simple problems, e.g. linear advection, the choice of upwinding direction is obvious. However, for complex systems the direction may alternate, or a combination of both left and right values may be needed for a system of PDEs. As stated in Section 3, in these cases an approximate (sometimes exact) Riemann solver is used to calculate the advective flux in the code using a combination of knowledge about the PDE and left and right solution values, see e.g. [Pennington and Berzins 1994]. As an example consider the viscous Burgers' equation given by

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}\left(\frac{u^2}{2}\right) + \frac{\partial}{\partial y}\left(\frac{u^2}{2}\right) = p\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right), \tag{5}$$

$$(x, y, t) \in [0, 1] \times [0, 1] \times (0, 1.25].$$

Defining the convective flux functions by simple averaging of the left and right values `u_l`, `u_r` respectively, results in the following code for the numerical flux values:

```
u = ( u_l + u_r ) / 2.0 ;
f_x = 0.5*u*u;
f_y = 0.5*u*u;
```

and may lead to unphysical negative solution values. The negative values vanish when Roe's Riemann solver, see e.g. [Pennington and Berzins 1994], is implemented by replacing the above evaluation of `u` by:

```
if ( u > 0.0 ) u = u_l;
else       u = u_r;
```

before the assignment to `f_x` and `f_y`. The Riemann solver routine from the driving program for this problem is given in Appendix B. In the case of the jet problem described in Section 5.2 below, the fluxes $\mathbf{f^x}$ and $\mathbf{f^y}$ must be calculated given left $\mathbf{U}_L = (\rho, \rho v_r, \rho v_z, E)_L^T$ and right $\mathbf{U}_R = (\rho, \rho v_r, \rho v_z, E)_R^T$ solution values at the midpoint of each edge. This calculation is a nontrivial task, see e.g. [Pennington and Berzins 1994] and [Fairlie et al. 1997].

4. SOFTWARE ENGINEERING AND USER INTERFACE ISSUES

SPRINT2D successfully combines several existing packages into a cohesive environment for the user. Since the existing packages were not originally intended to form part of a larger package, a new control layer was inserted between the user and the underlying packages. The objective of this layer is to aid the user in initialising the separate packages without removing the possibility of fine control that may be required in more complex applications. An obvious approach is to explicitly enumerate all possible inputs and choices in the interface. However such an interface may appear daunting and so in SPRINT2D, the user *constructs* an integration *object* by applying a series of operations that gradually embed more information about the problem being solved into the object. A novice user may therefore supply only the minimum information and the package will invisibly complete the specification with default settings, while an expert user can override some or all of the default settings by invoking more utility functions.

The key issue for the interface between the control layer and the packages is extendability. Rather than contain an explicit list of all possible choices of package routines, the control layer maintains a *virtual function table* containing the addresses of routines with the individual packages to be used. The virtual function table contains entries for generic routines such as: integrating for a single step; factorising a Jacobian matrix; backsubstitution solve of a Jacobian matrix; single iteration of a non-linear solver. The package specific routine names are installed in the virtual function table by *constructor* routines for the integration object. It is therefore possible to standardise the naming conventions of the SPRINT2D interface without restricting the scope of the interface.

The software is written in strict ANSI C so as to make easier use of pointer based data structures when writing unstructured mesh generation, discretization and adaptivity codes. The use of C required the translation of the SPRINT time integration software [Berzins et al. 1989] from Fortran 77. Prior to any coding changes, a comprehensive test suite was implemented for the Fortran version of SPRINT. Each following stage in the code modification was then regression tested using this test suite. The first stage was to remove any undesirable features of the Fortran code (e.g. undeclared variables, uninstantiated variables, etc). The publicly available translation tool *f2c* was then used to perform a bulk translation of all source files. The source code was then hand edited and *f2c* specific routines replaced by ANSI C counterparts or new utility routines.

5. USING SPRINT2D AND ITS MODULES

In this section a description of the user's driving program is given, along with an overview of the modules available to the user in the areas of mesh generation, time integration, linear and nonlinear equation solvers, adaptivity and visualization.

5.1 User Program

To use SPRINT2D the user must write a driving program in C. The suggested form of this program can be seen pictorially in Figure 3. The first stage is to include the relevant SPRINT2D header files:

```
#include "S2D.h"
#include "S2D_FVM_finite_volume_discretization.h"
#include "S2D_BBOX_black_box_soln_strategy.h"
```

Here the include file `S2D.h` contains the definitions of the primitive types that SPRINT2D uses. The other two include files are for the two modules the user wishes to use. Here the user intends to use a finite volume spatial discretization module and a simple black box solution strategy.

The spatial discretization will generally require the user to write some routines to specify the problem. These routines typically include ones for the initial and boundary conditions, the convective and diffusive fluxes and the source terms. Different spatial discretization modules will require the problem to be specified in a different form.

The next stage is the main driving routine (usually `main()`). This will specify different options about the integration before finally invoking SPRINT2D to carry out the integration. An *integration object* is created to contain all the relevant

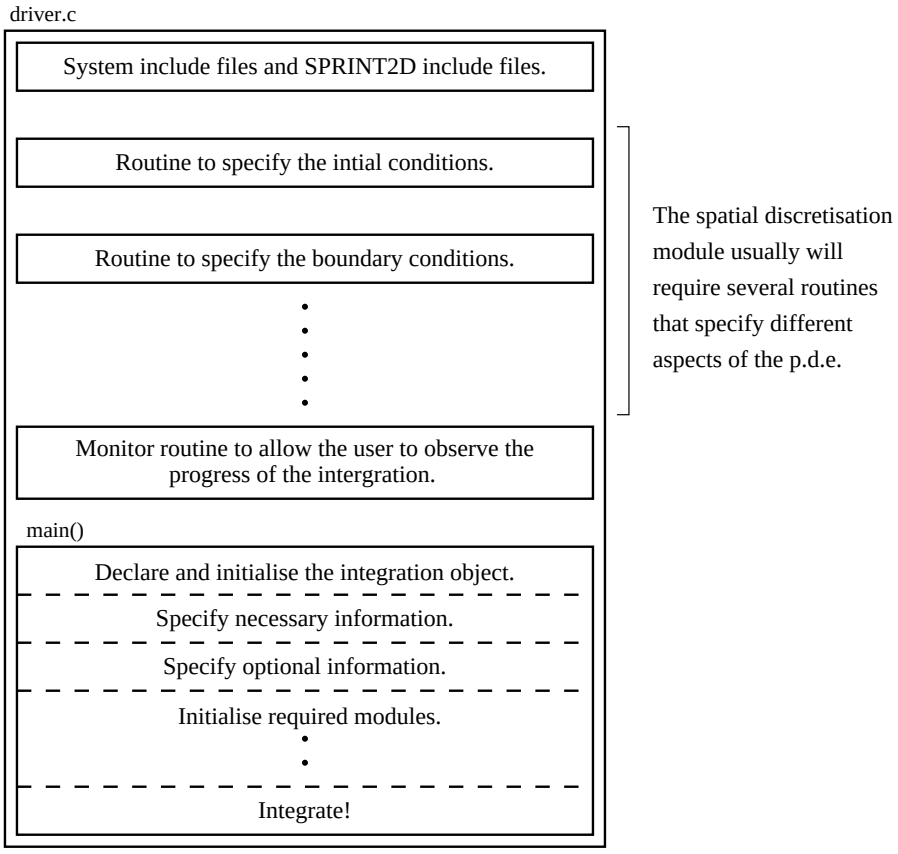

Fig. 3. Form of the SPRINT2D driving program

details and options that the user specifies. Before the integration object can be used it must be initialised as follows:

```
void    main( int argc, char *argv[] )
{
        S2D_Intgrtn_Obj_Type    my_integ ;

        S2D_initialise( &my_integ ) ;
```

where **my_integ** is the integration object. The act of initialising the integration object is to fill it with default values chosen by SPRINT2D. The user may then replace the defaults using utility routines. Prior to any optional values being specified certain essential values or options must be specified. The first is specify whether the integration is time-dependent or steady, e.g.

```
        S2D_time_dependent( &integ_obj, npde, neqmax, t_start,
                            n_t_out, t_out ) ;
        S2D_steady( &integ_obj, npde, neqmax ) ;
```

where **npde** is number of PDEs, **neqmax** is the maximum number of unknowns

allowed, `t_start` is the initial time, `n_t_out` is the number of fixed output times and `t_out` is an array of output times.

