TTGR - A Package for Solving Partial Differential Equations in Two Space Variables.

L. Kaufman

N. L. Schryer

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

A formulation is presented for partial differential equations in two space variables which facilitates their numerical solution. An algorithm taking full advantage of this formulation is briefly outlined.

An implementation of the algorithm in portable Fortran, called TTGR (Transient Tensor Galerkin for partial differential equations on Rectangles), is described. The package is especially easy to use since only the spatial mesh and the accuracy desired in the solution of the equations in time need to be specified. The time evolution is then automatically carried out to achieve the desired accuracy. A user's guide to TTGR is given along with many examples.

The examples are available through electronic mail.

There are 6 examples and the **fortran** for each is available in single or double precision. For example, the command

```
mail research!netlib
send only ttgrx1 from port
send only dttgrx6 from port
.
```

will cause you to receive in the mail the first example in single and the sixth example in double precision.

June 5, 1985

TTGR - A Package for Solving Partial Differential Equations in Two Space Variables.

L. Kaufman

N. L. Schryer

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction.

Many physical problems require the solution of partial differential equations (**pde**'s) in two space variables. Typically these equations are sufficiently complex that their solution must be carried out numerically.

This paper describes a formulation for solving systems of **pde**'s in two spatial variables, on a rectangle, and time. The formulation allows for terms of the form $\mathbf{u}$, $\mathbf{u}_x$, $\mathbf{u}_y$, $\mathbf{u}_t$, $\mathbf{u}_{xt}$, $\mathbf{u}_{yt}$, $\mathbf{u}_{xx}$, $\mathbf{u}_{xxt}$, $\mathbf{u}_{xy}$, $\mathbf{u}_{xyt}$, $\mathbf{u}_{yy}$, $\mathbf{u}_{yyt}$ in the **pde**'s, and $\mathbf{u}$, $\mathbf{u}_x$, $\mathbf{u}_y$, $\mathbf{u}_t$, $\mathbf{u}_{xt}$, $\mathbf{u}_{yt}$ in the boundary conditions (**bc**'s), where $\mathbf{u}$ is a vector of **pde** variables, and $\mathbf{u}_x$ denotes $\partial \mathbf{u} / \partial x$, etc. The mathematical formulation is given in section 2.

Getting Started.

If you have not read this screed before, want to solve a simple **pde** and have no interest in fancy things having nothing to do with your immediate needs, just do the following

- Read section 2, Statement of the Problem, p 3.
- Read section 4, Formulation, Example 1, p 6.
- Skim section 5, Software, pp 11-16.
- Read Appendix 4, Programs, Example 1, pp 1-9.
- Copy the example program (either from a file or the paper).
- Run it and make sure it works as advertised.
- Alter it to solve your problem; see the start of Appendix 4 for the steps involved here.

With luck, your problem is now solved. The above scheme typically involves changing only a couple of dozen lines of code in the example to get a problem solved.

The Examples

If the above "Getting Started" reading does not seem to address your problem, there are many examples discussed in section 4. One of them is quite likely to be of help. The examples are

- A simple **pde**, the heat equation, see Example 1, p-6.
- A coupled system of **pde**s, see Example 2, p 6.
- A material interface, see Example 3, p 7
- A non-rectangular domain, see Example 4, p 8
- A static problem, see Example 5, pp 8-9
- Error estimation, see Example 6, pp 9-10

A Principle.

The guiding principle used during the design of TTGR was

> It is better a user complain the package runs slowly than complain the package cannot solve the problem at hand.

As a result, people wanting to solve various model equations in a hurry, so they can get on to other models and problems (that is, people whose time is more valuable than machine time), should find TTGR very useful. However, people wanting to solve the same problem many times in a production environment may find TTGR, with the default settings, slow for their needs; see section 6 for ways to speed TTGR up considerably.

The numerical solution technique employs Galerkin's method in space, using B-splines, and a variable order, variable time-step extrapolated backward difference procedure in time; see section 3 for an outline. Many examples are formulated in section 4. Section 5 is a user manual for the software called TTGR for Transient Tensor product Galerkin for partial differential equations on Rectangles. Section 6 describes ways of making TTGR run faster than the default settings allow and also describes alternative ways of entering and using the package.

Appendix 1 gives a brief tutorial on B-splines. Appendix 2 gives a brief tutorial on extrapolation. Appendix 3 discusses improvements that could be made in TTGR. Appendix 4 presents the programs used to solve the examples discussed in section 4.

Appendix 5 summarizes the basic procedures available in TTGR, along with their arguments, and gives a list of error states and problems that may arise when using TTGR, along with the common causes of such difficulties.

A Warning.

This software is in an infant state. Bugs may be anywhere, and there are oodles of corners in which they can lurk. The code has been created modularly, using the best tools at hand. The goal is to create a robust and widely applicable package for solving two-dimensional **pde**s. However, this modularity has hurt a bit. For example, the **ode** solver IODE from the Port Library has been used to solve the spatially discretized equations. Thus, the user can change the name of the **pde** defining subroutine, but not that for the **bc**s: IODE only has one subroutine name it can pass below it. This is a learning experience for that software as well. The spatial discretization scheme is very robust, but slow. The linear algebra is robust, but not necessarily optimal in run-time and space. However, to quote a user: "Expensive solutions are better than none at all." The bottom line here is: be careful with and mistrustful of this code. Please use it any way you see fit, report any bugs or anomalous behavior to us, and let us know generally how things go with it. Since we plan many improvements to TTGR, see Appendix 3, your comments are both welcome and likely to be effective. Our electronic mail addresses are

research!lck
research!nls

The examples are available through electronic mail.

There are 6 examples and the **fortran** for each is available in single or double precision. For example, the command

```
mail research!netlib
send only ttgrx1 from port
send only dttgrx6 from port
.
```

will cause you to receive in the mail the first example in single and the sixth example in double precision.

2. Statement of the Problem.

The general form of equations that can be handled by TTGR is an essentially classical, text-book divergence-form **pde** with a general set of boundary conditions (**bc**s).

The pde-bc Formulation.

The general **pde-bc** form that can be solved with the approach used in TTGR is given by the following equations, where **u** is a vector of **pde** variables of length n_u. Since all physical laws that are second order in space can be written in divergence form, the **pde**'s are assumed to be in semi-linear, divergence-form

$$\frac{\partial}{\partial x}\,\mathbf{a}^{(1)}(t,\,x,\,y,\,\mathbf{u},\,\mathbf{u}_x,\,\mathbf{u}_y,\,\mathbf{u}_t,\,\mathbf{u}_{xt},\,\mathbf{u}_{yt})\;+$$
$$\frac{\partial}{\partial y}\,\mathbf{a}^{(2)}(t,\,x,\,y,\,\mathbf{u},\,\mathbf{u}_x,\,\mathbf{u}_y,\,\mathbf{u}_t,\,\mathbf{u}_{xt},\,\mathbf{u}_{yt})\;= \tag{2.1}$$
$$\mathbf{f}(t,\,x,\,y,\,\mathbf{u},\,\mathbf{u}_x,\,\mathbf{u}_y,\,\mathbf{u}_t,\,\mathbf{u}_{xt},\,\mathbf{u}_{yt})\;,$$

where **a** and **f** are vector-valued functions of their arguments, for $L_x \le x \le R_x$ and $L_y \le y \le R_y$. It is required that the length of **a** and **f** be equal to n_u, the number of **pde** variables, that is, the length of the vector **u**. The boundary conditions are assumed to have the form

$$\mathbf{b}\;(\;t,\,x,\,y,\,\mathbf{u},\,\mathbf{u}_x,\,\mathbf{u}_y,\,\mathbf{u}_t,\,\mathbf{u}_{xt},\,\mathbf{u}_{yt}\;)\;=\;0 \tag{2.2}$$

where **b** is a vector-valued function, of length n_u, of its arguments. Any identically zero component of the **bc** vector **b** is treated as an inactive **bc**. If each of the **pde**'s is second order in space, then each of the **bc**'s will have to be active. If any of the **pde**'s are of order less than 2 in space, some of the **bc**'s must accordingly be inactive. Initial conditions (**ic**'s) $\mathbf{u}(0,x,y)$ must be supplied, but need not satisfy the **bc**'s (2.2).

A classical example of the above form (with $n_u = 1$) is the heat equation [22]

$$u_t = u_{xx} + u_{yy} \qquad \text{on} \quad [0,1] \times [0,1]$$

subject to boundary conditions

$$u(t,x,y) = g(x,y)$$

with initial conditions

$$u(0,x,y) = 0.$$

Note that these initial conditions do not satisfy the **bc**'s, unless $g(x,y) \equiv 0$. For this equation we have

$$a^{(1)} = u_x, \quad a^{(2)} = u_y \quad \text{and} \quad f = u_t$$

with **bc**s

$$\mathbf{b} = u - g.$$

Note that the form of (2.1)-(2.2) encompasses parabolic ($u_t = u_{xx} + u_{yy}$), elliptic ($u_{xx} + u_{yy} = 0$) and hyperbolic ($u_t = u_x + u_y$) problems. It also encompasses **pde**'s that have no solution, such as

$$u_x^2 + u_y^2 = -1$$

over the real field.

History.

Several other Fortran software packages are available for solving **pde**'s in two spatial variables, for example [28,29]. These packages assume that the **pde** has a form like

$$\mathbf{u}_t = \mathbf{f}(\;t,\,x,\,y,\,\mathbf{u},\,\mathbf{u}_x,\,\mathbf{u}_y,\,\mathbf{u}_{xx},\,\mathbf{u}_{yy},\,\mathbf{u}_{xy}\;) \tag{2.3}$$

with boundary conditions of the form

$$\mathbf{u} = \boldsymbol{\alpha}(t)$$

or

$$\boldsymbol{\alpha}(t,x,y,\mathbf{u}) + \boldsymbol{\beta}(t,x,y,\mathbf{u})\mathbf{u}_N = \boldsymbol{\gamma}(t,x,y,\mathbf{u}). \tag{2.4}$$

where $\mathbf{u}_N$ is the normal derivative. There are slight variations on this theme. For example, [29] allows no $\frac{\partial^2 \mathbf{u}}{\partial x\, \partial y}$ terms in the **pde** (2.3). Also, [28] allows no dependence of **u** in the coefficients of the **bc**s in (2.4), and only applies to *scalar* **pde**s. While formulation (2.3)-(2.4) covers a wide range of physically interesting problems, it does not cover problems with $\mathbf{u}_{xt}$ or $\mathbf{u}_{xxt}$ terms [32], nor does it deal with problems having tangential components in their **bc**s [34].

3. General Method of Solution.

Let the solution $\mathbf{u}(t,x,y)$, for a given instant in time, be approximated by a tensor-product of B-splines [27,1,2,3] of **order** k_x and k_y on **meshes** $X(1) \le \cdots \le X(NX)$, and $Y(1) \le \cdots \le Y(NY)$, see Appendix 1. That is, for any fixed y, each component of the solution will be approximated in x by a piecewise polynomial function of degree less than k_x, with $k_x - 2$ continuous derivatives, where $k_x \ge 2$ is any integer the user desires; a similar statement holds about the degree in y. Let $B_p(x)$ be the basis functions in x and $C_q(y)$ be those in y. Then

$$u_i(t,x,y) = \sum_p \sum_q U_{q,p,i}(t)\; B_p(x)\; C_q(y). \tag{3.1}$$

If we set $h_x = \max_i \mid X(i+1) - X(i) \mid$ and h_y similarly, then the error, $\| u_i - \hat{u}_i \|_\infty \equiv \max_{x,y} \mid u_i(t,x,y) - \hat{u}_i(t,x,y) \mid$ is $O(h_x^{k_x} + h_y^{k_y})$, for some B-spline $\hat{u}_i$, see [3]. Since k_x and k_y may be taken to be any integer ≥ 2, this gives a powerful technique for approximating the solution $\mathbf{u}(t,x,y)$ in space. We can use the Rayleigh-Ritz-Galerkin (R-R-G) method [31,25] to find essentially the projection of the solution of the **pde** onto the space of B-splines we have selected. This reduces the **pde**'s in space and time to **ode**'s in time [15,31] for the coefficients $U_{q,p,i}(t)$ in the expansion (3.1).

Thus, after the spatial discretization, only **ode**'s in time remain to be solved. Since these **ode**'s are known to be in general "stiff" [9,10], an implicit differencing scheme must be used to solve them. This virtually requires that the partial derivatives of the **a** and **f** in (2.1) and of the **b** in (2.2), with respect to their arguments be known, either analytically or numerically.

The next step is the solution of these time-varying **ode**'s. Here we assume that some basic one-step **ode** solver is available. For example, a backwards-Euler or Crank-Nicholson scheme [22], or an exponentially-fitted technique [18], or even an explicit method such as Gragg's modified mid-point rule [17,18], could be used. `TTGR` uses backwards-Euler as the default time discretization scheme. See [26] for a description of the method used to solve the nonlinear equations arising at each time-step.

All the above techniques, and many others, have the property that for a given time-step δ they produce an approximate solution accurate to $O(\delta^\gamma)$, where typically γ is 1 or 2. Moreover, if the equations are solved using time-steps of δ and $\delta/2$, the results of these two computations can be combined using extrapolation [5,17] to obtain a result accurate to $O(\delta^{2\gamma})$. This process can be repeated indefinitely, so a basic process of accuracy δ^γ can be used to generate a sequence of processes of accuracy $O(\delta^\gamma)$, $O(\delta^{2\gamma})$, $\cdots$, $O(\delta^{P\gamma})$, $\cdots$, see Appendix 2.

A step-size and order monitor is available [23,24] for carrying out this extrapolation process and *automatically* deciding what time-step δ and order $P\gamma$ should be used, given the accuracy desired in the solution. The user need only specify how accurately the solution in time should be computed, and the time integration then proceeds automatically, with no need for the user to worry about choosing δ.

The algorithm implemented by `TTGR` for solving such **pde**'s then consists of 3 steps:

1) Discretize the equations in space using R-R-G with B-splines.
2) Produce a one-step method for solving the resulting **ode**'s.
3) Feed that one-step process to the extrapolation step-size and order monitor.

Section 6 gives a somewhat more detailed outline of the spatial discretization process and the various parameters that describe the solution procedure.

4. Examples - Formulation.

This section gives the formulation of many examples in terms of (2.1)-(2.2). See Appendix 4 for programs that solve these problems using `TTGR`.

Example 1 - A Simple Heat Equation.

As a simple example of the use of `TTGR`, consider solving the scalar heat equation

$$u_t + u_x + u_y = \mathbf{u}_x + \mathbf{u}_{xx} + .1\, \mathbf{u}_{xy} + \mathbf{u}_y + \mathbf{u}_{yy} + .1\, \mathbf{u}_{xy} + g(t,x,y), \tag{4.1}$$

on the unit square $[0,1]\times[0,1]$, where the source term $g(t,x,y)$ is chosen so that the solution is a known function, $u(t,x) = t\cdot x\cdot y$. The boundary conditions are then taken to be

$$u(t,x,y) = t\cdot x\cdot y \tag{4.2}$$

with initial conditions

$$u(0,x,y) = 0\,. \tag{4.3}$$

The **pde** (4.1) is equivalent to (2.1) with

$$a^{(1)} = u + u_x + .1\, u_y,$$

$$a^{(2)} = u + u_y + .1\, u_x,$$

$$f = u_t + u_x + u_y - g(t,x,y)$$

while the **bc**s (4.2) are equivalent to (2.2) with

$$\mathbf{b} = u(t,x,y) - t\,x\,y.$$

See example 1 in Appendix 4 for code solving this problem.

Example 2 - Two Heat Equations.

Consider solving the coupled system of heat equations

$$\begin{aligned} u_{1t} &= u_{1xx} + u_{1yy} - u_1\, u_2 + g_1 \\ u_{2t} &= u_{2xx} + u_{2yy} - u_1\, u_2 + g_2 \end{aligned} \tag{4.4}$$

on the unit square $[0,1]\times[0,1]$, where g_1 and g_2 are chosen so that the solution is given by

$$u_1(t,x,y) = e^{t\,(x-y)} \qquad \text{and} \qquad u_2(t,x,y) = e^{-t(x-y)}.$$

The boundary conditions are then taken to be

$$u_1(t,x,y) = e^{t(x-y)} \qquad \text{and} \qquad u_2(t,x,y) = e^{-t(x-y)} \tag{4.5}$$

with initial conditions

$$u_1(0,x,y) = 1 = u_2(0,x,y). \tag{4.6}$$

The **pde** (4.4) is equivalent to (2.1) with

$$a_1^{(1)} = u_{1x}, \quad a_2^{(1)} = u_{2x},$$

$$a_1^{(2)} = u_{1y}, \quad a_2^{(2)} = u_{2y},$$

$$f_1 = u_{1t} + u_1\, u_2 - g_1, \quad f_2 = u_{2t} + u_1\, u_2 - g_2$$

while the **bc**s (4.5) are equivalent to (2.2) with

$$b_1 = u_1(t,x,y) - e^{t(x-y)} \qquad \text{and} \qquad b_2 = u_2(t,x,y) - e^{-t(x-y)}.$$

See example 2 in Appendix 4 for code solving this problem.

More Meaty Examples

There are many excellent schemes for spatially discretizing (2.1)-(2.2), reducing them to a system of **ode**'s, and thus solving the problem. The most elegant and most acceptable to people in the physical sciences is Galerkin's method [25,31]. That method has the property that the term $\mathbf{a}(\cdot)$ in (2.1) is forced to be continuous everywhere, see [25]. Since $\mathbf{a}(\cdot)$ is a flux, physically, and physicists expect fluxes to be continuous, this means that Galerkin's method gives the solution the physicists want. Other spatial discretization methods such as finite-differences, least-squares and collocation do not automatically make $\mathbf{a}(\cdot)$ continuous. Most of the rest of the examples in this section are stripped down versions of real problems and the use of Galerkin's method in `TTGR` is important in their formulation.