Time-dependent and steady integrations have their own essential options and values that must be set. The essential options include the choice of modules with which to perform the integration. Time-dependent integrations must specifiy a spatial discretization module, temporal discretization module and linear algebra module. Steady integrations must specify a spatial discretization module and solution strategy module. These modules will then have essential and optional values that must/can be set.

Once all the necessary information and any optional information has been attached to the integration object the integration can be started with the function call:

```
S2D_integrate( &my_integ ) ;
```

The SPRINT2D package then performs the required integration. The user can monitor the integration by supplying a monitor function in the driving program. This monitor function is called at several stages in the code, e.g. at the end of each time step and at the user-supplied fixed output times.

5.2 Finite Volume Discretization Module

The finite volume discretization module is based on algorithms described in [Berzins and Ware 1995]. The solution is represented as a set of piecewise constant triangular element with the solution values associated with the centroids of the triangles. The user describes the PDE system using the master equation template for time-dependent problems given by equation 1. These terms are defined in the following way:

$\boldsymbol{\beta}\left(\mathbf{x}, t, \mathbf{U}\right)$	Optional function to specify temporal nature.
$\mathbf{f}\left(\mathbf{x}, t, \mathbf{U}\right)$	Essential function: *convective* fluxes.
$\mathbf{g}\left(\mathbf{x}, t, \mathbf{U}, \frac{\partial}{\partial \mathbf{x}}\mathbf{U}\right)$	Essential function: *diffusive* fluxes.
$\mathbf{S}\left(\mathbf{x}, t, \mathbf{U}\right)$	Optional function: source terms.

The required functions are specified in user supplied C functions in the driving program. The names of these user functions are passed to the discretization routine by the calls

```
S2D_FVM_initialise( &my_integ ) ;
S2D_FVM_riemann_solver( &my_integ, electro_rs ) ;
S2D_FVM_diffusive_flux( &my_integ, electro_g ) ;
```

for example. The routine `S2D_FVM_initialise` instructs SPRINT2D to use the finite volume spatial discretization module when integrating. The function `electro_rs` contains the Riemann solver which specifies the convective fluxes as described in Section 3.2. Examples of the user-supplied functions are given in the appendix.

5.2.1 *Specification of Initial and Boundary Conditions.* The user writes two functions specifying the initial and boundary conditions and passes the names of these to the discretization routine by the calls

```
S2D_FVM_initial_conditions( &my_integ, electro_ic ) ;
```

```
S2D_FVM_boundary_conditions( &my_integ, electro_bc ) ;
```

in the driving program. Appendices A and D contain examples of these routines. The Riemann solver is used to implement flux or derivative boundary conditions. Reflective boundary conditions are imposed by setting the exterior 'normal' velocity to be the opposite sign to the normal velocity at the boundary from the interior. The values of all the other variables on the 'exterior' being the same as the interior values. All other 'outside' solution values are the same as those on the interior.

5.3 Mesh Generation

The initial creation of the mesh is performed by the user by using either a visual interface [Berzins et al. 1997] or by creation of a file which specifies a hierarchy of points, lines and objects in which the higher dimensional objects are composed in terms of the lower dimensional ones. In this latter case the form of the file originates from the work of [Furzeland et al. 1989] and may be illustrated by considering the case of a square domain with vertices at (0,0), (1,0), (1,1) and (0,1). In this case the domain is described by

```
DOMAIN square
VERTICES
          1      0.0     0.0
          2      1.0     0.0
          3      1.0     1.0
          4      0.0     1.0
ZERO_D_SUBDOMAINS
        101       1
        102       2
        103       3
        104       4
ONE_D_SUBDOMAINS
        201       s     101     102
        202       s     102     103
        203       s     103     104
        204       s     104     101
TWO_D_SUBDOMAINS
        301     201     202     203     204
END_OF_DOMAIN
```

The file, say `burgers.dmn`, containing this description is then passed to a setup routine for either the TRIFIT [Furzeland et al. 1989] or GEOMPACK [Joe 1991] mesh generators, e.g.

```
S2D_trifit_mesh_generator( &my_integ, "burgers.dmn", ilevel ) ;
S2D_geompack_mesh_generator( &my_integ, "burgers.dmn", ntri, ilev ) ;
```

In these calls there is a mesh level parameter `ilev` to specify the number of levels of uniform refinement of the coarse mesh, and a pointer to the integration object `my_integ`. The parameter `ntri` in the call to `S2D_geompack_mesh_generator` is the target number of triangles in the coarse mesh.

5.4 Time Integration

Although in many time-dependent PDE codes a CFL stability condition is used to control the timestep, the SPRINT2D Theta [Berzins and Furzeland 1992] or Backward Differentiation Formula codes with functional iteration, Newton Krylov or operator splitting methods allow automatic control of the local error. The interface to the integration methods is essentially a C implementation of the Fortran routines described in [Berzins et al. 1989], but with no reverse communication. Level 1 BLAS are also now utilised.

The local error control used is the same as that in SPRINT:

$$||\mathbf{le}(t_n)||_w \ < \ 1, \tag{6}$$

where the weighted norm means that the ith component of the time local error $le_i(t_n)$ in the norm calculation is replaced by

$$\frac{le_i(t)}{atolt_i + rtolt_i \times u_i}, \tag{7}$$

where $atolt$ and $rtolt$ are the user-supplied absolute and relative time error tolerances, set by the user via the call:

```
S2D_temporal_tol( &my_integ, S2D_Scalar_TOL, &atolt, &rtolt ) ;
```

in the driving program. In this case the tolerances are scalar values (as specified by `S2D_Scalar_TOL`), but for PDE *systems* the user will usually supply vectors of length $npde$ in order to apply different tolerances to each PDE variable.

Efficient time integration requires that the spatial and temporal errors are roughly the same order of magnitude. The need for spatial error estimates unpolluted by temporal error requires that the spatial error is the larger of the two errors. The SPRINT2D software also has an option to use the strategy of Berzins, see [Berzins 1995], which controls the local time error so that

$$||\mathbf{le}(t_n)||_w \ < \ \epsilon||\mathbf{es}(t_n)||, \tag{8}$$

where $\mathbf{es}(t_n)$ is the growth in spatial discretization error over the timestep. In the case of convection dominated problems, implicit methods with functional iteration are used, thus automatically imposing a hidden CFL type condition, [Berzins 1995].

5.5 Mesh Adaptivity

The general approach adopted in SPRINT2D is one of static rezoning in which the mesh is adapted at discrete times, the solution interpolated onto the new new mesh and integration recommenced either via full restart or a flying restart, [Berzins et al. 1989]. The times at which remeshing takes place are chosen purely on the basis of growth in the spatial error. The discrete remeshing interface consists of a function which is called to define the mesh refinement or mesh coarsening level in terms of the number of mesh levels to be added (or removed) on a particular triangle. In this way these meshes are refined and coarsened by the TRIAD [Ware 1993] adaptivity module which uses data structures to enable efficient mesh adaptation. For the ith PDE component on the jth triangle, a local error estimate $e_{i,j}(t)$ is calculated from the difference between the solution using a first order method and that using a second order method, see [Berzins and Ware 1996] for details. For

time dependent PDEs this estimate shows how the spatial error grows locally over a time step. A refinement indicator for the jth triangle is defined by an average scaled error ($serr_j$) measurement over all $npde$ PDEs

$$serr_j = \sum_{i=1}^{npde} \frac{e_{i,j}(t)}{atols_i/A_j + rtols_i \times u_{i,j}}, \tag{9}$$

where $atols$ and $rtols$ are the user-supplied absolute and relative spatial error tolerances and where $e_{i,j}(t)$ is the growth in spatial error over the timestep for the ith PDE at the jth mesh point. The tolerances are set by the user in a similar manner to the time integration tolerances, e.g.

```
S2D_spatial_tol( &my_integ, S2D_Vector_TOL, rtols, atols ) ;
```

The default action of SPRINT2D is to use spatial adaptivity. The user can switch off adaptivity by calling a routine from the main driving program or from the monitor routine, which is useful if for example the user wishes to switch off adaptivity after a certain time. There are also a number of other optional parameters which can be set via function calls in order to give further control over the adaptivity.

This formulation for the scaled error provides a flexible way to weight the refinement towards the error in any of the PDEs. An integer refinement level indicator is calculated from this scaled error to give the number of times the triangle should be refined or derefined.

In the refinement case, all the neighbouring triangles which share an edge with a refined triangle are refined towards that shared edge. Similarly, all triangles with a vertex in common with the original triangle are refined towards that vertex. Finally Bank's green rule is applied to ensure the mesh is conforming, [Bank 1995]. This is illustrated in Figure 4 in which a level 2 refinement is applied to the central triangle where dashed lines represent the bisecting edges of green triangles.

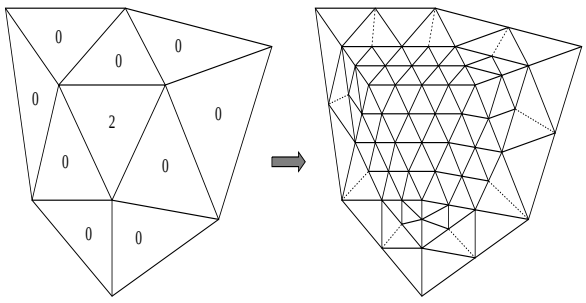

Fig. 4. Regular and green refinement

De-refinement is a reversal of the refinement process, that is, the four children created through regular subdivision can be deleted, leaving the parent. Only one level of de-refinement is allowed at any one remeshing time and, in addition, all four children must be marked for deletion. De-refinement will not be allowed if a triangle in the initial mesh, produced by the mesh generator, is specified. The triangles created as a result of application of the green rule may be of poor quality and so are removed before any further mesh refinement takes place [Bank 1995].

5.6 Linear Algebra

The linear algebra options available to the user are those in the SPRINT software i.e. full, banded and sparse matrices, with the additional option of iterative methods (preconditioned orthomin) [Jackson and Seward 1993]. Each module has a setup routine, a formulate matrix or preconditioner routine and a solve routine.

5.7 Steady Problems: calling NAESOL from SPRINT2D

Steady problems are solved inside SPRINT2D either by time marching to steady state or by using the powerful NAESOL nonlinear equations solver. This solver is written in C in an object style and provides a fully integrated range of techniques from which it is possible to build a solution capability tailored to a specific problem in such a way as to optimize reliability, computational effort and memory usage. It is constructed from four classes of object. For example, the dynamic environment class consists of nonlinear algorithms, e.g., robust Newton, truncated Newton and homotopy algorithms, while the static environment class covers initialization, termination and error handling for the dynamic objects.

The object oriented structure is particularly powerful in the present context as it permits transparent integration of class members into a tailored nonlinear equation solver with sensible defaults and short parameter lists, with additional function calls to provide extra information as needed.

Locally convergent techniques, e.g. standard Newton-type methods, are limited by the fact that a root can not always be found from an arbitrary starting point. Globally convergent homotopy methods can overcome this limitation in many cases. The concept of a homotopy is concerned with the deformation of one problem into another by the continuous variation of a single parameter. One problem being easy to solve and being continuously deformed into the difficult target problem. In practice, the deformation process must be discretized and a sequence of intermediate problems solved. However, a locally convergent method can be successfully applied at each step provided the changes are sufficiently small. In this continuation [Allgower and Georg 1990] approach, solving a series of locally convergent problems provides a route to global convergence. The algorithms in NAESOL are more sophisticated than simple continuation techniques, by providing implicit step length adjustment and dealing with limit points and bifurcations. The continuation parameter can be artificial, where only the final value corresponds to a problem of interest, or it can occur in the problem itself. In the latter case, the continuous change in behaviour of the system of equations with respect to this parameter may be of interest.

5.7.1 *Using NAESOL in SPRINT2D.* The simplest level of NAESOL use is the black box level, in which the user creates the required objects, defines an initial estimate of the root, runs the solver, and finally destroys the objects. In this case the initialisation call is simply

```
S2D_BBOX_initialise( &my_integ, -1.0 ) ;
```

where `-1.0` instructs it to solve to complete accuracy.

Sometimes more control is needed over what happens at each major iteration of the nonlinear solver, such as in SPRINT2D, where, for example, it is not necessary

to refresh the Jacobian each time as is done in the black box approach. To achieve this, running the solver is broken down into setting up the static environment, initializing the dynamic environment and looping over the nonlinear iterations by carrying out one step of the main nonlinear algorithm at a time. NAESOL also provides an interface at an even lower level, the glass box level, not currently used in SPRINT2D.

5.8 Visualization and Interpolation

The driving program also allows the user to extract information about the numerical solution each time it changes or is updated. This is achieved by the user providing a *monitor* routine which SPRINT2D calls at regular intervals with a large amount of solution information. For example, each triangle has a solution value, a spatial error value and, for time-dependent problems, a temporal error value. The code can also provide a large quantity of spatial information about the unstructured mesh such as areas of triangles, lengths of edges, unit normals to edges etc. This information is used by the visualization package which complements the SPRINT2D solver. This visualization package is written in IRIS GL and runs on a local host whilst SPRINT2D generally runs on a computationally intensive platform elsewhere. Solution frames are sent across the network to the visualization package within which the user can interrogate the solution whilst the next frame is being calculated. The visualization package displays the solution values or error estimates for each triangle in the spatial mesh. The visualization package is not used to steer the calculation directly, but has proved to be invaluable for users learning how to apply adaptivity to their applications.

In displaying the numerical solution values for convection-dominated problems great care must be taken to avoid introducing physically unreal values not already present in the numerical solution. Numerical PDE solvers take great care to preserve, say, the positivity of the solution. For example physical values of density should always be positive. The solution to convection-dominated PDE problems may have shocks and discontinuities present. However, standard interpolation techniques may lead to numerical undershoot and overshoot at discontinuities, which can be misleading. A triangular based interpolant is used, see [Pratt and Berzins 1996], which achieves the desired properties by bounding the values it produces to be between the maximum and minimum values used to define it. This interpolant provides a more reliable and natural way for the user to view the solution.

6. CASE STUDIES

This section will demonstrate the use of the tools by three case studies of increasing complexity.