Example 3 - Interfaces

This example shows how to deal with material interfaces. Assume a layered structure with three rectangles piled one on another, each with its own material constant, $\kappa\,(x,y)$. For a heat flow problem, the modeling equations might look like

$$\mathbf{u}_t = \nabla \cdot (\ \kappa\,(x,y)\ \nabla\, \mathbf{u}\ ) + g(t,x,y) \tag{4.7}$$

on the domain $[0,1]\times[0,3]$, where κ is piecewise constant on the different rectangles, say,

$$\kappa \equiv \begin{cases} 1 0\le y\le 1 \\ 1/2 1<y\le 2 \\ 1/3 2<y\le 3 \end{cases}$$

and g is chosen so that

$$u \equiv \begin{cases} ty 0\le y\le 1 \\ t\,(2y-1) 1<y\le 2 \\ 3t\,(y-1) 2<y\le 3. \end{cases}$$

The **bc**s on the bottom and top are given by, say, insulation

$$\mathbf{u}_N = 0 \tag{4.8a}$$

where $\mathbf{u}_N$ is the normal derivative, and on the sides Dirichlet data

$$\mathbf{u} = s(t,x,y) \tag{4.8b}$$

is used, for some known s.

The **pde** is equivalent to (2.1) with

$$a^{(1)} = \kappa\,(x,y)\ \mathbf{u}_x$$
$$a^{(2)} = \kappa\,(x,y)\ \mathbf{u}_y$$
$$f = \mathbf{u}_t - g.$$

The bottom and top **bc**s are equivalent to (2.2) with

$$\mathbf{b} = u_y$$

and those on the side are equivalent to

$$\mathbf{b} = \mathbf{u} - s(t,x,y).$$

The sporting aspect of this problem is that the normal component of $\kappa\ \nabla\ \mathbf{u}$, that is $\kappa\ u_y$, across each material interface is expected to be continuous by any physical scientist. Galerkin's method in fact forces this to be the case. See Example 3 of Appendix 4 for code solving this problem.

Example 4 - A Non-Rectangular Domain.

Suppose you wanted to solve the **pde**

$$\mathbf{u}_t = \frac{\partial}{\partial x} (u_x - \frac{u_y}{10}) + \frac{\partial}{\partial y} (u_y - \frac{u_x}{10}) + g \tag{4.9}$$

on the domain

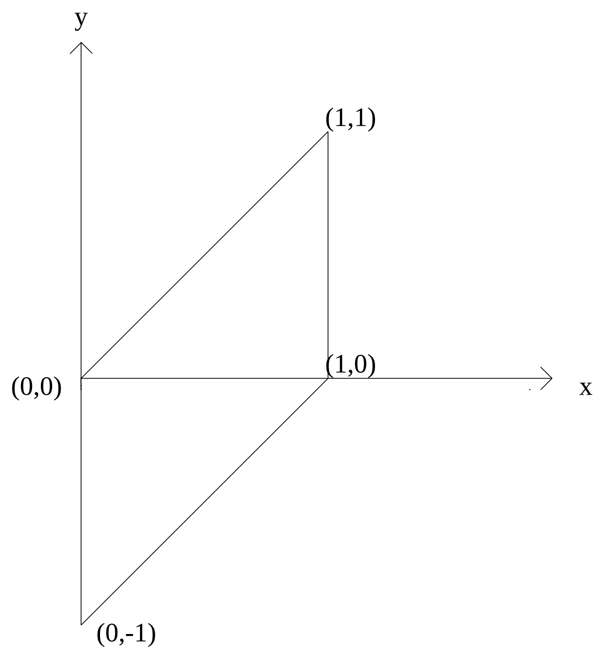


Figure 1

where g is chosen so that $u \equiv t\,x\,y$. We prescribe consistent Neumann data on the top and bottom, and consistent Dirichlet data on the left and right hand sides.

How do you solve this? Well, it is easy to map that domain onto a rectangle, namely $[0,1]\times[0,1]$. Just map

$$\begin{aligned} x &= \xi \\ y &= -1+\xi + \eta \end{aligned} \tag{4.10}$$

and the equation may be solved on the new rectangular ξ,η domain $[0,1]^2$. That is, $(x, y) (\xi, \eta) \equiv (x (\xi, \eta), y (\xi, \eta))$ maps $[0,1] \times [0,1]$ into Figure 1. Then, TTGR is told to solve for

$$\mathbf{w}(t, \xi, \eta) \equiv \mathbf{u} (t, (x,y)(\xi, \eta , t))$$

on the domain $[0,1]^2$, with the **pde** and **bc**s mapped into that domain as well. See Example 4 of Appendix 4 for code solving this problem.

Example 5 - A Static Problem

This example shows how to solve a static **pde** and also shows that using a non-uniform grid can be very useful and effective. Let the **pde** be Laplace's equation

$$u_{xx} + u_{yy} = 0 \tag{4.11}$$

on the domain $[0,1]\times[0,1]$. Since the Real part of any analytic function solves Laplace's equation, let's pick u to be the Real part of $z \log (z)$, where $z = x + i\,y$. We choose Dirichlet **bc**s on the left, right and top of the domain, consistent with that choice for u. For the bottom we choose Neumann data $u_y = 0$. This gives

$$
\begin{aligned}
u_y(x,0) &= 0 \\
u(1,y) &= Real(z\ log(z)) \\
u(x,1) &= Real(z\ log(z)) \\
u(0,y) &= \frac{-\pi\, y}{2}
\end{aligned}
\tag{4.12}
$$

Note that even though the coefficients of (4.11) and (4.12) are analytic, the solution of this static problem has singular first partials at $z = 0$.

The **pde** is equivalent to (2.1) with

$$
\begin{aligned}
a^{(1)} &= \mathbf{u}_x \\
a^{(2)} &= \mathbf{u}_y \\
f &= 0
\end{aligned}
$$

The boundary conditions are equivalent to (2.2) with

$$
\mathbf{b} = \begin{cases}
u_y(x,0) \\
u(1,y) - Real(z\ log(z)) \\
u(x,1) - Real(z\ log(z)) \\
u(0,y) + \dfrac{\pi\, y}{2}
\end{cases}
$$

The fact that there is no $\mathbf{u}_t$ term in f is ok, since `TTGR` does not require it.

The logarithmic singularity will result in slow convergence unless a non-uniform grid is used. See example 5 in Appendix 4 for code solving this problem.

Example 6 - Estimating the Error in the Computed Solution.

Picking a spatial mesh and an accuracy requirement for the time evolution of a **pde** guarantees absolutely **nothing** about the accuracy of the computed solution. Without at least a little preliminary error estimation, the computed results may look nice and be garbage. The error in a computed solution can be easily estimated however.

There are two sources of error in the computed solution of (2.1)-(2.2). The first is the error due to the spatial discretization and the second is due to the time discretization. Crudely speaking, the error in a solution $\mathbf{u}$ is of the form

$$
O\,(\,h^k\,) + O\,(\,e_1\ \|\mathbf{u}\|_\infty + e_2\,)
\tag{4.13}
$$

where h is the spatial mesh spacing and e_1 and e_2 are user settable numbers controlling the time evolution error. To make sure that $\mathbf{u}$ is as accurate as we want, some testing of the size of the two components of (4.13) is in order.

Assume that the solution $\mathbf{u}$ has been obtained on the mesh $\mathbf{x}\times\mathbf{y}$ using $\mathbf{e}$ for the error tolerances. Then let $\mathbf{u}_h$ be the solution of the same problem but on a mesh $\mathbf{x}_h\times\mathbf{y}_h$ whose mesh spacing is half that of $\mathbf{x}\times\mathbf{y}$. Also, let $\mathbf{u}_e$ be the solution of the same problem on the mesh $\mathbf{x}\times\mathbf{y}$ but using $\mathbf{e}/10$ for the error controls. We can then estimate the $O(\,h^k\,)$ error term in (4.13) as

$$
\|\ \mathbf{u} - \mathbf{u}_h\ \|_\infty ,
$$

because $\mathbf{u}_h$ is so much more accurate than $\mathbf{u}$ that it can be regarded as the true solution of the problem.

Similarly, we estimate the $O(\,e_1 \|\ \mathbf{u}\ \|_\infty + e_2\,)$ term in (4.13) as

$$
\|\ \mathbf{u} - \mathbf{u}_e\ \|_\infty ,
$$

because, again, $\mathbf{u}_e$ is so much more accurate than $\mathbf{u}$ that it can be regarded as the true solution of the problem.

We can assemble the above two estimates for the terms in (4.13) to obtain

$$\| \mathbf{u} - \mathbf{u}_{true} \|_\infty \le \| \mathbf{u} - \mathbf{u}_h \|_\infty + \| \mathbf{u} - \mathbf{u}_e \|_\infty \qquad (4.14)$$

The estimate (4.14) is easily computable in practice since everything on the right hand side is known.

The first term is easily evaluated, see Appendix 4. The second term is a little trickier to estimate, but it has a pretty result. We have, using (3.1),

$$\| \mathbf{u} - \mathbf{u}_e \|_\infty \equiv \| \sum_{qp} (U_{qp} - U_{eqp}) B_p C_q \|_\infty \le \sum_{qp} | U_{qp} - U_{eqp} | B_p C_q \le$$

$$\sum_{qp} \| \mathbf{U} - \mathbf{U}_e \|_\infty B_p C_q = \| \mathbf{U} - \mathbf{U}_e \|_\infty \sum_{qp} B_p C_q = \| \mathbf{U} - \mathbf{U}_e \|_\infty .$$

This gives the estimate

$$\| \mathbf{u} - \mathbf{u}_e \|_\infty \le \| \mathbf{U} - \mathbf{U}_e \|_\infty . \qquad (4.15)$$

See example 6 in Appendix 4 for code implementing the above estimation scheme.

5. Software for the pde-bc Problem.

This section is a brief user's manual for a software package called `TTGR`, (Transient Tensor Galerkin method for **pde**s on Rectangles), implementing the algorithm outlined in section 3.

Before invoking `TTGR` the user must

- Make B-spline meshes for x and y.
- Make initial conditions for the B-spline coefficients **U** in (3.1).
- Write subroutines
  - `AF` - to evaluate **a** and **f** in (2.1).
  - `BC` - to evaluate **b** in (2.2).
  - `HANDLE` - to output (print) the solution results.

Each of these preparatory steps are illustrated in Appendix 4 and will not be described here.

The outer layer of the `TTGR` package is called `TTGR` and is invoked by

```
Call TTGR (U,nu,kx,X,nx, ky,Y,ny,
           tstart,tstop,dt,
           AF,BC,
           errpar,
           HANDLE)
```

The input to `TTGR` is

`U`	-	The B-spline coefficients (3.1) for the initial values of the **pde** variables **u**. `U`(i,j,l), for $i=1, \cdots,$ `nx-kx`, $j=1, \cdots,$ `ny-ky`, are the coefficients for u_l, $l=1, \cdots,$ `nu`. See below for ways to get the initial conditions, `TSL2W`.
`nu`	-	The number n_u of **pde** variables **u**.
`kx`	-	The B-spline order to be used in x. `kx` ≥ 2 is necessary.
`X`	-	The B-spline mesh to be used in x. The multiplicity of `X`(1) and `X(nx)` must be `kx`, see Appendix 1. The Port Library routines for making uniform meshes, `UMB` and `LUMB`, guarantee the first and last mesh points have multiplicity `kx`.
`nx`	-	The length of the mesh array `X`.
`ky`	-	The B-spline order to be used in y. `ky` ≥ 2 is necessary.
`Y`	-	The B-spline mesh to be used in y. The multiplicity of `Y`(1) and `Y(ny)` must be `ky`, see Appendix 1. The Port Library routines for making uniform meshes, `UMB` and `LUMB`, guarantee the first and last mesh points have multiplicity `ky`.

ny - The length of the mesh array Y.

tstart - Start integration at time tstart.

tstop - Stop integration at time tstop. tstop should be a variable, *not* a constant, in the program calling TTGR; see output description below.

dt - The initial choice for the time-step. The performance of TTGR is substantially independent of the initial value of dt chosen. It is sufficient that dt be within several orders of magnitude of being "correct." The value of dt will automatically be adjusted by TTGR to obtain the solution to the desired accuracy at the least possible cost. Thus, dt should be a variable, *not* a constant, in the user's calling program.

AF - A subroutine for specifying the **a** and **f** terms in the **pde** (2.1). AF must be declared External in the user's calling program. This user-supplied subprogram will be described later.

BC - A subroutine for specifying the boundary conditions **b** in (2.2). BC must be declared External in the user's calling program. This user-supplied subprogram will be described later. If there are no **bc**s then the dummy subroutine TTGRP may be used in place of BC.

errpar - A Real vector of length 2 for determining the error desired (to be allowed) in the solution of the equations in time. The two components govern, roughly, the relative and absolute error in the computed solution. For the i^{th} component of the **pde** solution **u**, the error at each time-step in the time integration will be at most

$$\mathtt{errpar}(1) * \|u_i\|_\infty + \mathtt{errpar}(2)$$

Thus, errpar(1)=0 gives the solution accurate to an absolute error of errpar(2), and errpar(2)=0 gives the solution accurate to a relative error of errpar(1). The choice of errpar(1) and errpar(2) is highly problem dependent. A sound technique is the following: If the scale of the problem is such that *S* is the smallest value for which a prescribed relative error tolerance is desired, then the choice

$$\mathtt{errpar}(1) = 10^{-2} \quad ; \qquad \mathtt{errpar}(2) = 10^{-2} \times S$$

will essentially give 1% relative accuracy in all values down to around S in size. Values below S will have absolute error smaller than $10^{-2} \times$ S. Users should be very careful to avoid setting errpar(2) = 0 when the solution is zero at any point in either space or time, or the integration will die for the obvious reasons.

HANDLE - A user-supplied subroutine that will be called by TTGR at the end of each time-step. HANDLE must be declared External in the user's calling program. This subprogram will be described later. If no output is desired, then the dummy subroutine TTGRH may be used in place of HANDLE.

The output from TTGR is

U - The B-spline coefficients for the **pde** solution **u** at time tstop.

tstop - The time at which integration stopped. tstop may be altered by the user-supplied subroutine HANDLE. If an error state exists on return (see Appendix 5), tstop is set to the last instant in time when the solution was known accurately. Thus, tstop should be a variable, *not* a constant, in the user's call to TTGR.

dt - The final value of the "optimal" time-step.

Static Problems

For static problems, where t, u_t, u_{xt} and u_{yt} do not appear in the **pde**, tstart, tstop and dt must be chosen consistently but they may be otherwise arbitrary. For example,

```
tstart = 0
tstop  = 1
dt     = tstop
```

is a fine choice.

Choosing the initial `dt` to go less than all the way to the final time will waste run-time by solving the static problem once on the first time-step and then solving it again and again until the final time is reached. `TTGR` raises the time-step rapidly when solving a static problem, but it is wasteful to solve a problem more than once.

A "restart" in `HANDLE`, see below, for a static problem is a disaster − the user should **STOP** right there. `TTGR` thinks that by lowering `dt` it can make the problem easier, a correct assumption if the problem is transient, but lowering the time-step for a static problem changes nothing. The difficulty is that Newton's method cannot converge from the initial conditions, or the initial guess in this case.

Scratch Space Used.

The amount of scratch space used on the dynamic stack of the Port Library [14] is, neglecting lower order terms, when the default setting are used,

$$n_u \; n_x \; n_y \; (\; 3 \; H \; - \; 1)$$

Real words (storage units), where $H \equiv n_u \; (\; k_x \; + \; (\; n_x \; - \; k_x \;) \; (k_y \; - \; 1 \;) \;)$ is the half-band-width of the Jacobian, n_u is the number of **pde**s, n_x is the number of points in the spatial mesh for *x*, k_x is the B-spline order for the mesh *x*, n_y is the number of points in the spatial mesh for *y* and k_y is the B-spline order for the mesh in *y*. For sufficiently large values of n_x and n_y, the storage is roughly $3 \; (\; k_y \; - \; 1 \;) \; n_u^2 \; n_x^2 \; n_y$. See section 6 for ways of causing `TTGR` to use much less space.

All scratch space for `TTGR` is taken from the stack. Solving **pde**s is a non-trivial process, requiring substantial work space. For virtually all problems solved by `TTGR` the user will have to declare and initialize the PORT stack to a size larger than the default size of 1000 Real words. This process will be illustrated in the first example of Appendix 4.

Run-time.

The run-time of `TTGR` is proportional to $n_u \; n_x \; n_y \; (\; H \; - \; 1 \;)^2$ for the default settings. See section 6 for ways to make `TTGR` run much faster.

The storage and run-time of `TTGR` are far from optimal with the default settings, being on the order of n^3 and n^4, respectively, for an *n* by *n* grid. Optimal space and time would be $O(\; n^2 \;)$. The reasons for the default use of a banded, pivoting matrix solver are detailed in section 6. That section also shows how to make the package run much faster, albeit on a smaller class of problems (parabolic), and use much less space.

Double Precision Version.

The Double Precision version of `TTGR` is called `DTTGR`. The calling sequence for `DTTGR` is precisely the same as that for `TTGR`, with *all* floating-point arguments Double Precision, *except* `errpar`, which remains Real. The amount of scratch space used by `DTTGR` on the dynamic stack of the Port Library [14] is, neglecting lower order terms,

$$n_u \; n_x \; n_y \; (\; 3 \; H \; - \; 1)$$

Double Precision words (storage units).

`AF` and `BC` Descriptions.

The user-supplied subroutines `AF` and `BC`, which define the **pde** -**bc** problem to be solved, are now described. When `TTGR` needs to compute **a** and **f**, it will

```
Call AF(t,Xe,Ye,nxe,nye,Nu, U,Ut,Ux,Uy,Uyt,Uxt,
        A,AU,AUt,AUx,AUy,AUxt,AUyt,
        F,FU,FUt,FUx,FUy,FUxt,FUyt)
```

Before `TTGR` calls `AF`, it sets to **0** the 14 arrays `A` through `FUyt` and provides the *input* values

- `t` - The current value of time.
- `Xe` - A list of points x where **a** and **f** are to be evaluated. This `Xe` is *not* the B-spline mesh X. The points `Xe` at which **a** and **f** are desired are determined by the quadrature rule used by `TTGR` to implement Galerkin's method.
- `nxe` - The length of Xe.
- `Ye` - A list of points y where **a** and **f** are to be evaluated. This `Ye` is *not* the B-spline mesh Y. The points `Ye` at which **a** and **f** are desired are determined by the quadrature rule used by `TTGR` to implement Galerkin's method.
- `nye` - The length of Ye.
- `nu` - The number n_u of **pde** variables **u**.
- `U` - The *values* of **u** at the `Xe`(p) and `Ye`(q), **not** the B-spline coefficients **U** of (3.1). `U`(p,q,j) = u_j(t,`Xe`(p),`Ye`(q)), $p=1, \cdots$, `nxe`, $q=1, \cdots$, `nye` and $j=1, \cdots$, `nu`.
- `Ut` - The values of $\mathbf{u}_t$ at the `Xe`(p) and `Ye`(q), stored as above.
- `Ux` - The values of $\mathbf{u}_x$, stored as above.
- `Uy` - The values of $\mathbf{u}_y$, stored as above.
- `Uxt` - The values of $\mathbf{u}_{xt}$, stored as above.
- `Uyt` - The values of $\mathbf{u}_{yt}$, stored as above.