6.1 Viscous Burgers' Equation

The adaptive solution techniques used by the code may be illustrated by considering the PDE defined by equation 5 with an exact solution of

$$u(x, y, t) = (1 + \exp((x + y - t)/p))^{-1}.$$

The value of p is chosen to be 0.0001 so that the partial differential equation is convection-dominated, and the boundary and initial conditions are given by the

exact solution. The Riemann solver used was the Roe solver for the inviscid Burgers' equation outlined in Section 3.2, and in code form in Appendix B. The domain file is given in Section 5.3. The code for this problem with four levels of mesh refinement and a maximum of 10^5 triangles is given in Appendix A. Figure 5 shows the mesh and the solution at $t = 0.65$. Further solutions are shown in [Berzins and Ware 1994].

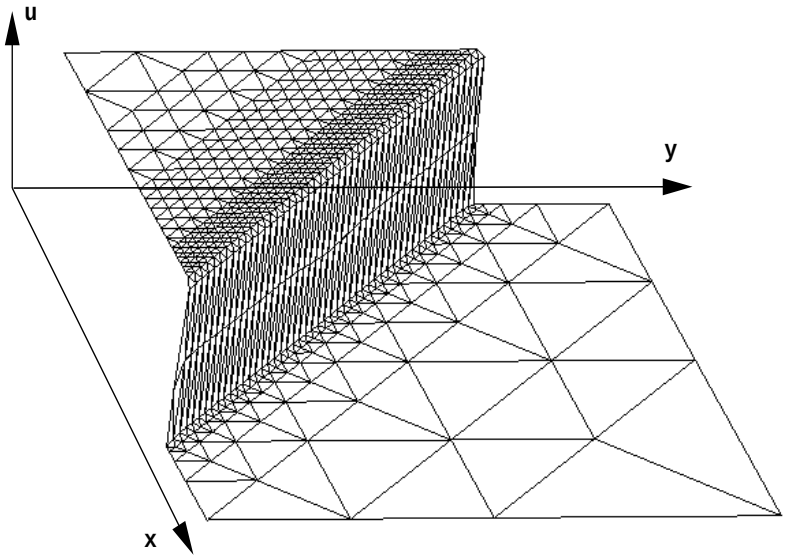

Fig. 5. Mesh and solution for the Burgers' problem

6.2 Electrostatic Elliptic Problem

Recent work on the mathematical modelling of elastohydrodynamically (EHD) lubricated line contacts e.g.[Nurgat et al. 1997] has enabled the routine computation of precise contact shapes over a wide range of operating parameters. SPRINT2D has been used to solve for the electrical potential field under an applied potential difference in such a contact shape arising from elastic deformation. This has enabled accurate computations of the energy density field (the squared gradient of the electrical potential), the integral of which over the entire contact gives the electrical capacitance.

Electrical capacitance techniques have been used for many years to measure oil film thicknesses in heavily loaded contacts. However, it has always been necessary to make simplifying assumptions about the geometry of the contact and about the relative contribution of different parts of the contact to the capacitance. The

use of unstructured mesh techniques has made it possible to compute the capacitance of any EHD line contact accurately using the correct geometry, and, equally importantly, to evaluate the accuracy of various simplifying assumptions.

The PDE is given by Laplace's equation with Dirichlet boundary conditions on the domain defined by the 35 points in Appendix C. The edge specifications have been omitted for brevity, but are simply defined as edge i connecting points i to $i+1$ except that edge 35 connects points 35 and 1. Edges 1 to 32 are named 2001 to 2032 respectively. Edges 33, 34 and 35 are the right-hand side, bottom and left-hand side of the domain and are named 2997, 2998 and 2999 respectively. The computational domain is then the interior of this region, shown in Figure 6. The boundary condition on edge 2997 is $U(x,y) = y/.49537$, on edge 2998 it is $U(x,y) = 0.0$ and on edge 2999 it is $U(x,y) = y/1.5880$. On all other edges $U = 1.0$. The initial estimate is that $U = 0$ in the interior of the domain. The code for this is shown in Appendix D.

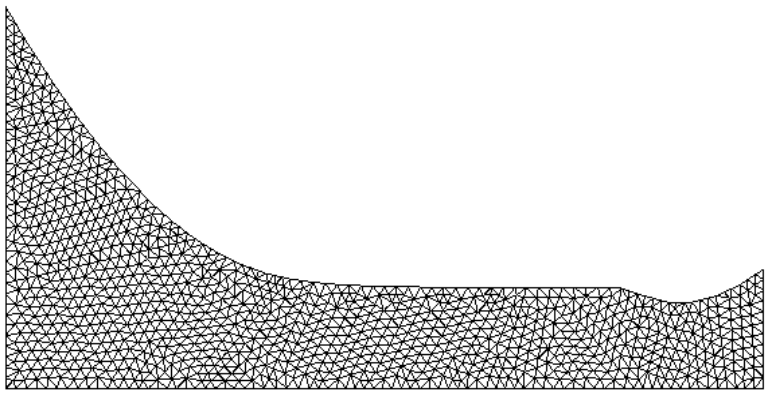

Fig. 6. Typical mesh for electrostatics problem

Figure 6 shows the GEOMPACK mesh used, given a target of 2000 triangles, and Figure 7 shows a contour plot of the energy density (the squared gradient of the electrical potential) on this fixed domain.

Using the SPRINT2D solutions, it has been possible to establish that the accuracy of a certain simplified model is such that it can be used for most capacitance modelling of EHD contacts.

6.3 Jet Modelling Problem

The 2D time dependent Euler equations in axial symmetry are used to model the gas flow resulting from the rupture of a high pressure gas pipeline containing gas at a higher pressure than the surrounding ambient air medium. The resulting flow consists of a rapid expansion of the pipeline gas into the surrounding medium in the form of a compressible jet. The flow is multi-phase but for ease of exposition an expanding air jet into air is considered. The jet is typically *underexpanded*, that is the initial jet pressure at inflow is greater than the ambient pressure.

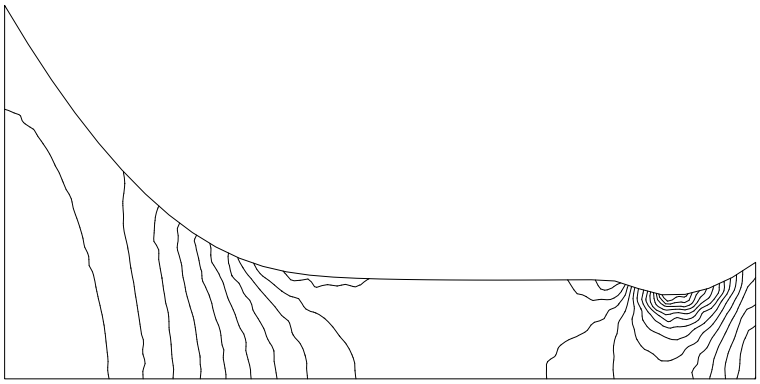

Fig. 7. Contour plot of the energy density for the electrostatics problem

It can be shown [Falle 1991] that the initial time dependent structure of the jet is approximately self-similar in time, and involves a complex structure of shocks, contact surfaces and vortices. In order to produce high quality solutions it is therefore necessary to employ shock capturing upwind schemes. The steady solution is obtained by time-stepping to a steady state. Although the time dependent Euler equations are of hyperbolic character, the time independent are of mixed elliptic/hyperbolic type depending on whether the flow is locally supersonic or subsonic respectively. This has important consequences for the boundary conditions of the problem, as along the jet outflow boundary a typical steady jet will have both subsonic and supersonic regions.

The 2D Euler equations formulated in cylindrical polar coordinates have the form [Falle 1991]:

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{1}{r}\frac{\partial (r\mathbf{F})}{\partial r} + \frac{\partial \mathbf{G}}{\partial z} = \mathbf{S}, \tag{10}$$

where r and z represent the radial and axial coordinates respectively, and

$$\mathbf{U} = \begin{bmatrix} \rho \\ \rho v_r \\ \rho v_z \\ E \end{bmatrix}, \mathbf{F} = \begin{bmatrix} \rho v_r \\ \rho {v_r}^2 + p \\ \rho v_r v_z \\ v_r(E+p) \end{bmatrix}, \mathbf{G} = \begin{bmatrix} \rho v_z \\ \rho v_r v_z \\ \rho {v_z}^2 + p \\ v_z(E+p) \end{bmatrix}, \mathbf{S} = \begin{bmatrix} 0 \\ p/r \\ 0 \\ 0 \end{bmatrix}. \tag{11}$$

The source term, $\mathbf{S}$, consists only of a single pressure term, this being associated with the non-conservation of the radial momentum. All the other quantities in $\mathbf{U}$ are conserved. The energy $\mathbf{E}$ is defined by:

$$E = \frac{1}{2}\rho({v_r}^2 + {v_z}^2) + \rho e, \tag{12}$$

where the specific internal energy: $e{=}e(\rho, p){=}\frac{p}{(\gamma-1)\rho}$.

The numerical flux is calculated using a Riemann solver based on the first order Godunov scheme available for the Euler equations. When the solution is sufficiently close to an initial state linear expansions are used. Note that the scheme is no

longer conservative as the numerical flux for each element edge is being multiplied by the ratio r_i/r_c, where r_i and r_c are the radial coordinates of the edge midpoint and the cell centroid respectively. The r_c value will be different for each of the two elements sharing an edge, so when calculating the numerical flux at an edge midpoint, different weights are applied to the Euler flux in order to get the numerical flux for each element. Inside SPRINT2D this is implemented through an *extended* Riemann solver interface where the user can return separate fluxes for the two elements involved in the calculation.

6.3.1 *Numerical Initial conditions.* The layout of the domain can be seen in Figure 8. The region upstream of the nozzle is the high pressure reservoir. Initially the pressure in this region is set to a given uniform high pressure. The density is set to be equal to this pressure to give the same temperature in the reservoir as in the ambient region. In addition the gas in the reservoir is assumed to be at rest. Downstream of the nozzle the ambient conditions occupy most of the domain initially. The exception is a region along the axis, of width equal to the nozzle width. This is set to have pressure and density equal to the reservoir to speed up the convergence to the steady state. Elsewhere in the computational domain the flow conditions are set equal to the ambient pressure and density values, with zero axial and radial velocity. The initial values of the system variables are given in Section 6.4.

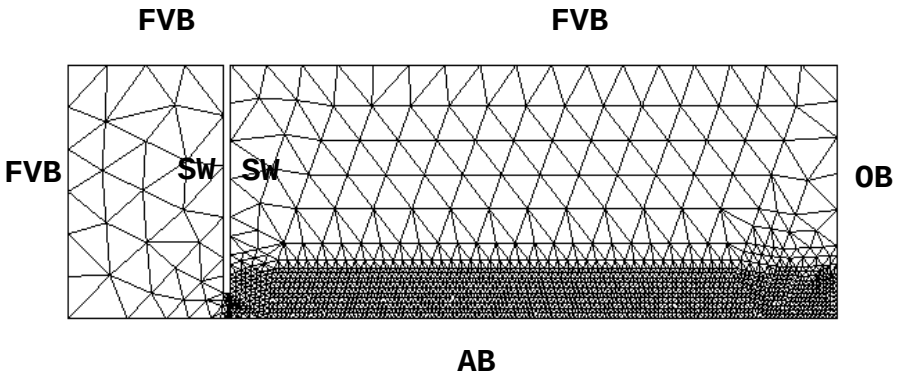

Fig. 8. Typical fixed mesh and boundary types for the jet problem. OB indicates Outflow Boundary, SW indicates Solid Wall, AB indicates Axial (reflective) Boundary and FVB indicates Fixed Variable Boundary.

6.3.2 *Numerical Boundary conditions.* The choice of the numerical boundary conditions for this problem represents the major source of difficulty in the solution to the steady jet problem. SPRINT2D requires the user to specify in the user routine the type of each boundary and associated data specific to that type, as described in Section 5.2.1. The boundary types in use in the jet problem (see Figure 8) are treated as follows :

The **solid wall boundary (SW)** is equivalent to a **reflective boundary (AB)** in non-viscous flow. The boundaries are set to be of Dirichlet type and a wall function is used to calculate mirror image 'exterior' values as described in Section 5.2.1. These values are then used in the Riemann problem.

The **fixed variable boundaries (FVB)** are those where negligible flow across the boundary is expected. The boundaries are set to be of Dirichlet type and the returned variable values are fixed at the initial values inside the domain, adjacent to the boundary.

The **outflow boundary (OB)** is the most difficult to model, as it must deal with both subsonic and supersonic outflows at the same time. This condition is implemented by setting the boundary to have a Flux condition. A function is then written to supply the fluxes across the boundary. The equations solved at the boundary are the same as the governing equations with the only difference being in the variables used to calculate the fluxes. These variables are calculated using a complex modified Riemann solver, described in detail in [Fairlie et al. 1997].

6.4 Description of Cases

Numerical results will be shown for two flow regimes: (1) the *mildly underexpanded* jet, and (2) the *highly underexpanded* jet. The mildly underexpanded jet produces a repeating shock cell structure while the highly underexpanded jet produces a single Mach disc with a long downstream barrel and core. Table 1 shows the initial pressure and density values in the reservoir and ambient regions for both cases. The length (L) and radius (R) of the domain downstream of the nozzle are also shown in terms of the nozzle diameter, D.

Case	P_{res}	ρ_{res}	P_{amb}	ρ_{amb}	L	R
1	3.5	3.5	1.0	1.0	14D	5D
2	31.0	31.0	1.0	1.0	20D	5D

Table 1. Initial values for the jet problem cases

Timestepping is controlled via absolute and relative time tolerances, both being 10^{-5}. Initially the timestep is set to be 10^{-7} and has a subsequent maximum value of 2.6×10^{-4}.

6.5 Meshing

The structures of the jets are well-known and regions which require heavier meshing can thus be predicted quite accurately. These regions are the edge of the jet, around the shock structure and the centre of the jet (where one-dimensional line plots are taken). For narrow jets this means that the majority of the jet is heavily meshed. Although initial experiments used remeshing and identified the regions where fine meshing was needed, for production runs a fixed irregular mesh such as that shown in Figure 8 was used.

This fixed mesh is obtained by coarsely meshing the domain then adding extra refinement. The initial mesh is generated using the GEOMPACK mesh generator with additional refinement points added downstream of the nozzle in rows parallel

to the $r = 0$ axis. Each row is refined to level 3 except for the two rows furthest from the axis which are refined to level 2. This is done to ensure a smoothly varying mesh. The number of initial cells, initial level of refinement and the details of the additional refinement points are contained in Table 2.

Case	1	2
Initial no. of cells	180	310
Initial refinement level	1	1
Addt. points in a row	140	200
No. of rows	8	11
No. of refinement points	1120	2200
Final no. of cells	3476	7182

Table 2. Mesh values for the jet problem

6.6 Obtaining Steady Solutions

Although time marching to a steady state is often used in CFD, convergence problems can be encountered with nonlinear schemes of this type [Venkatakrishnan 1995]. In this case such problems manifested themselves in the strong jet model in the form of waves travelled repeatedly across the domain, preventing convergence. After much investigation it was found that the waves occurred whenever the edge of the jet was too finely meshed and oscillations near the edge of the jet were unable to dissipate. To solve the problem a small amount of viscosity was added by including the Navier-Stokes diffusive terms in the PDE defined by equation (10).