`AF` must return as *output*

- `A` - The value of **a** at the `Xe`(p) and `Xe`(q). `A`(p,q,j) = a_j(t,`Xe`(p),`Ye`(q)), for $p=1, \cdots$, `nxe`, $q=1,...,$ `nye` and $j=1, \cdots$, `nu`.
- `AU` - The partial derivatives of **a** with respect to **u** at the `Xe`(p) and `Ye`(q). `AU`(p,q,i,j) = $\partial a_i / \partial u_j$ (t, `Xe`(p),`Ye`(q)), for $p=1, \cdots$, `nxe`, $q=1,...,$ `nye` and $i,j=1, \cdots$, `nu`.
- `AUt` - The partial derivatives of **a** with respect to $\mathbf{u}_t$, as above.
- `AUx` - The partial derivatives of **a** with respect to $\mathbf{u}_x$, as above.
- `AUy` - The partial derivatives of **a** with respect to $\mathbf{u}_y$, as above.
- `AUxt` - The partial derivatives of **a** with respect to $\mathbf{u}_{xt}$, as above.
- `AUyt` - The partial derivatives of **a** with respect to $\mathbf{u}_{yt}$, as above.
- `F` - The value of **f** at the `Xe`(p) and `Ye`(q). `F`(p,q,j) = f_j(t,`Xe`(p),`Ye`(q)), for $p=1, \cdots$, `nxe`, $q=1,...,$ `nye` and $j=1, \cdots$, `nu`.
- `FU` - The partial derivatives of **f** with respect to **u** at the `Xe`(p) and `Ye`(q). `FU`(p,q,i,j) = $\partial f_i / \partial u_j$ (t, `Xe`(p),`Ye`(q)), for $p=1, \cdots$, `nxe`, $q=1,...,$ `nye` and $i,j=1, \cdots$, `nu`.
- `FUt` - The partial derivatives of **f** with respect to $\mathbf{u}_t$, as above.
- `FUx` - The partial derivatives of **f** with respect to $\mathbf{u}_x$, as above.
- `FUy` - The partial derivatives of **f** with respect to $\mathbf{u}_y$, as above.
- `FUxt` - The partial derivatives of **f** with respect to $\mathbf{u}_{xt}$, as above.
- `FUyt` - The partial derivatives of **f** with respect to $\mathbf{u}_{yt}$, as above.

When `TTGR` needs the boundary conditions it will

```
Call BC(t,Xe,nxe,Ye,nye,Lx,Rx,Ly,Ry,
        U,Ut,Ux,Uy,Uxt,Uyt,nu,
        B,BU,BUt,BUx,BUy,BUxt,BUyt)
```

Before `TTGR` calls `BC`, it sets to **0** the 7 arrays `B` through `BUyt`, and provides the *input*

- `t` - The current value of time.
- `Xe` - A list of points x where **b** is to be evaluated. This `Xe` is *not* the B-spline mesh X. The points `Xe` at which **b** is desired are determined by the quadrature rule used by `TTGR` to implement Galerkin's method.
- `nxe` - The length of Xe.
- `Ye` - A list of points y where **b** is to be evaluated. This `Ye` is *not* the B-spline mesh Y. The points `Ye` at which **b** is desired are determined by the quadrature rule used by `TTGR` to implement Galerkin's method.
- `nye` - The length of Ye.
- `Lx` - The left-hand end-point of the x spatial domain.
- `Rx` - The right-hand end-point of the x spatial domain.
- `Ly` - The left-hand end-point of the y spatial domain.
- `Ry` - The right-hand end-point of the y spatial domain.
- `U` - `U`$(p,q,i) = u_i$(t,`Xe`(p),`Ye`(q)) for $i=1,\cdots,$ `nu`, $p=1,...,$ `nxe` and $q=1,\cdots,$ `nye`.
- `Ut` - `Ut` $= \mathbf{u}_t$, stored as above.
- `Ux` - `Ux` $= \mathbf{u}_x$, as above.
- `Uy` - `Uy` $= \mathbf{u}_y$, as above.
- `Uxt` - `Uxt` $= \mathbf{u}_{xt}$, as above.
- `Uyt` - `Uyt` $= \mathbf{u}_{yt}$, as above.
- `nu` - The number n_u of **pde** variables **u**.

`BC` must return as *output*

- `B` - `B`$(p,q,i) = \mathbf{b}$, $i=1,\cdots,$ `nu`, $p=1,...,$ `nxe` and $q=1,\cdots,$ `nye`. see (2.2).
- `BU` - `BU`$(p,q,i,j) = \partial b_i / \partial u_j$(`Xe`$(p)$,`Ye`$(q)$), $i,j=1,\cdots,$ `nu` and $p=1,...,$ `nxe` and $q=1,\cdots,$ `nye`.
- `BUt` - `BUt`$(p,q,i,j) = \partial b_i / \partial u_{jt}$, as above.
- `BUx` - `BUx`$(p,q,i,j) = \partial b_i / \partial u_{jx}$, as above.
- `BUy` - `BUy`$(p,q,i,j) = \partial b_i / \partial u_{jy}$, as above.
- `BUxt` - `BUxt`$(p,q,i,j) = \partial b_i / \partial u_{jtx}$, as above.
- `BUyt` - `BUyt`$(p,q,i,j) = \partial b_i / \partial u_{jty}$, as above.

`HANDLE` Description.

The user-supplied output and control subroutine `HANDLE` is now described. The numerical solution at an instant in time is obtained only after some rather lengthy and complex calculations involving trying several small sub-steps in time; see Appendix 2 for more detail. When `TTGR` has finally come up with a solution as accurate as requested by the user, it just has to tell the user the good news. At the end of each time-step, `TTGR` will

```
Call HANDLE(t0,U0,t,U,lU,dt,tstop)
```

so that the user may look at, print out, plot, fondle, or do whatever is desired with the solution. If the output at the end of each time-step is not desired, and only the solution at time `tstop` is needed, the "Return-End" `HANDLE` subroutine `TTGRH` may be used.

TTGR also invokes HANDLE whenever it tries to take a time step and fails to obtain the user desired accuracy. This may be caused by the time step dt being too large. Or it may mean that there is something "funny" going on near time t. In either case, the user may want to know that TTGR failed at time t. Such things are called "restarts" and are typically expensive and worth knowing about.

The input provided by TTGR to HANDLE is

- t0 - Time at the beginning of the time-step just completed.
- U0 - **pde** solution **u** at time t0 is given by B-spline coefficients U0. This array is Real of length lU $= n_x\, n_y\, n_u$. The coefficients are stored as if U0 were dimensioned (nx,ny,Nu), where the *x* grid has nx points, similarly for *y* and there are Nu **pde**s.
- t - Time at the end of the time-step just completed. The "current" value of time.
- U - **pde** solution **u** at time t is given by B-spline coefficients U. If t0 = t, then a restart is in progress and the values in U are meaningless.
- lU - The length of the array U.
- dt - The current "optimal" value of dt.
- tstop - The current final value for time.

The use of lU above is a botch required by the use of IODE to solve the spatially discretized problem - the output routine calling sequence is fixed. In the example code of Appendix 4, the needed values of nx, ny and Nu are obtained, magically, from Common regions internal to TTGR. This botch will probably be fixed in the next edition of TTGR.

The output from HANDLE is

- t - May be altered by the user. Not too many people want to do this, but it is allowed.
- U - May be altered by the user. For example, the user may want to force the solution to be non-negative or monotone.
- dt - May be altered by the user. The user may want to choose dt so that some particular value of time is achieved on the next time-step.
- tstop - May be altered by the user. For example, the user may only want to integrate until $\mathbf{u}_t$ is "small enough" and then stop.

Evaluating the Solution.

To evaluate the solution created by TTGR, simply

```
Call  TSD1(p, k, t, it, nt, a, e, ie, ne, m, f) .
```

This is general purpose, multi-dimensional tensor spline evaluation software written by E. H. Grosse and distributed with TTGR to make evaluation easier for both users and the authors. Example 1 in Appendix 4 shows TSD1 at work. The argument descriptions below are from that software, and users should think of p as 2. The input to TSD1 is

- p - The number of coordinates, that is, 2.
- k - The order of the tensor product spline. Note that k, it, nt, ie, ne and m are vectors of length p with an independent value for each coordinate. Think of k(1)=kx and k(2)=ky.
- t - An array containing the meshes for each spatial coordinate. t(it(1)),...,t(it(1)-1+nt(1)) contains the mesh for the first coordinate (*x*), t(it(2)),...,t(it(2)-1+nt(2)) for the second (*y*), and so on.
- it - The pointers to the meshes in each coordinate, as used above.
- nt - Number of mesh points in each dimension.
- a - B-spline coefficients, stored as if dimensioned (nt(1)-k(1),...,nt(p)-k(p)).

e - The grid on which evaluation is to be done. It is stored like `t`, `it` and `nt`.

ie - Evaluation indices in each dimension, as with `it` above.

ne - The number of evaluation points in each dimension.

m - Order of the partial derivatives. See `f` below.

The output of `TSD1` is

f - Derivative values, stored as if dimensioned `( ne(1),...,ne(p) )`. The derivative is of order `m(1)` with respect to the first coordinate, `m(2)` with respect to the second, and so on. That is, `f(i,j)` $= \frac{\partial^{m(1)}}{\partial x^{m(1)}} \frac{\partial^{m(2)}}{\partial y^{m(2)}} s(x_i, y_j)$, where s is the spline whose coefficients are in `a`, for `i` = 1,...,`ne(1)` and `j` = 1,...,`ne(2)`. This means that `m = (0,0)` gives the function value, for example.

The double precision version of `TSD1` is `DTSD1` with all Real arguments typed double precision.

Obtaining Initial Conditions.

If your initial conditions are constant, setting the initial values for **U** in (3.1) is easy: simply set $U_{qp} \equiv$ constant. If your initial data is not constant, then you can

```
Call TSL2W(p, k, t, it, nt, e, ie, w, iw, ne, f, a)
```

This is general purpose, multi-dimensional tensor spline fitting software written by E. H. Grosse [19] and distributed with `TTGR` to make fitting easier for both users and the authors. The argument descriptions below are from that software, and users should think of `p` as 2. The input to `TSL2W` is

p - The number of coordinates, that is, 2.

k - The order of the tensor product spline. Note that `k`, `it`, `nt`, `ie` and `ne` are vectors with an independent value for each coordinate.

t - The mesh on which the spline is defined. See the introduction to the PORT chapter on approximation for advice on laying out the mesh; `t(it(1)),...,t(it(1)-1+nt(1))` contains the mesh for the first coordinate, `t(it(2)),...,t(it(2)-1+nt(2))` for the second, and so on.

it - Mesh point indices.

nt - Number of mesh points in each dimension.

e - The grid on which data is defined. It is stored like `t`, `it` and `nt`.

ie - Data indices in each dimension.

w - Weights, stored like `e`. The weight for the `(i1,i2,...)` data point is `w(iw(1)-1+i1) * w(iw(2)-1+i2) * ...`

iw - Indices for the weights in each dimension.

ne - The number of data points in each dimension.

f - Data values, stored as if dimensioned `(ne(1),...,ne(p))`.

The output of `TSL2W` is

a - B-spline coefficients, stored as if dimensioned `(nt(1)-k(1), ... , nt(p)-k(p))`.

The double precision version of `TSL2W` is `DTSL2W` with all Real arguments typed double precision.

Other Ways to Use and Speed-Up `TTGR`.

There are many "knobs" in `TTGR`; section 6 describes their use.

Trouble in AF or BC.

If, for one reason or another, the user cannot evaluate the appropriate functions when AF or BC are called, this fact can be communicated to TTGR through the named Common region

```
Common / TTGRF / Failed ; Logical Failed .
```

Before TTGR calls AF or BC, it sets Failed = .False. . Thus, if the user doesn't use or even know of the existence of TTGRF, TTGR assumes that everything has been correctly computed on return from those subroutines. If, however, the user has a problem, uses TTGRF, and sets Failed = .True., TTGR will automatically lower the time-step in an attempt to obtain a more accurate, and hence more reasonable, numerical solution so that AF and BC can do their job.

The Double Precision version of TTGRF is DTTGRF.

Mapping Software

There are many ways to map ink-blots onto a rectangle, with conformal mapping preferable. However, conformal mapping is a tricky business and is not strictly speaking necessary when solving **pde**s. Much simpler non-conformal maps can often be used. For example, suppose you want to map a domain like

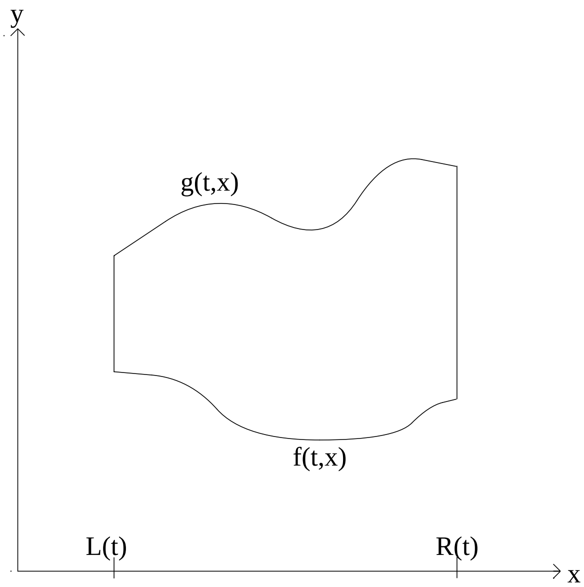


Figure 2

onto the unit rectangle, where f and g are nice, smooth functions of x. The mapping

$$
\begin{aligned}
x &= L(t)+\xi\,(R(t)-L(t)) \\
y &= f(t,x)+\eta\,(g(t,x)-f(t,x))
\end{aligned}
\tag{4.1}
$$

takes the unit square in ξ,η into the domain of Figure 2. Moreover, the mapping is smooth. This implies that the mesh in ξ,η need only be fine where the domain or the solution is kinky.

Mappings like the one above are so common that they have been canned for use by TTGR. If you wish to map some ξ,η pairs into x,y pairs, then the

```
Call BTMAP(t,c,e,nx,ny, LR,BT, x,y,D)
```

will do the job. The input arguments are

t - The time t.

`c` - The ξ values to be used in the mapping.

`e` - The η values to be used in the mapping.

`nx` - The length of the ξ array.

`ny` - The length of the η array.

`LR` - The name of a subroutine, declared `External` in the program calling `BTMAP`, for giving information about $L(t)$ and $R(t)$.

When `BTMAP` wants the values of $L(t)$ and $R(t)$ it will

```
Call LR(t,L,R,Lt,Rt)
```

The input to `LR` is

`t` - Time t.

The output from `LR` is

`L` - The value of $L(t)$.

`R` - The value of $R(t)$.

`Lt` - The value of $L'(t)$.

`Rt` - The value of $R'(t)$.

`BT` - The name of a subroutine, declared `External` in the program calling `BTMAP`, for giving information about $f(t,x)$ and $g(t,x)$.

When `BTMAP` wants to evaluate f and g it will

```
Call BT(t,x,f,g,fx,gx,ft,gt)
```

The input to `BT` is

`t` - The time t.

`x` - The point x for which $f(t,x)$ and $g(t,x)$ are desired.

The output from `BT` is

`f` - The value of $f(t,x)$.

`g` - The value of $g(t,x)$.

`fx` - The value of $\frac{\partial f}{\partial x}(t,x)$.

`gx` - The value of $\frac{\partial g}{\partial x}(t,x)$.

`ft` - The value of $\frac{\partial f}{\partial t}(t,x)$.

`gt` - The value of $\frac{\partial g}{\partial t}(t,x)$.

The output from `BTMAP` is

`x` - `x`$(p,q) = x(\xi(p), \eta(q))$, for $p = 1,\ldots,$ `nx` and $q = 1,\ldots,$ `ny`.

`y` - `y`$(p,q) = y(\xi(p), \eta(q))$, for $p = 1,\ldots,$ `nx` and $q = 1,\ldots,$ `ny`.

`D` - An array giving the partial derivatives of x and y wrt. ξ, η and t. This information is necessary in the **pde** mapping software to be described shortly.

$$\mathrm{D}(p,q,1,1) = \frac{\partial x}{\partial \xi}\ (t,\xi(p),\eta\ (q))$$
$$\mathrm{D}(p,q,1,2) = \frac{\partial x}{\partial \eta}\ (t,\xi(p),\eta\ (q))$$
$$\mathrm{D}(p,q,1,3) = \frac{\partial x}{\partial t}\ (t,\xi(p),\eta\ (q))$$
$$\mathrm{D}(p,q,2,1) = \frac{\partial y}{\partial \xi}\ (t,\xi(p),\eta\ (q))$$
$$\mathrm{D}(p,q,2,2) = \frac{\partial y}{\partial \eta}\ (t,\xi(p),\eta\ (q))$$
$$\mathrm{D}(p,q,2,3) = \frac{\partial y}{\partial t}\ (t,\xi(p),\eta\ (q))$$

The above example is given to show the ease with which domains can be mapped onto rectangles. In fact, here is the `Ratfor` source for `BTMAP`, which maps the domain of Figure 2 onto the unit square

```
    Subroutine BTMAP(t,c,e,nx,ny, LR,BT, x,y,D)

  # To map (c,e): (0,1)x(0,1) to Left-right x Bottom-top.

    Double Precision t,c(nx),e(ny),x(nx,ny),y(nx,ny),D(nx,ny,2,3)
    Integer nx,ny
    External LR,BT

    Double Precision L,R,Lt,Rt,f,g,fx,gx,ft,gt,l
    Integer p,q

    Do p = 1, nx
      {
      Call LR(t,L,R,Lt,Rt)    # Get (L,R)(t) and (L’,R’)(t).
      l = R-L

      Do q = 1, ny
        {
        x(p,q) = L+c(p)*l
        D(p,q,1,1) = l
        D(p,q,1,2) = 0
        D(p,q,1,3) = Lt+c(p)*(Rt-Lt)

        Call BT(t,x(p,q),f,g,fx,gx,ft,gt)    # Get (f,g,fx,gx,ft,gt)(t,x).

        y(p,q) = f+e(q)*(g-f)
        D(p,q,2,1) = (fx+e(q)*(gx-fx))*l
        D(p,q,2,2) = g-f
        D(p,q,2,3) = ft + e(q)*(gt-ft) + ( fx + e(q)* ( gx-fx ) )*D(p,q,1,3)
        }
      }

    Return

    End
```

The programs for performing such maps are trivial and new ones can easily be created for other types of domains.

pde Mapping Software.