6.7 Numerical Results

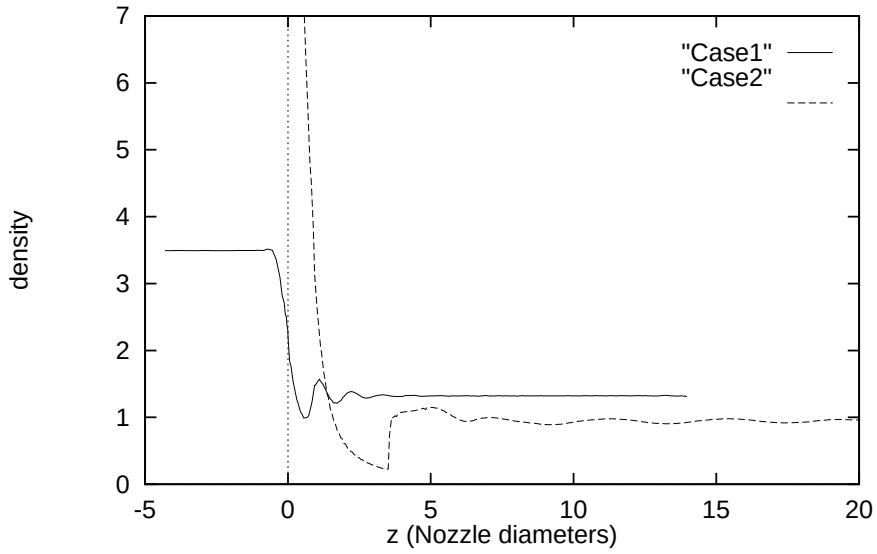

Fig. 9. Density along the line $r = 0.005$ for the jet problem

Figure 9 shows density plots for the two cases along the $r = 0.005$ line. It can be seen that increasing the strength of the jet results in a number of trends. Firstly the distance to the first shock downstream of the nozzle increases with jet strength, and the value of the minimum density which occurs at this point decreases. As expected, the wavelength downstream of this shock increases with jet strength, but the rate of decrease of their amplitudes decreases. A feature which may seem unexpected is the amplitude of the oscillations downstream of the first shock. These are small for Case 1 but are as small in Case 2, despite the large Mach disc jump in the second case. This can be explained by the presence of this Mach disc. Only Case 2 is strong enough to produce one and hence it results in a barrel shock. The oscillations are largely contained within this with only small perturbations occurring in the core of the jet; the shock cells which are prominent in the first case no longer lying along the centre of the jet.

There are two further features to be noted from Figure 9. In Case 2 the first oscillation after the Mach disc is not a symmetric wave, it contains some structure. Secondly the effect of the downstream boundary on both the plots can be seen to be so small as to be negligible.

Figures 10 and 11 show the steady jets in 2-D axial velocity contour plot form. The contours do not include the whole density range as the high reservoir values would have the effect of masking the detail downstream of the nozzle. In Figure 10 only a few shock cells exist and these quickly decrease in strength leaving the jet to simply consist of a core which is slowly narrowing and a boundary which is slowly expanding. Figure 11 displays the typical behaviour expected in a strong jet. A Mach disc exists and there is a barrel type region between the core and the jet boundary. The shock cells are now contained in this region with only small disturbances affecting the jet core. This shows why, in Figure 9 the amplitude of the waves along the centreline in this case are so small. In Figure 11 a small area of recirculation can be seen just downstream of the Mach disc.

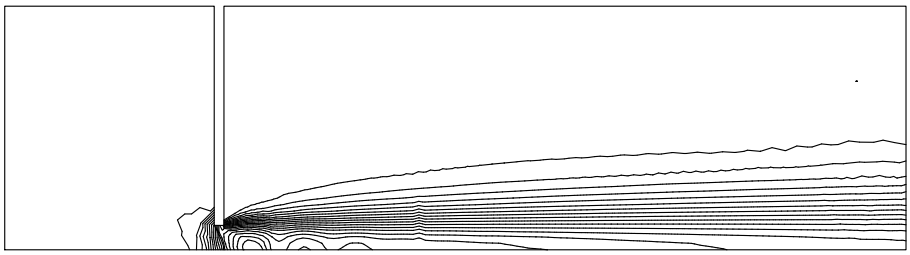

Fig. 10. Axial velocity contour plot for the jet problem, case 1

7. CONCLUSIONS AND FUTURE DEVELOPMENTS

The main lesson from the construction of the SPRINT2D software is that it is possible to write a general package for the efficient and reliable solution of a broad class of convection dominated physical problems using adaptive methods in space and time. The case studies in this paper and work on shallow water equations [Sleigh et al. 1997], atmospheric dispersion [Tomlin et al. 1997] and combustion problems

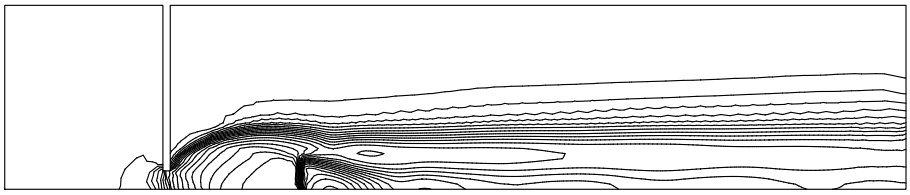

Fig. 11. Axial velocity contour plot for the jet problem, case 2

[Berzins and Ware 1996] have provided convincing evidence of this. The modularity of the software makes it possible to add efficient components for particularly important and/or difficult problems; one example of this being the operator splitting iterative scheme described in [Berzins and Ware 1996].

There are a number of important areas for future developments. New spatial discretization methods and time integration methods are being developed. With regard to the spatial adaptivity, the interface has been seen by advanced users as too automatic, in that they need to experiment with different forms of error control strategies when developing new applications. Examples of such strategies are remeshing only after a fixed number of timesteps, and completely user-defined remeshing.

A key development is the automation of the use of SPRINT2D, and the recent construction of a problem solving environment (PSE) [Berzins et al. 1997] has partially achieved this. This PSE consists of a number of tools which ease the problem specification phase. A Visual Domain Specification (VDS) tool aids the key task of specifying the initial domain prior to meshing, and a Visual Problem Specification (VPS) tool creates a suitable driving program for the numerical software via a preprocessing step and so helps to avoid the need for explicit programming. Users are generally enthusiastic about the VDS and VPS tools, particularly about the easy generation of working code. The short-term benefit of this however must be balanced against the many months of effort sometimes spent on difficult problems, such as the jet problem, experimenting with different meshes, tolerances, Riemann solvers and initial conditions.

ACKNOWLEDGMENTS

Thanks are due to Shell Research for funding the SPRINT2D project, EPSRC and Shell for funding a CASE award for one of the authors (JMW) during which time the software was written in prototype form, and the IMA at the University of Minneapolis and the Computer Science Department at the University of Utah for supporting the first author as a visitor while parts of this paper were written.

APPENDIX

A. DRIVER PROGRAM FOR BURGERS EQUATION PROBLEM

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
/* Normal include files for SPRINT2D. */
#include "S2D.h"
#include "S2D_FVM_finite_volume_discretisation.h"
#include "S2D_STHMID_theta_temporal_discretisation.h"
#include "S2D_SPWATSIT_sparse_iterative_solver.h"
/* Extra SPRINT2D include files to access the solution. */
#include "S2D_mesh_data_objects.h"
#include "S2D_FVM_mesh_data_objects.h"
/* Type to store problem data in. */
struct my_struct_st { double  p ; } ;
typedef struct my_struct_st     My_Struct_Type ;

void burgers_u( double x, double y, double t, void *users_data,
        double *u )
{ /* Exact solution. */
        My_Struct_Type  *my_struct = (My_Struct_Type *) users_data ;
        double  B, p ; /* Factor inside exponential. */
        p = my_struct->p ;
        B = (x + y - t) / p ;
        if (B > 300.0) { u[0] = 0.0 ;}
        else { u[0] = 1.0 / (1.0 + exp(B)) ;}
}

void burgers_ic( TRIAD_Triangle *tri, int npde, double x, double y,
          double t, int sub_name, void *users_data, double *u )
{ /* Initial conditions. */
     burgers_u( x, y, t, users_data, u ) ;
}

void burgers_bc( TRIAD_Line *line, int npde, double x,
          double y, double time, int edge_name, int sub_name,
          double norm_x, double norm_y, double *u,
          void *users_data, S2D_FVM_BC_Type *type, double *ub,
          double *dubdn, double *fb )
{ /* Boundary conditions. */
     type[0] = S2D_FVM_Dirichlet ;
     if (norm_x + norm_y > 0.0) { ub[0] = u[0] ;
     } else { burgers_u( x, y, time, users_data, ub ) ; }
}

void burgers_g( TRIAD_Line *line, int npde, double x, double y,
```

```
          double t, int sub_name, double norm_x, double norm_y,
          double *u, double *dudx, double *dudy, void *users_data,
          double *g_x, double *g_y )
{ /* Diffusion function. */
     My_Struct_Type  *my_struct = (My_Struct_Type *) users_data ;
     double          p ;
     p = my_struct->p ;
     g_x[0] = p * dudx[0] ;
     g_y[0] = p * dudy[0] ;
}

void main( int argc, char *argv[] )
{ /* Driving program. */
     S2D_Intgrtn_Obj_Type  my_integ ;
     My_Struct_Type        my_struct ;
     int                   n_t_out = 20 ;
     double      t_out[] = {0.05, 0.10, 0.15, 0.20, 0.25, 0.30, 0.35,
                            0.40, 0.45, 0.50, 0.55, 0.60, 0.65, 0.70,
                            0.75, 0.80, 0.85, 0.90, 0.95, 1.00 } ;
     int                   ilevel, ntrimax ;
     double                atol, rtol, p ;
     double                temp_atol = 1.0e-5, temp_rtol = 0.0 ;

     ilevel = 4;      /* level of initial mesh refinement */
     ntrimax = 10000; /* max number of triangles  */
     atol = 2.5e-2;   /* absolute and relative space tolerances */
     rtol = 0.0 ;
     p    = 1.0e-4;   /* diffusion parameter passed as users data */
     my_struct.p = p ;
     S2D_initialise( &my_integ ) ;
     S2D_time_dependent( &my_integ, 1, ntrimax, 0.0, n_t_out, t_out ) ;
     S2D_temporal_tol( &my_integ, S2D_Scalar_TOL, &temp_atol,
               &temp_rtol ) ;
     S2D_spatial_tol( &my_integ, S2D_Scalar_TOL, &atol, &rtol ) ;
     /* Use finite vol scheme -- pass across names of routines. */
     S2D_FVM_initialise( &my_integ ) ;
     S2D_FVM_initial_conditions( &my_integ, burgers_ic ) ;
     S2D_FVM_boundary_conditions( &my_integ, burgers_bc ) ;
     S2D_FVM_riemann_solver( &my_integ, burgers_rs ) ;
     S2D_FVM_diffusive_flux( &my_integ, burgers_g ) ;
     /* Use the trifit mesh generator */
     S2D_trifit_mesh_generator( &my_integ, "burgers.dmn", ilevel ) ;
     S2D_diagnostics_on( &my_integ ) ;
     /* Use the theta method integrator. */
     S2D_STHMID_initialise( &my_integ, S2D_STHMID_Theta, 4,
             S2D_STHMID_FI, S2D_STHMID_NoSwitch, 0.55 ) ;
     /* Use the watsit linear algebra. */
```

```
    S2D_SPWATSIT_initialise( &my_integ, 10.0, 10.0 ) ;
    /* Monitor routine. */
    S2D_monitor( &my_integ, monitor ) ;
    /* Have my data passed back by SPRINT2D. */
    S2D_users_data( &my_integ, (void *) &my_struct ) ;
    S2D_FVM_derivative_estimate( &my_integ, S2D_FVM_First_Deriv ) ;
    /* Integrate! */
    S2D_integrate( &my_integ ) ;
} /* main */
```

B. RIEMANN SOLVER CODE FOR BURGERS EQUATION PROBLEM

The routine from the driving program for the simple averaging Riemann solver is given below; its use results in negative solution values close to the wave front.

```
void problem_rs(TRIAD_Line *line, int npde, double x,
                double y, double t, int sub_name,
                double norm_x, double norm_y, double u_l[],
                double u_r[], void *users_data, double nf[])
  { /*  Burgers eqn: simple averaging Riemann solver */
    double u, f_x, f_y;
    u = ( u_l[0] + u_r[0] ) / 2.0 ;
    f_x = 0.5*u*u;
    f_y = 0.5*u*u;
    /* form the normal component of the flux */
    nf[0] = f_x * norm_x + f_y * norm_y;
  }
```

The negative values vanish when Roe's Riemann solver, see e.g. [Pennington and Berzins 1994] is implemented by replacing the assignment to `u` with :

```
    /* Burgers eqn: Roe Riemann solver */
    if u > 0.0 u = u_l[0];
    else       u = u_r[0];
```

C. PARTIAL DOMAIN FILE FOR ELECTROSTATIC PROBLEM

```
DOMAIN concave_ehd
VERTICES
 1      -2.0000e+00       1.5880e+00
 2      -1.9000e+00       1.4237e+00
 3      -1.8000e+00       1.2713e+00
 4      -1.7000e+00       1.1310e+00
 5      -1.6000e+00       1.0032e+00
 6      -1.5000e+00       8.8809e-01
 7      -1.4000e+00       7.8600e-01
 8      -1.3000e+00       6.9726e-01
 9      -1.2000e+00       6.2214e-01
10      -1.1000e+00       5.6080e-01
11      -1.0000e+00       5.1312e-01
```