The mapping (4.1) is a specific example of a general mapping $(x,y)\ (\xi, \eta, t)$ which can be used to transform a **pde** for solution by `TTGR`. `TTGR` has general purpose mapping transformation procedures for taking the results of a map $(x,y)\ (\xi, \eta, t)$ and converting the **pde**s and **bc**s from $\mathbf{u}(t, x, y)$ to $\mathbf{w}(t, \xi, \eta) \equiv \mathbf{u}(t, (x,y)(\xi, \eta, t))$ coordinates, where $(x, y)\ (\xi, \eta)$ maps a rectangle onto the user's physical domain.

The user tells `TTGR` to solve for **w**, but the **pde** and **bc** definition can be made in the physical **u** system using the software to be described shortly.

The **pde** mapping proceeds as follows. We have

$$\mathbf{w}(t, \xi, \eta) \equiv \mathbf{u}(t, (x,y)(\xi, \eta, t))$$

and thus $\mathbf{w}_t \equiv \mathbf{u}_x \dfrac{dx}{dt} + \mathbf{u}_y \dfrac{dy}{dt} + \mathbf{u}_t$, $\mathbf{w}_\xi \equiv \mathbf{u}_x x_\xi + \mathbf{u}_y y_\xi$, $\mathbf{w}_\eta \equiv \mathbf{u}_x x_\eta + \mathbf{u}_y y_\eta$. This implies that **pde**s which began in the form (2.1) now have the form

$$\frac{\partial}{\partial \xi}\left(-x_\eta \mathbf{a}^{(2)} + y_\eta \mathbf{a}^{(1)} \right) + \frac{\partial}{\partial \eta}\left(-y_\xi \mathbf{a}^{(2)} + x_\xi \mathbf{a}^{(1)} \right) = \left(x_\xi y_\eta - x_\eta y_\xi \right) \cdot \mathbf{f}.$$

assuming that the mapping $(x,y)(\xi, \eta)$ has continuous second partials.

At the beginning of the user's `AF` procedure it is necessary to convert from the internal coordinate system of `TTGR`, namely **w** and ξ, η, to the user's physical coordinate system, **u** and x,y. To accomplish this the user must calculate the value of the map $(x,y)\ (\xi, \eta, t)$ at the ξ, η input to `AF` and its partials with respect to time, ξ, η, etc. That is, the "x" in the calling sequence to `AF` should be called "xi", similarly for "y", and the values of $(x,y)(\xi, \eta, t)$ should be computed by the user inside `AF`. Then

```
Call TTGRU(nx,ny,D, Ux,Uy,Ut,Nu)
```

will do the mapping from internal (**w**) to User (**u**) coordinates. Example 4 of Appendix 4 illustrates the use of `TTGRU`. The arguments to `TTGRU` are now described. The input to `TTGRU` is

- `nx` - The number of ξ points.
- `ny` - The number of η points.
- `D` - An output array as described above in `BTMAP`.
- `Ux` - The input argument of `AF` of the same name.
- `Uy` - The input argument of `AF` of the same name.
- `Ut` - The input argument of `AF` of the same name.
- `Nu` - The input argument of `AF` of the same name.

The output of `TTGRU` is

- `Ux` - The derivatives $\mathbf{u}_x$.
- `Uy` - The derivatives $\mathbf{u}_y$.
- `Ut` - The derivatives $\mathbf{u}_t$.

With the map of **w** into user coordinates **u** done using `TTGRU` above, the user then continues with `AF` as if there were no mapping being used at all. That is, the user can then think only in terms of the original variables, **u** and x,y.

However, before leaving `AF` the user must map the physical coordinates **u** back into the internal system `TTGR` is using, **w**. This is done by

```
Call TTGRG(nx,ny,D, Nu, A,AU,AUx,AUy, F,FU,FUx,FUy)
```

just before leaving `AF`.

The input arguments to `TTGRG` are as in `TTGRU` above.

The output of `TTGRG` is

A - **a** for the *w* and ξ, η coordinate system.

AU - The partials of **a** with respect to **w**.

AUx - The partials of **a** with respect to $\mathbf{w}_\xi$.

AUy - The partials of **a** with respect to $\mathbf{w}_\eta$.

F - **f** for the **w** and ξ, η system.

FU - The partials of **f** with respect to **w**.

FUx - The partials of **f** with respect to $\mathbf{w}_\xi$.

FUy - The partials of **f** with respect to $\mathbf{w}_\eta$.

The Double Precision versions of TTGRG and TTGRU are called DTTGRG and DTTGRU, respectively, with all Real arguments typed Double Precision.

Example 4 of Appendix 4 shows TTGRU and TTGRG at work.

Mapping Boundary Conditions

The **bc** mapping paradigm generally follows that for the **pde** mapping software described above. At the beginning of the user's BC procedure it is necessary to convert from the internal coordinate system of TTGR, namely **w** and ξ, η, to the user's physical coordinate system, **u** and x,y. To accomplish this the user must calculate the value of the map $(x,y)\ (\xi, \eta, t)$ at the ξ, η input to BC and its partials with respect to time, ξ, η, etc. Then TTGRU should be invoked, the **bc**s entered in terms of **u** and x,y, and then TTGRB invoked before returning from BC.

At the start of BC the user calls a mapping procedure to do the map from internal TTGR coordinates ξ, η, the BC input arguments "xi,eta", to x,y coordinates. The

```
Call TTGRU(nx,ny,D, Ux,Uy,Ut,Nu)
```

will do this mapping from internal (**w**) to user (**u**) coordinates. The input to TTGRU is as described above.

With the map of **w** into user coordinates **u** done using TTGRU above, the user continues with BC as if there were no mapping being used. That is, the user can then think only in terms of the original variables, **u** and x,y.

However, before leaving BC the user must map the physical coordinates **u** back into the internal system TTGR is using, **w**. This is done by

```
Call TTGRB(nx,ny,D, BUx,BUy,BUt)
```

just before leaving BC.

The input arguments to TTGRB are as above, and the outputs are the values of $\frac{\partial \mathbf{b}}{\partial \mathbf{w}}$, etc.

Example 4 in Appendix 4 shows TTGRB, TTGRG, TTGRU, as well as BTMAP, hard at work.

6. Advice for the Sporting User

This section describes, in gory detail, the various "knobs" and ways of overriding the default options and subprograms used in TTGR. Several examples of knob twiddling that can increase the speed of TTGR considerably are also given.

An exceedingly brief outline of the organization of TTGR is, in pseudo-English,

```
t0 = tstart
While ( t0 != tstop )        # Time-step loop.
    {
    Do m = 1, ..., mmax        # Loop to build extrapolation tableau.
        {
        Do istep = 1, ..., N(m)        # Sub-steps loop.
            {
            Do iter = 1, ..., maxit        # Newton loop.
                {
                Solve the linearized Galerkin Equations; Update solution
                Check ERROR for "convergence"; Check Convergence Rate
                }
            }
        Extrapolate and check the ERROR.
        }
    Get_Optimal_dt for next time-step.
    Output the results for this time-step ( HANDLE )
    t0 = t0 + dt
    }
```

where the capitalized items in parentheses refer to procedures names, and the linearized Galerkin solution is obtained by

> Get integrals for matrix (`AF`), mesh interval by mesh interval,
> Get the **bc**s (`BC`).
> Get **pde** (`AF`) on the boundary.
> Solve the **pde** system.

This section describes the many variables and procedures that define the above algorithm. These include the maximum number of levels of extrapolation to allow (`mmax`), the sequence of sub-steps to take (`N`(1), `N`(2), $\cdots$), how to solve the linearized Galerkin equations, and the maximum number of Newton iterations to allow (`maxit`).

Twiddling Procedure Knobs.

Control over the procedure `ERROR` is given by `TTGRR`. This routine allows the user to override some of the default routines of `TTGR`. `TTGRR` is invoked by

```
Call TTGRR(U,Nu,kx,x,nx, ky,y,ny,
           tstart,tstop,dt,
           AF,BC,
           ERROR,errpar,
           HANDLE)
```

The extra argument `ERROR` in this subroutine provides direct user control over the accuracy of the integration process.

The default routine, used by `TTGR`, is `TTGRE` for `ERROR`.

Error Options

There are several possible options available for error specification. First, the user, via the subprogram `ERROR`, may specify literally any accuracy requirement desired for the solution. This option has not yet been implemented because of using `IODE` raw. Users can roll their own `ERROR` routines, but `TTGR` will use its own internal one. Second, there are several popular methods of error control which are controlled by the switch `erputs`, and implemented by the subprogram `TTGRE`.

The error control provided in the subroutine `TTGRE` is based on the local value of the variables. That is, the error acceptable in $u_i(t,x,y)$ is

$$\texttt{errpar}(1) * \| \mathbf{U}_{..i} \|_\infty + \texttt{errpar}(2)$$

where $\mathbf{U}_{..i}$ denotes the block of B-spline coefficients for u_i, which depends only upon the current value of **u**.

The error control provided in the subroutine TTGRE is an *error per time-step* criterion. This can be rather bad if the time-steps taken during the solution process get very small - many time-steps will be taken and the errors may pile up in unacceptable amounts. Another error option is to use an *error per unit-time-step* criterion. By making the error tolerance in $u_i(t,x,y)$ look like

$$| \texttt{dt} | * (\texttt{errpar}(1) * \| \mathbf{U}_{..i} \|_\infty + \texttt{errpar}(2)) ,$$

when the time-step gets small, so does the error requirement. However, when using the error per unit-time-step criterion, the reverse argument holds - when dt is large, so is the error tolerance. The error per time-step versus unit-time-step option is controlled by the switch erputs as described below.

The output from TTGRR is the same as that for TTGR with the additional possibility that the user may alter errpar through his subprogram ERROR.

The double precision version of TTGRR is DTTGRR, with all Real arguments typed double precision, except errpar, which remains Real. Similarly for TTGRE and TTGRP.

Twiddling non-Procedure Knobs.

The main knob-twiddling routine is TTGRV. When the user wants to tinker some internal variable of TTGR the

```
Call TTGRV(j,f,r,i,l)
```

just before calling TTGR will do the trick. The input to TTGRV is an index (j) identifying the knob to be turned, and its value (one of f, r, i or l). For any value of j, only **one** of f, r, i or l is used to set the knob, the other variables may be set to anything, like 0d0, 0e0, 0 or .true.. Any number of variables can be set by calling TTGRV any number of times. The input to TTGRV is

- j - The index of the variable to be set. If j = 0, then **all** variables are set to their default values. You should not have to use j = 0 unless you have already set a few parameters using TTGRV and you now want to reset the default values.
- f - If the variable to be set is a working-precision item, f is to be its new value. Working precision is Real for TTGR and Double Precision for DTTGR.
- r - If the variable to be set is a Real item (hfract, egive), r is to be its new value.
- i - If the variable to be set is an Integer item, i is to be its new value.
- l - If the variable to be set is a Logical item, l is to be its new value.

The output of TTGRV is the internal knowledge of the new value of the variable that has been set.

The following list gives the items that can be set using TTGRV.

- theta - The time-discretization parameter, see section 3 and Appendix 2. When theta = 1 the extremely stable, first order accurate Backwards-Euler formula is used. For theta = 1/2, the second-order Crank-Nicholson scheme is used. If theta $\neq$ 1/2, then N = { 1, 2, 3, 4, 6, $\cdots$ } and gamma = 1. If theta = $\frac{1}{2}$, then N = { 2, 4, 6, $\cdots$ } and gamma = 2. 0 <= theta <= 1 is required. Default: theta = 1. j = 1. theta = f.
- beta - The error in the discretization scheme is proportional to (t1-t0)**beta. Default: beta = 1. j = 2. beta = f.
- gamma - The error in the discretization scheme is proportional to dt**gamma. Default: gamma = 1. j = 3. gamma = f.

`delta` - The error request is proportional to `dt**delta`. Default: `delta` = 0. `j` = 4. `delta` = `f`.

`hfract` - A Real variable indicating how small a time-step the user will take relative to `dt`. Default: `hfract` = 1. `j` = 1001. `hfract` = `r`.

`egive` - A Real variable controlling how accurately Newton's method will try to solve the nonlinear equations, relative to the user's accuracy request for the solution to the **pde**. Specifically, it will try to solve the nonlinear equations to a tolerance of (the user's error request to `TTGR`) / `egive`. `egive` should thus be greater than 1. Default: `egive` = 1e+2. `j` = 1002. `egive` = `r`.

`kj` - A variable that controls the frequency with which the Jacobian is re-computed. Such evaluations can be very costly and `kj` (for "KeepJacobian") controls them in the following way:

0 - New Jacobian every Newton iteration.
Very safe and stable, expensive.
1 - New Jacobian every time sub-step.
Less safe, stable and expensive.
2 - New Jacobian for each time-step.
Not very safe, stable or cheap, except for nearly linear problems.
3 - New Jacobian whenever there is a re-start.
Mostly used for linear or nearly linear problems.
Cheap but flaky.
4 - New Jacobian whenever Newton iteration fails to converge.
Only updates Jacobian if it appears out-of-date.
Ditto rest of discussion of 3 above.
5 - Only computes the Jacobian ONCE.
Use only for nearly linear problems.
Exceedingly cheap, when it works.

Default: `kj` = 0. `j` = 2001. `kj` = `i`.

`minit` - The minimum number of Newton iterations to go before checking that the convergence rate is reasonable. Default: `minit` = 10. `j` = 2002. `minit` = `i`.

`maxit` - The maximum number of Newton iterations to use. Default: `maxit` = 50. `j` = 2003. `maxit` = `i`.

`kmax` - The maximum number of columns allowed in the extrapolation tableau. The maximal order that `TTGR` can achieve is then 2*`kmax` if `theta` = 0.5e0, or `kmax` if `theta` ≠ 0.5e0. Default: `kmax` = 10. `j` = 2004.

`kinit` - The initial level of extrapolation to use for the first time-step. This can allow `TTGR` to use a higher-order scheme from the start. Default: kinit = 2. `j` = 2005. `kinit` = `i`.

`mmax` - The maximum number of levels of extrapolation permitted. `mmax` >= `kmax` + 2 is required and `mmax` >= `kmax` + 4 is a good idea. Default: `mmax = 15`. `j` = 2006. `mmax` = `i`.

`mxq` - The number of *x* Gaussian quadrature points to be used to compute the Galerkin integrals, see section 3. `mxq` >= `kx-1` is required. Default: `mxq` = `kx`. `j` = 2008. `mxq` = `i`.

`myq` - The number of *y* Gaussian quadrature points to be used to compute the Galerkin integrals, see section 3. `myq` >= `ky-1` is required. Default: `myq` = `ky`. `j` = 2009. `myq` = `i`.

`LA` - An Integer variable indicating how the matrix equations should be solved.

−1 - non-pivoting banded solve.
+1 - pivoting banded solve. Default; see (6.1) below for reasons.
−2 - non-pivoting sparse solve.
+2 - pivoting sparse solve.

Default: `LA` = +1. `j` = 2010. `LA` = `i`.

`Pieces` - An Integer indicating how much of the Jacobian should be used.

+2 - the whole thing. Default.
0 - only the diagonal **pde** terms.
–1 - only the lower triangular terms.
+1 - the upper triangular terms.

Default: `Pieces` = +2. `j` = 2011. `Pieces` = `i`.

`PC` - An Integer variable indicating the pre-conditioner to be used when solving the matrix equations iteratively.

0 - none. Default - using a direct solve.
1 - incomplete LU.
2 - modified incomplete LU. See [20].

Default: `PC` = 0. `j` = 2012. `PC` = `i`.

`Accel` - An Integer variable indicating the accelerator to be used when solving the matrix equations iteratively.

0 - none. Default - doing a direct solve.
1 - conjugate gradient on the normal equations.
2 - Orthomin. See [11].

Default: `Accel` = 0. `j` = 2013. `Accel` = `i`.

`xpoly` - A Logical variable indicating whether polynomial (True) or rational (False) extrapolation is to be used. Default: `xpoly` = False. `j` = 3001. `xpoly` = `l`.

`erputs` - Logical variable controlling the use of error-per-unit-time-step. If True, then use per-unit-time-step error criterion. Otherwise, use the per-step criterion. Default: `erputs` = False. `j` = 3002. `erputs` = `l`.

`N` - An Integer array of length `mmax` giving the number of sub-steps to be used in the extrapolation. `N` must be strictly monotone increasing and positive. `N`(i) is set by `j` = 4000+i. If `N`(i) is set, then `N`(i+1) is set to 0 by default; so be sure to set `N` in increasing order of i. For any `N`(i) which is 0, a default value is computed. The rule is that if only `N`(1) is set, then the user wants `N`(i) ≡ $(\sqrt{2})^{i-1}$ `N`(1). If only `N`(1) and `N`(2) are set, then `N`(i) = (`N`(2)/`N`(1)) `N`(i-1) for any `N`(i) = 0. If `N`(3) is also set, then `N`(i) = 2 * `N`(i-2), for any `N`(i) = 0. See the `theta` description above for some default `N` values. `j` = 4000 + i. `N`(i) = `i`.

The following table summarizes the values that can be set by `TTGRV`

Name	j	Default	Set to
theta	1	1	f
beta	2	1	f
gamma	3	1	f
delta	4	0	f
hfract	1001	1	r
egive	1002	100	r
kj	2001	0	i
minit	2002	10	i
maxit	2003	50	i
kmax	2004	10	i
kinit	2005	2	i
mmax	2006	15	i

`mxq`	2008	`0`	`i`
`myq`	2009	`0`	`i`
`la`	2010	1	`i`
`pieces`	2011	+2	`i`
`pc`	2012	0	`i`
`accel`	2013	0	`i`
`xpoly`	3001	`False`	`l`
`erputs`	3002	`False`	`l`
N(i)	4000+i	–	`i`

Note that `mxq` = 0 means that `kx` points will be used in *x*, by default. Similarly for `myq`.

The Double precision version of `TTGRV` is `DTTGRV`, with all Real arguments typed Double precision, except `hfract` and `egive` which remain Real.

Space Used

The space used by the various knob settings is summarized in the following table

Space Required	
Method	Words
No pivot/Band	$n_u\, n\, (2\, H - 1)$
Pivot/Band	$n_u\, n\, (3\, H - 1)$
Sparse †	$O(n_u^2\, k_x\, k_y\, n_x\, n \log n)$
MIC	$3\, (n_u^2\, n\, (2\, k_x - 1)\, (2\, k_y - 1))$

where $H \equiv n_u\, (\, k_x + (\, n_x - k_x\,)\, (k_y - 1\,)\,)$, and $n \equiv n_x\, n_y$. When `Pieces` is 0, a factor of n_u can be removed from the figures.

† - This figure is *usually* accurate, but not always. See discussion of the **pde** $u_{xy} = 0$ below.

Time Used.

The floating-point operation count used by the various knob settings is summarized in the following table

Operation Count	
Method	Multiplies
No pivot/Band	$n_u\, n\, (H - 1)^2$
Pivot/Band	$2\, n_u\, n\, (H - 1)^2$
Sparse †	$O(n_u^2\, k_x\, k_y\, n\, n_x)$
MIC ††	$O(n_u^2\, n^{1.25}\, (2\, k_x - 1)\, (2\, k_y - 1))$

where $H \equiv n_u\, (\, k_x + (\, n_x - k_x\,)\, (k_y - 1\,)\,)$, and $n \equiv n_x\, n_y$. When `Pieces` is 0, a factor of n_u should be removed from the figures.

† - This figure is usually accurate, but see the discussion of the **pde** $u_{xy} = 0$ below.

†† - Modified incomplete factorization schemes *usually* converge in $O(n^{.25})$ iterations, but not always. For example, problems like $u_x + u_y = 0$ will cause MIC to fail. See also [12] for examples.

Note that the banded options vectorize and thus on the Cray X-MP may take less time even though they require more operations than the sparse options.

A Bad Example

Consider the **pde**

$$u_{xy} = 0 \tag{6.1}$$

on the domain $[0,1]\times[0,1]$. Use Dirichlet boundary conditions on the left and bottom sides of the domain. This problem generates a Jacobian most of whose diagonal elements are 0. This means that any non-pivoting LU factorization scheme will fail on this problem. If the pivoting sparse matrix option is used, the minimum degree ordering used there, which implicitly assumes that the diagonal elements are non-zero, causes the LU factorization to **fill-in** completely. That is, the pivoting sparse option uses $O(n^4)$ space and $O(n^6)$ work on an *n* by *n* grid. The pivoting banded solver has no trouble with this problem at all.

In general, the pivoting banded solver is slower on scalar machines and uses more space than the other options, for sufficiently large numbers of mesh points. However, all of the other options have the property that their **worst-case** behavior, on the above problem, is much worse than that of pivoting band solution, which always uses roughly the same space and run-time.

This explains why we chose pivoting banded as the default solver - it is predictable, dependable and never blows up on nice problems. It also turns out to be fast on vector machines for most practical grids.

Some Knob Twiddling

We now show how twiddling some of the knobs in `TTGRV` can affect the run-time for example 2 in section 4. The knobs considered are `maxit`, `mxq`, `myq`, `LA`, `Pieces`, `PC`, and `Accel`.

The default call to `TTGR` uses a full Newton iteration scheme to solve the nonlinear equations (`maxit` = 50). This will be referred to as NBE for Nonlinear Backwards Euler. The run-times reported in Appendix 4 for the examples are for the default NBE scheme.

The second scheme considered is the same as NBE but with `maxit` set to 1. This may seem strange, in that it does only *one* Newton iteration to solve each nonlinear equation. However, it works quite well on most problems and it possesses the asymptotic expansion needed for extrapolation to work, see Appendix 2. This scheme will be denoted by LBE, for Linearized Backwards Euler. The user tells `TTGR` that `maxit` is to be 1 by inserting the

```
call ttgrv(+2003,0d0,0e0,1,.true.)
```

just before the call to `TTGR`.

The third scheme will be the LBE scheme above with `mxq` = `kx-1` and `myq` = `ky-1`, which is usually safe for problems with a partial spatial derivative in them. A little care has to be used here however; if the time step `dt` gets exceedingly small, the Jacobian will become singular, because it will approach a multiple of the Jacobian for the problem $u = 0$. That Jacobian is singular unless k_x and k_y quadrature points are used. This scheme will be denoted by LBE.q for LBE with special Quadrature. The user tells `TTGR` that `mxq` is to be `kx-1`, as well as setting `myq`, by inserting

```
call ttgrv(+2008,0e0,0e0,kx-1,.true.)
call ttgrv(+2009,0e0,0e0,ky-1,.true.)
```

just before the call to `TTGR`.

The fourth scheme will be the above LBE.q with `Pieces = 0`. This scheme trades a few more time-steps for much cheaper Jacobian calculation and solution times. The main assumption here is that the "big" (like $\mathbf{u}_t$ and $\mathbf{u}_{xx}$) terms are on the block diagonal of the **pde**. This scheme will be denoted by LBE.q0. So long as all the $\mathbf{u}_t$ terms are on the diagonal, this option is theoretically guaranteed to converge. The issue is whether it is faster than dealing with the whole Jacobian. The user tells `TTGR` that `Pieces` is to be `0` by inserting

```
call ttgrv(+2011,0e0,0e0,0,.true.)
```

just before the call to `TTGR`.

Finally, we use the above LBE.q0 with `LA` = -2, `PC` = 2 and `Accel` = 2. This uses Modified incomplete factorizations and Orthomin to solve the linear systems. This scheme will be denoted by

LBE.q0m.

The user tells `TTGR` that `LA` is to be –2, etc by inserting the calls

```
call ttgrv(+2010,0d0,0e0,-2,.true.)
call ttgrv(+2012,0d0,0e0,2,.true.)
call ttgrv(+2013,0d0,0e0,2,.true.)
```

just before the call to `TTGR`.

We summarize the methods used in the table

Description of Runs			
Label	Order	Mesh	Method
2n	2	21 by 21	NBE
2l	2	21 by 21	LBE
2lq	2	21 by 21	LBE.q
2lq0	2	21 by 21	LBE.q0
2lq0m	2	21 by 21	above with Orthomin and MIC
n	4	3 by 3	NBE
l	4	3 by 3	LBE
lq	4	3 by 3	LBE.q
lq0	4	3 by 3	LBE.q0
lq0m	4	3 by 3	above with Orthomin and MIC

The order 2 grid was taken to be 21 by 21 to get the same accuracy as the order 4 solution on a 3 by 3 grid.

The results for the five methods described above are summarized in the table:

Effect of Different Solution Methods on Example 2			
Method	Space	Time (secs)	Jacobians
2n	134008	5219.2	24
2l	134008	1972.9	9
2lq	133812	1603.1	9
2lq0	72954	812.8	12
2lq0m	44743	506.7	12
n	9398	443.6	30
l	9398	133.0	9
lq	7990	108.6	9
lq0	6090	111.1	15
lq0m	9251	111.8	15

All of the above runs were made in single precision on a Vax 11/750, equipped with a floating-point accelerator, under the UNIX® operating system, Research Eighth Edition.

The first 5 runs above refer to using $k_x = 2 = k_y$ on a 21 by 21 grid. The last five refer to using fourth order splines on a 3 by 3 grid. The table shows that LBE saves the cost of several iterations per time step. Using fewer quadrature points saves even more time. LBE.q0 results in a big savings for linear splines, but not much for cubics. The reason is that the Jacobian is 800 by 800 for linear, but less than 100 by 100 for cubics. Note also that `Pieces=0` used more time-steps but still ran faster for the order 2 run because each step was significantly faster. Finally, Orthomin saves a lot for linear splines, and nothing at all for cubics.

Note that the *slowest* run with $k = 4$ is *faster* than the fastest $k = 2$ solution setting, and also uses less space. Using higher order schemes is even more important than twiddling knobs in TTGR, at least for this example.

The Right Stuff.

The discussion in [26] shows why NBE was chosen as the default method for TTGR. It is very robust and reasonably fast. Additional speed can be achieved, but only by sacrificing robustness.

If the user wishes to speed TTGR up, the simplest and safest thing to do is try LBE. It usually works and is faster than NBE. The user should make sure to compare at least a few LBE runs with NBE, to make sure it is correct and running efficiently. Once the utility of LBE for the class of problems at hand has been ensured by a few comparisons, the production runs can be done with LBE.

Clearly, the others knobs of TTGR can also make things run a lot faster. But their tuning is highly problem dependent. The authors would appreciate hearing from users what settings of knobs they find give useful, or awful, results.

Run-time Statistics.

A subroutine is provided to print run-time statistics for TTGR. The statement

```
Call TTGRX
```

will print a line of the form:

```
ttgr(j,f,ts,ss,nit,nd,nf,r) =  130  130   15   76  130    0    0    0
```

The fields of this line refer to

- j - The number of Jacobian evaluations.
- f - The number of factorizations of the Jacobian.
- ts - The number of time-steps.
- ss - The number of sub-steps.
- nit - The number of Newton iterations.
- nd - The number of predicted Newton failures (error increasing).
- nf - The number of Newton failures (more than maxit iterations).
- r - The number of restarts.

If TTGRX is invoked by the user while inside TTGR, the statistics reported will be the current values. If invoked outside TTGR, the statistics will be those of the last call to TTGR.

Another way Into TTGR.

The following subprograms are described for historical reasons. Their use is strongly discouraged, but possible.

For those who do not wish (or like) to use TTGR/TTGRV/TTGRR, there is another way of entering TTGR

```
Call TTGRS(U,Nu,kx,x,nx, ky,y,ny,
           tstart,tstop,dt,
           AF,BC,
           errpar,
           HANDLE)
```

with further control over the error of the solution given by

```
Call TTGR1(U,Nu,kx,x,nx, ky,y,ny,
           tstart,tstop,dt,
           mxq,myq,
           AF,BC,
           ERROR,errpar,
           HANDLE)
```

where the default method for both of the above is LBE. This is to keep them upwards compatible with their previous versions. Note that NBE is the default for `TTGR`, *not* LBE. Thus, the default method for `TTGRS` and `TTGR1` is different from that for `TTGR`

Even more control over the integration process is given by

```
Call TTGR2(U,Nu,kx,x,nx, ky,y,ny,
           tstart,tstop,dt,
           mxq,myq,
           LA,Pieces,PC,Accel,
           AF,BC,
           ERROR,errpar,
           HANDLE)
```

Acknowledgements

Georgia Fisanick, Dave Edelson and Leonilda Farrow, chemists all, were the reason for this package being created. Their constant badgering for something that "works" without worrying about being "optimal" has been successful. Also, Jack Dongarra of Argonne shared his knowledge and code for doing preconditioned conjugate-gradient solution of the linear systems. Without that leg-up from him, we might still be stuck with sparse-matrix and other much slower methods.

Bibliography

[1] C. deBoor, **A Practical Guide to Splines,** Springer, New York, Applied Math. Sciences 27, 1978.

[2] C. de Boor, "On Uniform Approximation by Splines," *J. Approx. Th.* **1,** 219-235(1968).

[3] C. de Boor, "On Calculating with B-splines," *J. Approx. Th.* **6,** 50-62(1972).

[4] R. Bulirsch and J. Stoer, "Fehlerabschatzungen und Extrapolation mit rationalen Funktionen bei Verfahren vom Richardson-Typus," *Numer. Math.* **6,** 413-427(1964).

[5] R. Bulirsch and J. Stoer, "Numerical Treatment of Ordinary Differential Equations by Extrapolation Methods," *Numer. Math.* **8,** 1-13(1966).

[6] R. Bulirsch and J. Stoer, "Asymptotic Upper and Lower Bounds for Results of Extrapolation Methods," *Numer. Math.* **8,** 93-104(1966).

[7] R. Courant and D. Hilbert, **Methods of Mathematical Physics,** Vol. 1, Interscience, New York, 1966.

[8] H.B. Curry and I.J. Schoenberg, "On Polya Frequency Functions IV: The Fundamental Spline Functions and their Limits," *J. of Anal. and Math.* **17,** 71-107(1966).

[9] G. Dahlquist, "A Special Stability Problem for Linear Multistep Methods," *BIT* **3,** 27-43(1963).

[10] G. Dahlquist, "Stability Questions for Some Numerical Methods for Ordinary Differential Equations," *Proc. Symp. for Applied Math.* **15,** 147-158(1963).

[11] Howard C. Elman, "Iterative Methods for Large, Sparse, Nonsymmetric Systems of Linear Equations," Yale Research Report YALEU/DCS/RR-229, February, 1982.

[12] Howard C. Elman, "A Stability Analysis of Incomplete LU Factorizations," Yale Research Report YALEU/DCS/RR-365, February, 1985.

[13] W.H. Enright, T.E. Hull and B. Lindberg, "Comparing Numerical Methods for Stiff Systems of Ordinary Differential Equations," *BIT* **15,** 10-48(1975).

[14] P.A. Fox, A.D. Hall and N.L. Schryer, "The PORT Mathematical Subroutine Library," *TOMS,* **4,** 104-126(1978).

[15] C.W. Gear, "The Automatic Integration of Ordinary Differential Equations," *Comm. ACM* **14,** 176-179(1971).

[16] W.B. Gragg, "Repeated Extrapolation to the Limit in the Numerical Solution of Ordinary Differential Equations," Thesis, UCLA (1963).

[17] W.B. Gragg, "On Extrapolation Algorithms for Ordinary Initial Value Problems" *SIAM J. Num. Anal.* **2,** 384-403(1965).

[18] W.B. Gragg, "Lecture Notes on Extrapolation Methods," presented at the SIAM National Meeting, Washington, June 1971, and at the Conference on Ordinary Differential Equations, Dundee, Scotland, August, 1971.

[19] E. H. Grosse, "Tensor Spline Approximation," **Linear Algebra and its Applications**, 34, 29-41 (1980).

[20] Ivar Gustafsson, "A Class of First Order Factorization Methods," **BIT**, 14, 142-156 (1978).

[21] B.W. Kernighan, "RATFOR - A Preprocessor for a Rational Fortran," *Software-Practice and Experience* **5,** 395-406(1975).

[22] R.D. Richtmeyer and K.W. Morton, **Difference Methods for Initial Value Problems,** Interscience, New York, 1967.

[23] N.L. Schryer, "An Extrapolation Step-Size and Order Monitor for use in Solving Differential Equations," Proceedings ACM National Meeting, San Diego, 1974.

[24] N.L. Schryer, "An Extrapolation Step-Size and Order Monitor for use in Solving Differential Equations," in preparation.

[25] N.L. Schryer, "A Tutorial on Galerkin's Method, using B-splines, for Solving Differential Equations," Bell Laboratories Computing Science Technical Report #52, 1976.

[26] N.L. Schryer, "Partial Differential Equations in One Space Variable", Bell Laboratories Computing Science Technical Report #115, 1984.

[27] M. Schultz, "*L^∞-Multivariate Approximation Theory*", *SIAM Journal on Numerical Analysis,* **6**, 161-183(1969).

[28] B. P. Sommeijer and P. J. van der Houven, "Algorithm 621. Software with Low Storage Requirements for Two-Dimensional Parabolic Differential Equations," *ACM Trans. on Math. Software* **10,** 378-396(1985).

[29] D. K. Melgaard and R.F. Sincovec "General Software for Two-Dimensional Nonlinear Partial Differential Equations," *ACM Trans. on Math. Software* **7,** 106-125(1981).

[30] H.J. Stetter, "Asymptotic Expansions for the Error of Discretization Algorithms for Non-Linear Functional Equations," *Numer. Math.* **7,** 18-31(1965).

[31] G. Strang and G. Fix, **An Analysis of the Finite Element Method,** Prentice-Hall, New York, 1973.

[32] Lars Wahlbin, "Error Estimates for a Galerkin Method for a Class of Model Equations for Long Waves," *Numer. Math.* **23,** 289-303(1975).

[33] L.O. Wilson, "Numerical Simulation of Gate Oxide Thinning in MOS Devices," *J. Electrochem. Soc.,* **129,** 831-837(1982).

[34] L. O. Wilson, "Lateral Epitaxial Growth over Oxide", to be submitted to the *J. of the Electrochem. Soc.*

Appendix 1

B-splines

The entire solution process is affected by the representation of the approximate numerical solution. Specifically, we would like to choose a space of functions, from which to obtain the element closest to the solution. This space should have several nice properties, including being (1) easy to work with and (2) capable of approximating the solution accurately.

Such a representation exists - expansion in B-splines of order k [1,2,3,8,27]. The rest of this brief tutorial on splines will discuss the one-dimensional case. See [27] for a complete discussion of the multi-dimensional case. This is a method for representing functions by piecewise polynomials, that is, polynomials of degree $k-1$ or less over each sub-interval of a mesh or grid. Here the integer k is any number $k \geq 2$ the user desires. The piecewise polynomial representation is required to satisfy certain continuity restrictions at the end points of each mesh sub-interval. Specifically, let $\pi = \{x_1, \cdots, x_N\}$, where $L = x_1 \leq x_2 \leq \cdots \leq x_N = R$, be a **grid** on the interval (L,R). Let m_i be the **multiplicity** of x_i, or the number of times x_i appears in the list π. The space of B-splines of **order** k on the mesh π is defined to be the collection of all functions f

(A1.1) that are polynomials of degree $<k$ on each interval (x_i, x_{i+1}) for $i=1,\ldots,N-1$,

(A1.2) for which $d^{k-1-m_i} f(x_i) / dx^{k-1-m_i}$ exists and is continuous at each x_i, for $i=1,\ldots,N$, when viewed as a function defined only on $[L,R]$, and

(A1.3) that have $f \equiv 0$ outside $[L,R]$.