```
12      -9.0000e-01       4.7842e-01
13      -8.0000e-01       4.5518e-01
14      -7.0000e-01       4.4089e-01
15      -6.0000e-01       4.3256e-01
16      -5.0000e-01       4.2765e-01
17      -4.0000e-01       4.2459e-01
18      -3.0000e-01       4.2259e-01
19      -2.0000e-01       4.2127e-01
20      -1.0000e-01       4.2043e-01
21      0.0000e+00       4.1997e-01
22      1.0000e-01       4.1984e-01
23      2.0000e-01       4.1998e-01
24      3.0000e-01       4.2036e-01
25      4.0000e-01       4.2091e-01
26      5.0000e-01       4.2129e-01
27      6.0000e-01       4.1613e-01
28      7.0000e-01       3.8382e-01
29      8.0000e-01       3.5758e-01
30      9.0000e-01       3.5920e-01
31      1.0000e+00       3.8476e-01
32      1.1000e+00       4.3100e-01
33      1.2000e+00       4.9537e-01
34      1.2000e+00       0.0000e+00
35      -2.0000e+00       0.0000e+00
```

Following this list of points comes a list of edges and then a list of how the edges form 2D domains.

D. DRIVING PROGRAM FOR ELECTROSTATIC PROBLEM

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "S2D.h"
#include "S2D_FVM_finite_volume_discretisation.h"
#include "S2D_BBOX_black_box_soln_strategy.h"
#include "newgrad.h"

/* Monitor routine - not included for the sake of brevity . */

void electro_ic( TRIAD_Triangle *tri, int npde, double x, double y,
         double t, int sub_name, void *users_data, double *u )
{ /* Initial conditions. */
     u[0] = 0.0 ;
}

void electro_bc( TRIAD_Line *line, int npde, double x, double y,
         double time, int edge_name, int sub_name, double norm_x,
```

```
        double norm_y, double *u, void *users_data,
        S2D_FVM_BC_Type *type, double *ub, double *dubdn,
        double *fb )
{ /* Boundary conditions. */
     dubdn[0] = 0.0 ;
     type[0] = S2D_FVM_Dirichlet ;
     if (edge_name == 2998) {
             ub[0] = 0.0 ;                /* Bottom edge. */
     } else if (edge_name == 2999) {
             ub[0] = y / 1.7307e+0 ;      /* Left hand edge. */
     } else if (edge_name == 2997) {
             ub[0] = y / 1.6129e-1 ;      /* Right hand edge. */
     } else {
             ub[0] = 1.0 ;                /* Top edge. */
     }
}

void electro_g( TRIAD_Line *line, int npde, double x, double y,
          double t, int sub_name, double norm_x, double norm_y,
          double *u, double *dudx, double *dudy, void *users_data,
          double *g_x, double *g_y )
{ /* Diffusion function. */
     g_x[0] = dudx[0] ;
     g_y[0] = dudy[0] ;
}

void electro_rs( TRIAD_Line *line, int npde, double x, double y,
          double t, int sub_name, double norm_x, double norm_y,
          double *u_l, double *u_r, void *users_data, double *nf )
{ /* Riemann solver -- zero convective flux in this problem. */
     nf[0] = 0.0 ;
}

void main( int argc, char *argv[] )
{ /* Driving program. */
     S2D_Intgrtn_Obj_Type    my_integ ;
     int                     itri = 2000,  ntrimax = 10000 ;
     double                  atol, rtol ;
     atol = 1.0e-4 ; rtol = 0.0 ;
     S2D_initialise( &my_integ ) ;
     S2D_steady( &my_integ, 1, ntrimax ) ;
     S2D_spatial_tol( &my_integ, S2D_Scalar_TOL, &atol, &rtol ) ;
     /* Use finite vol scheme - pass across names of routines. */
     S2D_FVM_initialise( &my_integ ) ;
     S2D_FVM_initial_conditions( &my_integ, electro_ic ) ;
     S2D_FVM_boundary_conditions( &my_integ, electro_bc ) ;
     S2D_FVM_riemann_solver( &my_integ, electro_rs ) ;
```

```
    S2D_FVM_diffusive_flux( &my_integ, electro_g ) ;
    /* Use GEOMPACK mesh generator. */
    S2D_geompack_mesh_generator( &my_integ, "el.dmn", itri, 0 ) ;
    /* Use the black box solution strategy. */
    S2D_BBOX_initialise( &my_integ, -1.0 ) ;
    /* Monitor routine. */
    S2D_monitor( &my_integ, monitor ) ;
    /* Integrate! */
    S2D_integrate( &my_integ ) ;
} /* main */
```

REFERENCES

Adjerid, S., Flaherty, J. E., Moore, P. K., and Wang, Y. J. 1992. High-order adaptive methods for parabolic equations. In *Experimental Mathematics: Computational Issues in Non-Linear Science, Physics D, 60, 1-4*, J.M. Hyman, Ed. North-Holland, 94–111.

Allgower, E. L. and Georg, K. 1990. *Numerical Continuation Methods*. Springer-Verlag.

Bank, R. E. 1995. *PLTMG: A Software Package for Solving Elliptic Partial Differential Equations. Users' Guide 7.0*. SIAM, Philadelphia.

Berzins, M. 1995. Temporal error control for convection-dominated equations in two space dimensions. *SIAM J. Sci. Comput. 16*, 3, 558–580.

Berzins, M., Dew, P. M., and Furzeland, R. M. 1989. Developing software for time-dependent problems using the method of lines and differential algebraic integrators. *Appl. Numer. Math. 5*, 375–397.

Berzins, M. and Furzeland, R. M. 1992. An adaptive theta method for the solution of stiff and non-stiff differential equations. *Appl. Numer. Math. 9*, 1–19.

Berzins, M. and Ware, J. M. 1994. Towards an automated finite element solver for time-dependent fluid-flow problems. In *MAFELAP93–Highlights*, J.R. Whiteman, Ed. John Wiley, 299–306.

Berzins, M. and Ware, J. M. 1995. Positive discretization methods for hyperbolic equations on irregular meshes. *Appl. Numer. Math. 16*, 417–438.

Berzins, M. and Ware, J. M. 1996. Solving convection and convection reaction problems using the M.O.L. *Appl. Numer. Math. 20*, 83–99.

Berzins, M., Pennington, S. V., Pratt, P. R., and Ware, J. M. 1997. SPRINT2D software for convection dominated PDEs, In *Modern Software Tools in Scientific Computing*, E. Arge, A.M. Bruaset and H.P. Langtangen, Eds., Birkhauser, 63–80.

Fairlie, R., Berzins, M. and Ware, J. M. 1997. Unstructured mesh calculation of compressible 2D steady jets. Submitted to *J. Comp. Phys.*

Falle, S. A. E. G. 1991. Self-similar jets. *Mon. Not. R. Astr. Soc. 250*, 581–596.

Furzeland, R. M, Rem, P. C., and Van der Wijngaart, R. F. 1989. General purpose software for multi-dimensional partial differential equations. *Technical Report, Shell Research Amsterdam.*

Jackson K. R. and Seward, W. L. 1993. Adaptive linear equations solvers in codes for large stiff systems of o.d.e.s. *SIAM J. Sci. Comp. 14*, 800–823.

Joe, B. 1991. GEOMPACK - a software package for the generation of meshes using geometric algorithms. *Adv. Eng. Soft. 13*, 5/6, 325–331.

Nurgat, E. M., Berzins, M., and Scales, L. E. 1997. Solving EHL problems using iterative multigrid and homotopy methods. To appear in *ASME Journal of Tribology.*

Pennington, S. V. and Berzins, M. 1994. New NAG library software for first-order partial differential equations. *ACM Trans. Math. Softw. 20*, 1, 63–99.

Pratt, P. R. and Berzins, M. 1996. Shock preserving quadratic interpolation for visualisation on triangular meshes. *Comput. and Graphics 20*, 5, 723–730.

Scales, L. E. 1993. NAESOL: User's Guide. *Internal report, Shell Research Ltd, Chester.*

Sleigh, P. A., Berzins, M., Gaskell, P. H., and Wright, N. G. 1997. An unstructured finite volume algorithm for predicting flow in rivers and estuaries. To appear in *Computers and Fluids*.

Suresh, A. and Liou, M-S. 1991. Osher's scheme for real gases. *AIAA Jour. 29*, 4, 920–926.

Tomlin, A., Berzins, M., Ware, J. M., Smith, J., and Pilling, M. 1997. On the use of adaptive gridding methods for modelling chemical transport from multi-scale sources. *Atmospheric Env. 31*, 18, 2945–2959.

Venkatakrishnan, V. 1995. Convergence to steady state of the Euler equations on unstructured grids with limiters. *J. Comp. Phys. 118*, 120–130.

Ware, J. M. 1993. *The Adaptive Solution of Time-Dependent Partial Differential Equations in Two Space Dimensions*, PhD thesis, Computer Studies, University of Leeds.