The multiplicity m_i of a point x_i is restricted to be in the range $1 \leq m_i \leq k$. For $m_i=1$ we have $d^{k-2} f / dx^{k-2}$ continuous at x_i. This is the most continuity that can be imposed at x_i without making f a polynomial of degree $k-1$ on (x_{i-1}, x_{i+1}). For $m_i = k$ the condition that $d^{-1} f / dx^{-1}$ be continuous is interpreted to mean that f is continuous from the right (but not necessarily from the left) at $x = x_i$, for $x_i < R$, and continuous from the left if $x_i = R$. This means that B-splines are continuous at the end points of the mesh when viewed as functions defined only on $[L,R]$. This collection of functions is denoted by $B_{\pi,k}$. These $B_{\pi,k}$ spaces have nice approximation properties, as summed up by deBoor [1,2], in the case when $m_1 = k = m_N$:

Theorem 1

Let f be any function with $f^{(0)}$ through $f^{(k)}$ continuous on $[L,R]$, where $f^{(j)}$ denotes the j^{th} derivative of f. Let $h = \underset{i=1,\ldots,N-1}{Max} | x_{i+1} - x_i |$ be the largest mesh interval length. Then there is an element g of $B_{\pi,k}$ so that

$$\| f^{(j)}(x) - g^{(j)}(x) \| \leq C(k,f) \ h^{k-j}$$

for $0 \leq j \leq k$, where $C(k,f)$ represents a constant that depends only upon k and f, but not h.

That is, as $h \to 0$, the error in the best B-spline approximation to f goes to zero like h^k; the error in its derivative behaves like h^{k-1}; etc.

Note that this theorem makes no assumption about the relative spacing of the mesh points of π to get h^k error. In many problems, the ability to grade the mesh with B-splines and still get h^k error is a decided advantage.

In practice, k is usually taken to be $4, 6, 8$ or even 10, depending on what the function f looks like and how much accuracy is desired. k is usually, but not always, taken to be even due to the natural way in which such splines arise and their smoothing properties when used to approximate functions described by discrete data [2]. Typically, the more accuracy desired, the larger the value of k should be. For example, if $k=8$ and the mesh length h is halved, then Theorem 1 indicates that the error should decrease by a factor of $2^8 = 256$. However, the work needed to solve a problem using `TTGR` is $O(Nk^4)$. Thus, a $k=8$ solution will cost 16 times as much as a $k=4$ solution for the *same* mesh. But if the error is held constant, the number of mesh points needed is less for higher than lower order. Hence, the optimal k results from minimizing $O(N_k k^4)$, where N_k is the number of mesh points needed to solve the problem to the desired accuracy using a k order B-spline. This optimization is highly problem dependent.

A computationally convenient basis exists for the spaces $B_{\pi,k}$. The dimension of $B_{\pi,k}$ is $N-k$ and the basis consists of elements $B_i(x)$, $i = 1, \ldots, N-k$. A complete description of the B_i is given in [8] and [3]. Briefly, when the multiplicities of the first and last mesh points are both k, so that

$$x_1 = \cdots = x_k$$

and

$$x_{N-k+1} = \cdots = x_N,$$

the main properties of the $B_i(x)$ follow

(A1.4) Each B_i is non-zero only on $[x_i, x_{i+k}]$ and is identically zero elsewhere, as well as at $x_1, \ldots, x_{i-1}$ and $x_{i+k+1}, \ldots, x_N$, even if they are in $[x_i, x_{i+1}]$.

(A1.5) The sum $B_1(x) + \cdots + B_{N-k}(x)$ is identically one.

(A1.6) Each B_i obeys $0 \le B_i(x) \le 1$ everywhere and possesses only one maximum.

The convergence result of Theorem 1 is independent of the multiplicities m_i of the interior points x_i ($k < i \le N-k$) of the mesh. Usually, for smooth functions f, $m_i = 1$ is taken for all these interior (that is, strictly between L and R) mesh points.

The end points of the mesh typically have multiplicity k since the function f usually has $f(L) \ne 0$ and $f(R) \ne 0$, and the elements of $B_{\pi,k}$ cannot be non-zero at L and R, unless $m_1 = k = m_N$ because of (A1.2) and (A1.3). Indeed, relations (A1.2)-(A1.5) show that the only B_i that are not zero at L and R are B_1 and B_{N-k}, and these values are simply $B_1(L) = 1$ and $B_{N-k}(R) = 1$.

If the function f has a discontinuity in its j^{th} derivative, but not its $(j-1)^{th}$, at x_i, then $m_i = k-j$ is chosen because this allows the elements of $B_{\pi,k}$ to have the same behavior. If a smaller multiplicity were chosen, the j^{th} derivative of all the elements of $B_{\pi,k}$ would be continuous at x_i, and the best fit to f from $B_{\pi,k}$ would not be very good at x_i.

Another important property of B-splines is their numerical stability or *condition.* Since any B-spline f is of the form $f = \sum_{i=1}^{N-k} a_i B_i$ and each B_i obeys $0 \le B_i \le 1$, we see that if $\|f\|$ is small compared with $\|\mathbf{a}\|$, then many significant digits are lost when computing f from $a_1, \cdots, a_{N-k}$ in floating-point arithmetic. Specifically,

$$d \le Log_{10}(\|\mathbf{a}\| / \|\sum_{i=1}^{N-k} a_i B_i\|) \tag{A1.7}$$

decimal digits are lost, due to cancellation, in evaluating f. In [2] de Boor shows that

$$\|\sum_{i=1}^{N-k} a_i B_i\| \ge C_k \|\mathbf{a}\| \tag{A1.8}$$

where C_k is a constant depending *only* upon k, and therefore

$$d \le Log_{10}(C_k^{-1}).$$

In particular, he shows for a uniform mesh, one where all the mesh intervals have the same length, that

$$C_k \approx 10^{-k/5} . \tag{A1.9}$$

Thus, when evaluating a B-spline defined on a uniform, or nearly uniform, mesh, we would expect to lose no more than about $k/5$ decimal digits. This is a very satisfactory result since it shows that, at least for uniform meshes, the conditioning of the B-spline basis is independent of the size of the mesh.

The interesting spline facts are, in summary,

$$B_i \geq 0$$
$$B_1 + \cdots + B_{N-k} \equiv 1$$
$$B_1(x_1) = 1 = B_{N-k}(x_N)$$
$$\int_{x_1}^{x_N} B_i \, dx = \frac{x_{i+k} - x_i}{k}.$$

Also, if we let i be such that x is in $[x_i, x_{i+1})$, or $i = N-k$ if $x = x_N$, then

$$\sum_{j=1}^{N-k} a_j B_j(x) \equiv \sum_{j=i+1-k}^{i} a_j B_j(x)$$

If we let

$$f \equiv \sum_{j=1}^{N-k} a_j B_j(x)$$

then we have

$$f'(L) = \frac{k-1}{x_{k+1} - x_1}(a_2 - a_1)$$

and

$$f'(R) = \frac{k-1}{x_{nx} - x_{nx-k}}(a_{nx-k} - a_{nx-k-1})$$

Appendix 2

Extrapolation.

The problem treated in [5] and [16,17] is the numerical solution of the canonical form **ode** initial value problem

$$\frac{d\mathbf{x}}{dt} = \mathbf{f}(t,\ \mathbf{x}) \qquad a \le t$$

$$\mathbf{x}(a) = \mathbf{x}_a \tag{A2.1}$$

where $\mathbf{f}(t,\ \mathbf{x})$ is some smooth vector-valued function of t and $\mathbf{x}$. A brief outline of the ideas developed in those papers follows.

There are many basic differencing schemes for solving (A2.1), such as Gragg's modified mid-point rule [16,17], backwards-Euler methods and Crank-Nicholson [13,22]. Most of these methods have the property [30] that if they take N time-steps $h = (t_1 - t_0)/N$ to go from t_0 to t_1 and result in an approximation to $\mathbf{x}(t_1)$ which we shall denote by $\mathbf{T}(h)$, then

$$\mathbf{T}(h) = \mathbf{x}(t_1) + \sum_{j=1}^{\infty} \mathbf{T}_j h^{j\gamma} \tag{A2.2}$$

where γ is a positive constant depending on the basic difference scheme used, and the $\mathbf{T}_j$ are unknown constant vectors independent of h. For Gragg's modified mid-point rule or Crank-Nicholson $\gamma = 2$ and for backwards-Euler methods $\gamma = 1$.

Let a sequence of h's be defined by

$$h_i = h_0/N_i, \qquad i = 1,\ 2,\ 3, \cdots . \tag{A2.3}$$

where $h_0 = t_1 - t_0$ and the N_i form a monotone increasing sequence of positive integers. Bulirsch and Stoer showed in [4] that given an operator $\mathbf{T}(h)$ satisfying (A2.2), and such a sequence h_i, then the value at $h=0$ of the polynomial of degree m that interpolates $\mathbf{T}(h_i)$ for $i=0, \cdots ,m$, is given by $\mathbf{T}_m^0$ determined from the recursion

$$\mathbf{T}_0^i = \mathbf{T}(h_i) \qquad \text{for} \quad 0 \le i \le m$$

$$\mathbf{T}_k^i = \mathbf{T}_{k-1}^{i+1} + (\mathbf{T}_{k-1}^{i+1} - \mathbf{T}_{k-1}^i)/\left\{(h_i/h_{i+k})^\gamma - 1\right\} \tag{A2.4}$$

for $0 \le i \le m-k$ and $1 \le k \le m$. If the T_k^i are organized into a **tableau** of the form

$$\begin{array}{lllllll}
\mathbf{T}(h_0) = & \mathbf{T}_0^0 & & & & & \\
 & & \mathbf{T}_1^0 & & & & \\
\mathbf{T}(h_1) = & \mathbf{T}_0^1 & & \mathbf{T}_2^0 & & & \\
 & & \mathbf{T}_1^1 & & \mathbf{T}_3^0 & & \\
\mathbf{T}(h_2) = & \mathbf{T}_0^2 & & \mathbf{T}_2^1 & & \mathbf{T}_4^0 & \\
 & & \mathbf{T}_1^2 & & \mathbf{T}_3^1 & & \mathbf{T}_5^0 \\
\mathbf{T}(h_3) = & \mathbf{T}_0^3 & & \mathbf{T}_2^2 & & \mathbf{T}_4^1 & \\
 & & \mathbf{T}_1^3 & & \mathbf{T}_3^2 & & \\
\mathbf{T}(h_4) = & \mathbf{T}_0^4 & & \mathbf{T}_2^3 & & & \\
 & & \mathbf{T}_1^4 & & & & \\
\mathbf{T}(h_5) = & \mathbf{T}_0^5 & & & & &
\end{array}$$

then (A2.4) expresses each element of the k^{th} column $(k>0)$ in terms of its two neighbors in column $k-1$. A similar result is established for interpolation by rational functions [4].

It is also possible to estimate the error in each element of the tableau [6]. For sufficiently small h_0, [6] shows that

$$|\mathbf{T}_k^i - \mathbf{T}| \approx \left[1 + \frac{1}{(h_i/h_{i+k+1})^\gamma - 1}\right] |\mathbf{T}_k^{i+1} - \mathbf{T}_k^i| \qquad \text{(A2.5)}$$

and we can estimate the error $\varepsilon_k^i = |\mathbf{T}_k^i - \mathbf{T}|$ in $\mathbf{T}_k^i$. We also know from [6] that for sufficiently small h_0,

$$\varepsilon_k^i = h_0^\beta \mathbf{d}_k (h_i \cdots h_{i+k})^\gamma \qquad \text{(A2.6)}$$

where the $\mathbf{d}_k$ are constant vectors, γ is the order of the basic process being extrapolated and β is a positive constant. When extrapolating Gragg's modified mid-point rule or Crank-Nicholson, $\gamma = 2$ and $\beta = 1$. When extrapolating a Backwards-Euler time differencing process, $\gamma = 1 = \beta$. Thus, we can both estimate the accuracy of each element in the tableau and tell how rapidly each column in the tableau is converging.

In (A2.4) m is called the **level** of extrapolation, while from (A2.6) we see that the **order** in column k is $(k+1)\gamma$. Thus, by extrapolating the results of a basic ordinary differential equation solver, a process of arbitrarily high order can be obtained. The value $h_0 = t_1 - t_0$ is referred to as the **time-step** while the h_i are called **sub-steps.** Extrapolation approximates the $\mathbf{x}(t_1)$ values accurately, but does not accurately approximate $\mathbf{x}(t+nh_i)$ for $0<n<N$.

Appendix 3

Wish-List

This section describes several improvements that could be made in TTGR. The improvements range from better human engineering (easier use) to making the algorithm more efficient and extending it to solve more general problems.

Periodic Boundary Conditions

Since the **pde**s are defined on a rectangle, and mapping is sometimes used to work on more general domains, it would be good to allow periodic boundary conditions. This will be done in the next version of the package.

Speed

The current assembly phase (making the Jacobian) and solution processes have been sped up by factors of 3 and more. This unpolished code will be ready for the next version.

One Dimensional Pde and Ode Coupling

In the spirit of the one-dimensional **pde** solver POST [26], it would be good if TTGR allowed solution of problems involving one and two dimensional **pde**s coupled to **ode**s. This is important for several reasons. The first is that it allows solution of some real world problems, like those of [33] and [34]. The second reason is just plain mathematical completeness: without such coupling, there are problems that cannot even be posed, let alone solved.

Faster Solution Times

The block sparse-matrix package of Kent Smith offers a factor of n_u improvement in run-time on vector machines like the Cray. This will probably be an option in the next version.

Multi-grid schemes offer optimal run-time and space solution methods. The robustness and applicability of these schemes is well worth exploring, but will take some time: such codes are complex, but probably worth the work.

Appendix 4

Examples - Programs

The program examples given below are intended to both illustrate the use of TTGR and provide prototypes for a prospective user. Anyone contemplating using TTGR would be well advised to pick an example program that invokes those capabilities of TTGR the intended problem will require, and type it in (or obtain a copy of the example code from the authors). After running the example and confirming the correctness of the program, the AF and BC subroutines specifying the **pde**-**bc** may simply be altered to solve the user's problem. This progression makes it much more likely that the user will easily produce a correct program unit for the problem at hand.

The examples are chosen to require small memory and run-time resources. This is to make their running on small machines, like Vaxen, not too onerous a chore for folks installing and testing the package.

The examples are taken in sequence from section 4 where they were formulated and analyzed. This section is only intended to show how to program the solution of the formulations given in section 4. The user must have read section 5 describing the TTGR software before proceeding in this appendix, otherwise the reading will be dark and obscure.

Before invoking TTGR the user must

- Make a B-spline mesh.
- Make initial conditions for the B-spline coefficients **U** in (3.1).
- Write subroutines
  - AF - to evaluate **a** and **f** in (2.1).
  - BC - to evaluate **b** in (2.2).
  - HANDLE - to output (print) the solution results.

The subroutine writing will be amply illustrated later in this section. The creation of a mesh and initial conditions (**ic**s) for **u** are now briefly described.

Mesh Making

The PORT Library [14] has several B-spline mesh generation subroutines available. There is UMB for generating uniform B-spline meshes on a given interval. For creating B-spline meshes that are the union of uniform meshes over basic contiguous intervals there are LUMB and PUMB. LUMB uses the same number of mesh points in each basic interval and PUMB allows that number to vary by interval.

If you find yourself using more than 50 to 100 mesh points, in any direction, think carefully about using LUMB to make the mesh more carefully tailored to the solution, or use mesh mapping. For example, example 5 below could be solved on a uniform mesh, but it would require roughly **90,000** grid points to do it for $k = 2$. Using the non-uniform mesh scheme given for that problem, 625 do the job nicely. Remember that the run-time and memory requirements of TTGR are proportional to the number of mesh points used. A little thought given to mesh construction can save a lot of computer run-time and memory.

Initial Conditions for u.

There are **ic**s for **u** and these must be converted into **ic**s for **U**, the B-spline coefficients for **u**, see (3.1). The simplest case is when the **ic**s for **u** are a constant. By simply setting all the spline coefficients to that constant, the spline **is** the constant and we are done.

If the **ic**s for u are not constant, then there is TSL2W available, see section 4.

The Examples

All examples reported here were run on a VAX 11/750, equipped with a floating-point accelerator, using single-precision arithmetic, under the UNIX operating system, Research Eighth Edition.

Example 1 - A Simple Heat Equation.

As a simple example of the use of `TTGR`, consider solving the scalar heat equation

$$\begin{aligned} a^{(1)} &= u + u_x + .1\, u_y , \\ a^{(2)} &= u + u_y + .1\, u_x , \\ f &= u_t + u_x + u_y - g(t,x,y) \end{aligned} \qquad \text{(A4.1)}$$

on the unit square, with **bc**s (4.2)

$$\mathbf{b} = u(t,x,y) - t\, x\, y. \qquad \text{(A4.2)}$$

The solution is $u \equiv t\, x\, y$, which can be gotten exactly. The initial conditions are taken to be 0.

The following program unit, written in Ratfor [21], solves (A4.1)-(A4.2) using `TTGR`, with a linear B-spline ($k = 2$) over a spatial mesh consisting of 3 equally spaced, distinct points on $(0,1)$, with the time evolution carried out to 10^{-2} absolute accuracy.

Ratfor is much easier to read and type than standard old `Fortran`. Although we use Ratfor to present the code in the examples, we ship standard `Fortran`.

Ratfor stands for "Rational Fortran" and provides modern control structures over Fortran. Specifically, it allows multiple statements on a line to be separated by semicolons (;), a # begins a comment (hence a comment and an executable statement can be together on the same line), and curly braces ({ and }) are used to delimit blocks of code. This last feature removes the need for statement labels on Do loops, for example. There is "syntactic sugar" in the form of "<" for ".lt.", "&" for ".and." and " | " for ".or.". Finally, if a line ends in one of ", + - * / (= < > & |", the next line is considered a continuation of it.

The main program uses several PORT [14] library subprograms.

- `ISTKIN` initializes the PORT Library stack. In this case, it initializes the stack to 350,000 double precision items, consistent with the declaration for `Ds` in the double precision alias of the stack in the common region `CSTAK`.
- `ENTER` and `LEAVE` bracket blocks of code in which stack allocations are done. The effect is that all allocations made after the `ENTER` but before the `LEAVE` are released by the `LEAVE`.
- `IUMB` makes uniformly spaced B-spline meshes on the stack. It returns a pointer to the mesh.
- `ISTKGT` allocates storage on the Port Library stack. In the example below, `iU = ISTKGT(L,3)` sets the pointer `iU` so that locations `Ws(iU), ..., Ws(iU+L-1)` are available for the B-spline coefficients of the solution.
- `SETR` sets an array to a given Real constant. `SETR` provides the constant **ic**'s (A4.3) via the B-spline coefficients (3.1), since if all the B-spline coefficients are equal to a constant, then the B-spline itself is identically equal to that constant (see Appendix 1).
- `WRAPUP` checks that a run has terminated without errors and prints out the stack space used.

At the end of each time-step the solution is printed out at $x,y = 0,\ \frac{1}{2}$ and 1. The main program is

```
# Main

# To solve the heat equation with solution u == t*x*y,

#   grad . ( U + Ux + .1 * Uy, U + Uy + .1 * Ux ) = Ut + Ux + Uy +g(x,t)

  Real tstart,tstop,dt,Lx,Rx,Ly,Ry
  Real errpar(2)
  Integer Nu,kx,ix,nx, ky,iy,ny,ISTKGT, IUMB,iU,ndx,ndy
  External AF,BC,HANDLE


  Common / CSTAK / Ds(350000); Double Precision Ds
  Real Ws(1000)    # The PORT Library stack and its aliases.
  Real Rs(1000) ; Integer Is(1000) ; Complex Cs(500) ; Logical Ls(1000)
  Equivalence ( Ds(1),Cs(1),Ws(1),Rs(1),Is(1),Ls(1) )

  Call ISTKIN(350000,4)    # Initialize the PORT Library stack length.

  Call ENTER(1)

  Nu = 1

  Lx = 0; Rx = 1
  Ly = 0; Ry = 1

  kx = 2; ky = 2

  ndx = 3; ndy = 3

  tstart = 0; tstop = 1; dt = 1

  errpar(1) = 1e-2; errpar(2) = 1e-4

  ix = IUMB(Lx,Rx,ndx,kx,nx)    # Uniform grid.
  iy = IUMB(Ly,Ry,ndy,ky,ny)    # Uniform grid.

  iU = ISTKGT(Nu*(nx-kx)*(ny-ky),3)    # Space for the solution.
  Call SETR(Nu*(nx-kx)*(ny-ky),0e0,Ws(iU))    # Initial conditions for U.

  Call TTGR (Ws(iU),Nu,kx,Ws(ix),nx, ky,Ws(iy),ny, tstart,tstop, dt,
             AF,BC,
             errpar,
             HANDLE)

  Call LEAVE

  Call WRAPUP

  Stop

  End
```

The dimension statements for the various arguments of the AF, BC and HANDLE subroutines given below are general and thus will not be repeated in subsequent **pde-bc** examples.

Note that since the arrays A, ... , FUyt are set to zero by TTGR before it calls AF, only the active **a** and **f** terms and their derivatives need be computed in AF. The subroutine AF for specifying the **pde** (A4.1) is

```
Subroutine AF(t,x,nx,y,ny,Nu,
              U,Ut,Ux,Uy,Uxt,Uyt,
              a,AU,AUt,AUx,AUy,AUxt,AUyt,
              f,FU,FUt,FUx,FUy,FUxt,FUyt)

Real t,x(nx),y(ny),
          U(nx,ny,Nu),Ut(nx,ny,Nu),Ux(nx,ny,Nu),Uy(nx,ny,Nu),
          Uxt(nx,ny,Nu),Uyt(nx,ny,Nu),
          a   (nx,ny,Nu   ,2), f   (nx,ny,Nu   ),
          AU  (nx,ny,Nu,Nu,2), FU  (nx,ny,Nu,Nu),
          AUt (nx,ny,Nu,Nu,2), FUt (nx,ny,Nu,Nu),
          AUx (nx,ny,Nu,Nu,2), FUx (nx,ny,Nu,Nu),
          AUy (nx,ny,Nu,Nu,2), FUy (nx,ny,Nu,Nu),
          AUxt(nx,ny,Nu,Nu,2), FUxt(nx,ny,Nu,Nu),
          AUyt(nx,ny,Nu,Nu,2), FUyt(nx,ny,Nu,Nu)
Integer nx,ny,Nu

Integer p,q,i

Do i = 1, Nu
  {
  Do q = 1, ny
    {
    Do p = 1, nx
      {
      a(p,q,i,1) = Ux(p,q,i) + .1*Uy(p,q,i) + U(p,q,i)
      a(p,q,i,2) = Uy(p,q,i) + .1*Ux(p,q,i) + U(p,q,i)
      AUx(p,q,i,i,1) = 1; AUy(p,q,i,i,2) = 1
      AUy(p,q,i,i,1) = .1; AUx(p,q,i,i,2) = .1
      AU (p,q,i,i,1) = 1; AU (p,q,i,i,2) = 1

      f(p,q,i) = Ut(p,q,i)+Ux(p,q,i)+Uy(p,q,i)
      FUt(p,q,i,i) = 1; fUx(p,q,i,i) = 1; fUy(p,q,i,i) = 1
      f(p,q,i) = f(p,q,i) + .2*t-x(p)*y(q)
      }
    }
  }

Return

End
```

Note that since the arrays B, ... , BUyt are set to zero by TTGR before it calls BC, only the active **b** terms and their derivatives need be computed in BC. The subroutine BC for specifying the **bc**'s (A4.2) is

```
Subroutine BC(t,x,nx,y,ny,Lx,Rx,Ly,Ry,
              U,Ut,Ux,Uy,Uxt,Uyt,Nu,
              b,bu,but,bux,buy,buxt,buyt)

Real t,x(nx),y(ny),Lx,Rx,Ly,Ry,
          U(nx,ny,Nu),Ut(nx,ny,Nu),Ux(nx,ny,Nu),Uy(nx,ny,Nu),
          Uxt(nx,ny,Nu),Uyt(nx,ny,Nu),
          b(nx,ny,Nu),bu(nx,ny,Nu,Nu),bux(nx,ny,Nu,Nu),buy(nx,ny,Nu,Nu),
          but(nx,ny,Nu,Nu),buxt(nx,ny,Nu,Nu),buyt(nx,ny,Nu,Nu)
Integer nx,ny,Nu

Integer i,j

Do j = 1, ny
  {
  Do i = 1, nx
    {
    bu(i,j,1,1) = 1; b(i,j,1) = U(i,j,1)-t*x(i)*y(j)
    }
  }

Return

End
```

The following output subroutine simply evaluates and prints $u(t,x,y)$, for $x,y = 0, \frac{1}{2}$, and 1, at the end of each successful time-step. The dimension statement for the various arguments is for arbitrary input and thus will not be repeated in subsequent examples.

Three PORT Library routines are used

- `I1MACH` determines the standard output unit number, `I1MACH`(2).
- `TSD1` evaluates a spline, given the mesh and the coefficients. See section 4.
- `ILUMD` generates a list of distinct points from a basic mesh by inserting a given number of points between the basic points.

Also, two `Common` regions from `TTGR` are used to provide, magically, the meshes for x and y and `Nu`. This is because the fixed calling sequence for the output routine from `IODE` doesn't currently allow the passing of such extra information. So we pass it under the table.

```
     Subroutine HANDLE(t0,U0,t,U,Nv,dt,tstop)

     Real t0,U0(Nv),t,U(Nv),dt,tstop
     Integer Nv

     Common / A7TGRM / kx,ix,nx, ky,iy,ny; Integer kx,ix,nx, ky,iy,ny

     Common / A7TGRP / errpar(2), Nu,mxq,myq; Real errpar; Integer Nu,mxq,myq

     If ( t0 == t )
       {
       iwunit = 6
       Write(iwunit,9000) t
    9000 Format(" Restart for t =",1p4e10.2)
       Return
       }

     Call GERR(kx,ix,nx, ky,iy,ny, U,Nu, t)   # Print results.

     Return

     End
```

where the procedure GERR is used to print out the results

```
  Subroutine GERR(kx,ix,nx, ky,iy,ny, U,Nu, t)

# To print the solution at each time-step.

  Real U(1),t    # U(nx-kx,ny,ky,Nu).
  Integer kx,ix,nx, ky,iy,ny,Nu

  Common / CSTAK / Ds(500); Double Precision Ds
  Real Ws(1000)    # The PORT Library stack and its aliases.
  Real Rs(1000) ; Integer Is(1000) ; Complex Cs(500) ; Logical Ls(1000)
  Equivalence ( Ds(1),Cs(1),Ws(1),Rs(1),Is(1),Ls(1) )

  Integer i,I1MACH,ISTKGT,
          KA(2),ITA(2),NTA(2),IXA(2),NXA(2),MA(2),iFA,
           ILUMD,ixs,iys,nxs,nys

  Call ENTER(1)

# Find the solution at 2 * 2 points / mesh rectangle.

  ixs = ILUMD(Ws(ix),nx,2,nxs)    # x search grid.
  iys = ILUMD(Ws(iy),ny,2,nys)    # y search grid.

  KA(1) = kx; KA(2) = ky; ITA(1) = ix; ITA(2) = iy; NTA(1) = nx; NTA(2) = ny
  IXA(1) = ixs; IXA(2) = iys; NXA(1) = nxs; NXA(2) = nys
  MA(1) = 0; MA(2) = 0    # Get solution.

  iFA = ISTKGT(nxs*nys,3)    # Approximate solution values.

  Call TSD1(2,KA,Ws,ITA,NTA,U, Ws,IXA,NXA,MA, Ws(iFA))    # Evaluate them.

  iwunit = I1MACH(2)
  Write(iwunit,9001) t,(Ws(i),i = iFA, iFA+nxs*nys-1 )
 9001 Format(" U(",1p1e10.2,",.,.) =",((1p5e10.2/20x,1p4e10.2)))

  Call LEAVE

  Return

  End
```

The output of the above program unit is

```
U(  1.00e+00,.,.) =  0.00e+00  5.14e-17  5.87e-09  0.00e+00  2.50e-01
                     5.00e-01  0.00e+00  5.00e-01  1.00e+00
used      825 /   700000 of the stack allowed.
```

The run-time for the above was 5.2 seconds.

A skeptic might observe that it is difficult to determine that the above output is in fact exact. Well, since the exact solution of the problem is known, the program may also check the accuracy of the numerical solution. The procedure GERR below checks the error rather than just evaluate the solution

```
    Subroutine GERR(kx,ix,nx, ky,iy,ny, U,Nu, t)

  # To get and print the error at each time-step.

    Real U(1),t    # U(nx-kx,ny0ky,Nu).
    Integer kx,ix,nx, ky,iy,ny,Nu

    Common / CSTAK / Ds(500); Double Precision Ds
    Real Ws(1000)    # The PORT Library stack and its aliases.
    Real Rs(1000) ; Integer Is(1000) ; Complex Cs(500) ; Logical Ls(1000)
    Equivalence ( Ds(1),Cs(1),Ws(1),Rs(1),Is(1),Ls(1) )

    Real errU
    Integer i,I1MACH,ISTKGT,
            KA(2),ITA(2),NTA(2),IXA(2),NXA(2),MA(2),iFA, ILUMD,ixs,iys,nxs,nys,iEwe

    Call ENTER(1)

  # Find the error in the solution at 2*kx * 2*ky points / mesh rectangle.

    ixs = ILUMD(Ws(ix),nx,2*kx,nxs)    # x search grid.
    iys = ILUMD(Ws(iy),ny,2*ky,nys)    # y search grid.
    iEwe = ISTKGT(nxs*nys,3)    # U search grid values.

    Call EWE(t,Ws(ixs),nxs,Ws(iys),nys,Ws(iEwe),Nu)    # The exact solution.

    KA(1) = kx; KA(2) = ky; ITA(1) = ix; ITA(2) = iy; NTA(1) = nx; NTA(2) = ny
    IXA(1) = ixs; IXA(2) = iys; NXA(1) = nxs; NXA(2) = nys
    MA(1) = 0; MA(2) = 0    # Get solution.

    iFA = ISTKGT(nxs*nys,3)    # Approximate solution values.

    Call TSD1(2,KA,Ws,ITA,NTA,U, Ws,IXA,NXA,MA, Ws(iFA))    # Evaluate them.
    errU = 0    # Error in solution values.
    Do i = 1, nxs*nys
      {
      errU = Max(errU,Abs(Ws(iEwe+i-1)-Ws(iFA+i-1)))
      }

    iwunit = I1MACH(2)
    Write(iwunit,9001) t, errU
   9001 Format(" error in U(.,",1p1e10.2,") =", 1p4e10.2)

    Call LEAVE

    Return

    End
```

and the procedure EWE below evaluates the exact solution at any position in time and space.

```
    Subroutine EWE(t,x,nx,y,ny,U,Nu)

  # The exact solution.

    Real t,x(nx),y(ny),U(nx,ny,Nu)
    Integer nx,ny,Nu

    Integer p,i,j

    Do p = 1, Nu
      {
      Do i = 1, nx
        {
        Do j = 1, ny
          {
          U(i,j,p) = t*x(i)*y(j)
          }
        }
      }

    Return

    End
```

The above program unit gives

```
error in U(.,  1.00e+00) =  5.87e-09
used      825 /   700000 of the stack allowed.
```

and we can see that (A4.1)-(A4.2) has indeed been solved to within rounding error. The run-time for the above example was 5.3 seconds.

Example 2 - A Coupled System of pdes.

The solution of a coupled system of **pde**'s is now illustrated. The **pde** is

$$
\begin{aligned}
&a_1^{(1)} = u_{1x}, \quad a_2^{(1)} = u_{2x}, \\
&a_1^{(2)} = u_{1y}, \quad a_2^{(2)} = u_{2y}, \\
&f_1 = u_{1t} + u_1\, u_2 - g_1, \quad f_2 = u_{2t} + u_1\, u_2 - g_2
\end{aligned}
\tag{A4.3}
$$

with **bc**s

$$
b_1 = u_1(t,x,y) - e^{t(x-y)} \qquad \text{and} \qquad b_2 = u_2(t,x,y) - e^{-t(x-y)} \tag{A4.4}
$$

The solution is $u_1 \equiv e^{t(x-y)}$ and $u_2 \equiv e^{-t(x-y)}$.

The following program solves (A4.3)-(A4.4) using `TTGR`, with a cubic B-spline, over a spatial mesh consisting of 3 equally spaced, distinct points on $(0,1)$, with the time-evolution carried out to 10^{-2} absolute accuracy. The error at each time-step is printed out to confirm the accuracy of the computed solution. The main program is

```
# Main

# To solve two coupled, nonlinear heat equations.

#   U1 sub t = div . ( U1x, U1y ) - U1*U2 + g1
#   U2 sub t = div . ( U2x, U2y ) - U1*U2 + g2

  Real tstart,tstop,dt,Lx,Rx,Ly,Ry
  Real errpar(2)
  Integer Nu,kx,ix,nx, ky,iy,ny,ISTKGT, IUMB,iU,ndx,ndy
  External AF,BC,HANDLE


  Common / CSTAK / Ds(350000); Double Precision Ds
  Real Ws(1000)    # The PORT Library stack and its aliases.
  Real Rs(1000) ; Integer Is(1000) ; Complex Cs(500) ; Logical Ls(1000)
  Equivalence ( Ds(1),Cs(1),Ws(1),Rs(1),Is(1),Ls(1) )

  Call ISTKIN(350000,4)    # Initialize the PORT Library stack length.

  Call ENTER(1)

  Nu = 2

  Lx = 0; Rx = +1
  Ly = 0; Ry = +1

  kx = 4; ky = 4

  ndx = 3; ndy = 3

  tstart = 0; tstop = 1; dt = 1e-2

  errpar(1) = 1e-2; errpar(2) = 1e-4

  ix = IUMB(Lx,Rx,ndx,kx,nx)    # Uniform grid.
  iy = IUMB(Ly,Ry,ndy,ky,ny)    # Uniform grid.

  iU = ISTKGT(Nu*(nx-kx)*(ny-ky),3)    # Space for the solution.
  Call SETR(Nu*(nx-kx)*(ny-ky),1e0,Ws(iU))

  Call TTGR (Ws(iU),Nu,kx,Ws(ix),nx, ky,Ws(iy),ny, tstart,tstop, dt,
             AF,BC,
             errpar,
             HANDLE)

  Call LEAVE

  Call WRAPUP

  Stop

  End
```

The only change in the subroutine AF of the last example is the code for specifying the **pde**

```
Integer p,q

Do q = 1, ny
  {
  Do p = 1, nx
    {
    a(p,q,1,1) = Ux(p,q,1); AUx(p,q,1,1,1) = 1
    a(p,q,1,2) = Uy(p,q,1); AUy(p,q,1,1,2) = 1

    f(p,q,1) = Ut(p,q,1) + U(p,q,1)*U(p,q,2)
    FU(p,q,1,1) = U(p,q,2); FU(p,q,1,2) = U(p,q,1); FUt(p,q,1,1) = 1

    a(p,q,2,1) = Ux(p,q,2); AUx(p,q,2,2,1) = 1
    a(p,q,2,2) = Uy(p,q,2); AUy(p,q,2,2,2) = 1

    f(p,q,2) = Ut(p,q,2) + U(p,q,1)*U(p,q,2)
    FU(p,q,2,1) = U(p,q,2); FU(p,q,2,2) = U(p,q,1); FUt(p,q,2,2) = 1

    f(p,q,1) = f(p,q,1) - ( Exp(t*(x(p)-y(q)))*( x(p)-y(q) - 2*t*t ) + 1 )
    f(p,q,2) = f(p,q,2) - ( Exp(t*(y(q)-x(p)))*( y(q)-x(p) - 2*t*t ) + 1 )
    }
  }
```

The only change in the subroutine BC of the previous example is the code for specifying the **bc**s

```
Do j = 1, ny
  {
  Do i = 1, nx
    {
    bu(i,j,1,1) = 1; b(i,j,1) = U(i,j,1)-Exp(t*(x(i)-y(j)))

    bu(i,j,2,2) = 1; b(i,j,2) = U(i,j,2)-Exp(t*(y(j)-x(i)))
    }
  }
```

The HANDLE subroutine simply checks the accuracy of the computed solution,

```
  Subroutine HANDLE(t0,U0,t,U,Nv,dt,tstop)

  Real t0,U0(Nv),t,U(Nv),dt,tstop
  Integer Nv

  Common / A7TGRM / kx,ix,nx, ky,iy,ny; Integer kx,ix,nx, ky,iy,ny

  Common / A7TGRP / errpar(2), Nu,mxq,myq; Real errpar; Integer Nu,mxq,myq

  Common / CSTAK / Ds(500); Double Precision Ds
  Real Ws(1000)    # The PORT Library stack and its aliases.
  Real Rs(1000) ; Integer Is(1000) ; Complex Cs(500) ; Logical Ls(1000)
  Equivalence ( Ds(1),Cs(1),Ws(1),Rs(1),Is(1),Ls(1) )

  Real errU
  Integer i,I1MACH,ISTKGT,j,
          KA(2),ITA(2),NTA(2),IXA(2),NXA(2),MA(2),iFA,
           ILUMD,ixs,iys,nxs,nys,iEwe

  If ( t0 == t )
    {
    iwunit = 6
    Write(iwunit,9000) t
 9000 Format(" Restart for t =",1p4e10.2)
    Return
    }

  Call ENTER(1)

# Find the error in the solution at 2*kx * 2*ky points / mesh rectangle.

  ixs = ILUMD(Ws(ix),nx,2*kx,nxs)    # x search grid.
  iys = ILUMD(Ws(iy),ny,2*ky,nys)    # y search grid.
  iEwe = ISTKGT(Nu*nxs*nys,3)    # U search grid values.

  Call EWE(t,Ws(ixs),nxs,Ws(iys),nys,Ws(iEwe),Nu)    # The exact solution.

  KA(1) = kx; KA(2) = ky; ITA(1) = ix; ITA(2) = iy; NTA(1) = nx; NTA(2) = ny
  IXA(1) = ixs; IXA(2) = iys; NXA(1) = nxs; NXA(2) = nys
  MA(1) = 0; MA(2) = 0    # Get solution.

  iFA = ISTKGT(nxs*nys,3)    # Approximate solution values.

  Do j = 1, Nu
    {
     Call TSD1(2,KA,Ws,ITA,NTA,U(1+(j-1)*(nx-kx)*(ny-ky)),
               Ws,IXA,NXA,MA, Ws(iFA))    # Evaluate them.
    errU = 0    # Error in solution values.
    Do i = 1, nxs*nys
      {
      errU = Max(errU,Abs(Ws(iEwe+i-1+(j-1)*nxs*nys)-Ws(iFA+i-1)))
      }

    iwunit = I1MACH(2)
```

```
    Write(iwunit,9001) t,j, errU
9001 Format(" error in U(.,",1p1e10.2,",",I2,") =", 1p4e10.2)
    }

  Call LEAVE

  Return

  End
```

where the following body for the subroutine `EWE` computes the exact solution.

```
Integer p,i,j

Do p = 1, Nu
  {
  Do i = 1, nx
    {
    Do j = 1, ny
      {
      U(i,j,p) = Exp( (-1)**(p+1) * t * ( x(i) - y(j) ) )
      }
    }
  }
```

The output of the above program unit is

```
error in U(.,  1.00e-02, 1) =  7.63e-06
error in U(.,  1.00e-02, 2) =  7.57e-06
error in U(.,  2.11e-01, 1) =  3.85e-04
error in U(.,  2.11e-01, 2) =  3.85e-04
error in U(.,  8.76e-01, 1) =  1.80e-03
error in U(.,  8.76e-01, 2) =  1.80e-03
error in U(.,  1.00e+00, 1) =  4.55e-04
error in U(.,  1.00e+00, 2) =  4.55e-04
used     9398 /   700000 of the stack allowed.
```

The run-time for the above example was 431.5 seconds.

Note that the solution of this problem is not exactly representable as a spline. Thus, it is the first non-trivial example use of `TTGR`.

Example 3 - Interfaces.

Consider the **pde**

$$
\begin{aligned}
a^{(1)} &= \kappa\,(x,y)\ \mathbf{u}_x \\
a^{(2)} &= \kappa\,(x,y)\ \mathbf{u}_y \\
f &= \mathbf{u}_t - g
\end{aligned}
\qquad \text{(A4.5)}
$$

where

$$
\kappa \equiv \begin{cases} 1 0 \le y \le 1 \\ 1/2 1 < y \le 2, \\ 1/3 2 < y \le 3 \end{cases}
$$

with **bc**s on the bottom and top

$$\mathbf{b} = u_y \tag{A4.6a}$$

and those on the sides

$$\mathbf{b} = \mathbf{u} - s(t,x,y) \tag{A4.6b}$$

The following program solves (A4.5)-(A4.6) using TTGR, with a linear B-spline ($k = 2$) over a spatial mesh on $(0,1)$ consisting of 3 equally spaced, distinct points, with the time evolution carried out to roughly 10^{-2} relative accuracy. The error at each time-step is printed out to confirm the accuracy of the numerical solution.

The main program uses two Port Library subroutines

- ILUMB makes a B-spline mesh by putting a uniform mesh on each of a set of contiguous intervals, in this case 3 points each on $[\,0,\,1\,]$, $[\,1,\,2\,]$ and $[\,2,\,3\,]$.
- IMMM makes $\eta = 1$ a mesh point of multiplicity k-1, so that $\mathbf{u}\,(\,x,\,y,\,t\,)$ can have its partials with respect to y discontinuous, see appendix 1. $k-2$ points must be added to the mesh to accomplish this. Since it is dangerous to assume that a fixed dimension array can be added to, the mesh is put on the stack by ILUMB where it may be safely lengthened. The mesh is put on the Port stack and a pointer to it is returned.

The main program is

```
# Main

# To solve the layered heat equation, with kappa = 1, 1/2, 1/3,

#   div . ( kappa(x,y) * grad U ) = Ut + g

  Real tstart,tstop,dt,Lx,Rx,yb(4)
  Real errpar(2)
  Integer Nu,kx,ix,nx, ky,iy,ny,ISTKGT, IUMB, ILUMB,iU,ndx,ndy,IMMM ,i
  External AF,BC,HANDLE


  Common / CSTAK / Ds(350000); Double Precision Ds
  Real Ws(1000)    # The PORT Library stack and its aliases.
  Real Rs(1000) ; Integer Is(1000) ; Complex Cs(500) ; Logical Ls(1000)
  Equivalence ( Ds(1),Cs(1),Ws(1),Rs(1),Is(1),Ls(1) )

  Call ISTKIN(350000,4)    # Initialize the PORT Library stack length.

  Call ENTER(1)

  Nu = 1

  Lx = 0; Rx = 1
  Do i = 1, 4 { yb(i) = i-1 }

  kx = 2; ky = 2

  ndx = 3; ndy = 3

  tstart = 0; tstop = 1; dt = 1

  errpar(1) = 1e-2; errpar(2) = 1e-4

  ix = IUMB(Lx,Rx,ndx,kx,nx)    # Uniform grid.
  iy = ILUMB(yb,4,ndy,ky,ny)    # Uniform grid.
  iy = IMMM (iy,ny,yb(2),ky-1)    # Make mult = ky-1.
  iy = IMMM (iy,ny,yb(3),ky-1)    # Make mult = ky-1.

  iU = ISTKGT(Nu*(nx-kx)*(ny-ky),3)    # Space for the solution.
  Call SETR(Nu*(nx-kx)*(ny-ky),0e0,Ws(iU))

  Call TTGR (Ws(iU),Nu,kx,Ws(ix),nx, ky,Ws(iy),ny, tstart,tstop, dt,
             AF,BC,
             errpar,
             HANDLE)

  Call LEAVE

  Call WRAPUP

  Stop

  End
```

The body of the AF subroutine for specifying the **pde** (A4.5) is

```
Real kappa
Integer p,q,i

Do i = 1, Nu
  {
  Do q = 1, ny
    {
    Do p = 1, nx
      {
      If      ( y(q) < 1 ) { kappa = 1 }
      Else If ( y(q) < 2 ) { kappa = 0.5 }
      Else                 { kappa = 1/3e0 }

      a(p,q,i,1) = kappa*Ux(p,q,i); AUx(p,q,i,i,1) = kappa
      a(p,q,i,2) = kappa*Uy(p,q,i); AUy(p,q,i,i,2) = kappa

      f(p,q,i) = Ut(p,q,i); FUt(p,q,i,i) = 1

      f(p,q,i) = f(p,q,i) - y(q)/kappa

      If ( 1 < y(q) & y(q) < 2 ) { f(p,q,i) = f(p,q,i) + 1 }
      If ( 2 < y(q) & y(q) < 3 ) { f(p,q,i) = f(p,q,i) + 3 }
      }
    }
  }
```

The body of the BC subroutine for specifying the **bc**s (A4.6) is

```
Do j = 1, ny
  {
  Do i = 1, nx
    {
    If ( x(i) == Lx | x(i) == Rx )    # Left or right.
      {
      BUx(i,j,1,1) = 1; B(i,j,1) = Ux(i,j,1)    # Neumann BCs.
      }
    Else If ( y(j) == Ly )    # Bottom.
      {
      B(i,j,1) = U(i,j,1); BU(i,j,1,1) = 1
      }
    Else    # Top.
      {
      B(i,j,1) = U(i,j,1)-6*t; BU(i,j,1,1) = 1
      }
    }
  }
```

There is no change in the HANDLE or GERR subroutines of example 1. The only change in the subroutine EWE, for computing *u*, of example 1 is the code for computing *u*

```
If (      y(j) < 1 ) { U(i,j,p) =  t*y(j) }
Else If ( y(j) < 2 ) { U(i,j,p) = 2*t*y(j)-t }
Else                 { U(i,j,p) = 3*t*y(j)-3*t }
```

The output from this program is

```
error in U(.,  1.00e+00) =  4.77e-07
used     1260 /   700000 of the stack allowed.
```

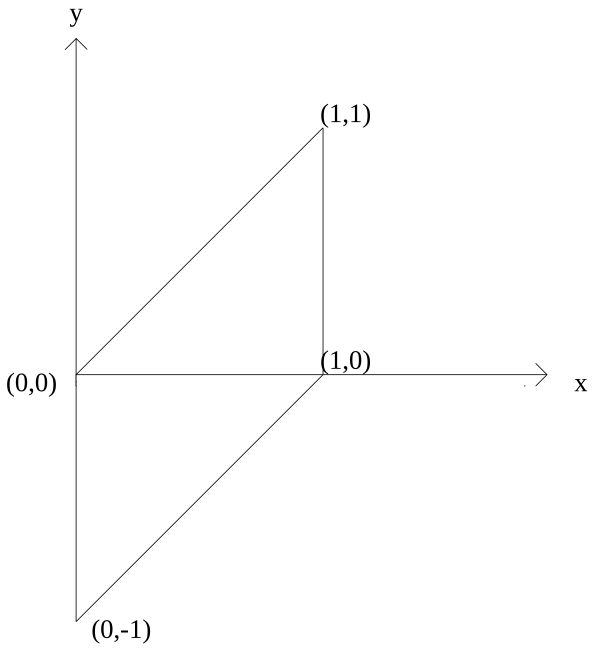
and we see that (A4.5)-(A4.6) has indeed been solved to rounding error. The run-time for the above example was 10.9 seconds.

Example 4 - Non-Rectangular Domains.

Consider the **pde**

$$a^{(1)} = \mathbf{u}_x - \frac{\mathbf{u}_y}{10}$$
$$a^{(2)} = \mathbf{u}_y - \frac{\mathbf{u}_x}{10} \qquad \text{(A4.7)}$$
$$f = \mathbf{u}_t - g$$

on the domain



with **bc**s on the bottom and top

$$\mathbf{b} = u_N - n(x,y) \qquad \text{(A4.8a)}$$

and those on the sides

$$\mathbf{b} = \mathbf{u} - s(t,x,y) \qquad \text{(A4.8b)}$$

where g, n and s are chosen so that the solution is $u = t\,x\,y$.

The following program solves the **pde-bc** combination (A4.7)-(A4.8), using cubic B-splines over a mesh consisting of 3 equally spaced, distinct points on $(0,1)$, with the time-evolution carried out to 10^{-2} absolute accuracy. The error at each time-step is printed out to confirm the accuracy of the computed solution. The main program is

```
# Main

# To solve the linear heat equation

#   grad . ( Ux - 0.1 * Uy , 0.1*Ux +  Uy ) = Ut - x*y

# with solution u == t*x*y on [0,+1]**2, exact for k = 4,
# with tilted top and bottom, normal BCs there.

  Real tstart,tstop,dt,Lx,Rx,Ly,Ry
  Real errpar(2)
  Integer Nu,kx,ix,nx, ky,iy,ny,ISTKGT, IUMB,iU,ndx,ndy
  External AF,BC,HANDLE


  Common / CSTAK / Ds(350000); Double Precision Ds
  Real Ws(1000)    # The PORT Library stack and its aliases.
  Real Rs(1000) ; Integer Is(1000) ; Complex Cs(500) ; Logical Ls(1000)
  Equivalence ( Ds(1),Cs(1),Ws(1),Rs(1),Is(1),Ls(1) )

  Call ISTKIN(350000,4)    # Initialize the PORT Library stack length.

  Call ENTER(1)

  Nu = 1

  Lx = 0; Rx = +1
  Ly = 0; Ry = +1

  kx = 4; ky = 4

  ndx = 3; ndy = 3

  tstart = 0; tstop = 1; dt = 1

  errpar(1) = 1e-2; errpar(2) = 1e-4

  ix = IUMB(Lx,Rx,ndx,kx,nx)    # Uniform grid.
  iy = IUMB(Ly,Ry,ndy,ky,ny)    # Uniform grid.

  iU = ISTKGT(Nu*(nx-kx)*(ny-ky),3)    # Space for the solution.
  Call SETR(Nu*(nx-kx)*(ny-ky),0e0,Ws(iU))

  Call TTGR (Ws(iU),Nu,kx,Ws(ix),nx, ky,Ws(iy),ny, tstart,tstop, dt,
           AF,BC,
           errpar,
           HANDLE)

  Call LEAVE

  Call WRAPUP

  Stop
```

```
End
```

The AF subroutine uses one Port subroutine

- SETERR is used to set an error state if $n_x\ n_y$ requires more space than reserved, in this case 100.

The body of the subroutine AF for (A4.7) is

```
Real xx(100),yy(100),D(600),x,y
Integer p,q,i
External LR,BT

If ( nx*ny > 100 ) { Call SETERR("AF - nx*ny .gt. 100",19,1,2) }

Call BTMAP(t,xi,yi,nx,ny, LR,BT, xx,yy,D)

Call TTGRU(nx,ny,D,Ux,Uy,Ut,Nu)    # Map into (x,y).

Do i = 1, Nu
  {
  Do q = 1, ny
    {
    Do p = 1, nx
      {
      x = xx(p+(q-1)*nx); y = yy(p+(q-1)*nx)

      a(p,q,i,1) = Ux(p,q,i) - .1*Uy(p,q,i)
      a(p,q,i,2) = Uy(p,q,i) + .1*Ux(p,q,i)
      AUx(p,q,i,i,1) = 1; AUy(p,q,i,i,2) = 1
      AUy(p,q,i,i,1) = -.1; AUx(p,q,i,i,2) = +.1

      F(p,q,1) = Ut(p,q,1) - x*y; FUt(p,q,1,1) = 1
      }
    }
  }

Call TTGRG(nx,ny,D,Nu, A,AU,AUx,AUy, F,FU,FUx,FUy)    # Map into (xi,eta).
```

where the subroutines LR and BT will be described shortly. The body of the subroutine BC for (A4.8) is

```
Real xx(100),yy(100),D(600),x,y
Integer i,j
External LR,BT

If ( nx*ny > 100 ) { Call SETERR("BC - nx*ny .gt. 100",19,1,2) }

Call BTMAP(t,xi,yi,nx,ny, LR,BT, xx,yy,D)

Call TTGRU(nx,ny,D,Ux,Uy,Ut,Nu)    # Map into (x,y).

Do j = 1, ny
  {
  Do i = 1, nx
    {
    x = xx(i+(j-1)*nx); y = yy(i+(j-1)*nx)

    If ( xi(i) == Lx | xi(i) == Rx )    # Left or right.
      {
      BU(i,j,1,1) = 1; B(i,j,1) = U(i,j,1)-t*x*y
      }
    Else If ( yi(j) == Ly )    # Bottom.
      {
      B(i,j,1) = (Ux(i,j,1)-t*y) - (Uy(i,j,1)-t*x)
      BUx(i,j,1,1) = 1; BUy(i,j,1,1) = -1    # Normal is (1,-1).
      }
    Else    # Top.
      {
      B(i,j,1) = -(Ux(i,j,1)-t*y) + (Uy(i,j,1)-t*x)
      BUx(i,j,1,1) = -1; BUy(i,j,1,1) = 1    # Normal is (-1,1).
      }
    }
  }

Call TTGRB(nx,ny,D, Nu, BUx,BUy,BUt)    # Map into (xi,eta).
```

The body of the subroutines `HANDLE` and `GERR` for computing and printing the error is same as before. The only change in the subroutine `EWE` for computing u in the previous example is the code for computing u.

```
Real t,xi(nx),yi(ny),U(nx,ny,Nu)
Integer nx,ny,Nu

Real xx(1000),yy(1000),D(6000),x,y
Integer p,i,j
External LR,BT

If ( ny > 1000 ) { Call SETERR("EWE - ny .gt. 1000",18,1,2) }

Do p = 1, Nu
  {
  Do i = 1, nx
    {
     Call BTMAP(t,xi(i),yi,1,ny, LR,BT, xx,yy,D)

    Do j = 1, ny
      {
      x = xx(j); y = yy(j)

      U(i,j,p) = t*x*y
      }
    }
  }
```

The subroutine LR is

```
  Subroutine LR(t,Lx,Rx,Lxt,Rxt)

# To get the L and R end-points of the mapping in x.

  Real t,Lx,Rx,Lxt,Rxt

  Lx  = 0; Rx  = 1
  Lxt = 0; Rxt = 0

  Return

  End
```

and the subroutine BT is

```
  Subroutine BT(t,x,f,g,fx,gx,ft,gt)

# To get the bottom and top of mapping in y.

  Real t,x,f,g,fx,gx,ft,gt

  f  = -1 + x; g  = x
  ft = 0;      gt = 0
  fx = 1;      gx = 1

  Return

  End
```

The output of this program is

```
error in U(.,  1.00e+00) =  8.94e-08
used     3388 /   700000 of the stack allowed.
```

The run-time for the above example was 28.1 seconds.

Example 5 - A Static Problem.

Consider the **pde**

$$a^{(1)} = \mathbf{u}_x$$
$$a^{(2)} = \mathbf{u}_y \tag{A4.9}$$
$$f = 0$$

on the domain $[0,1]\times[0,1]$, with **bc**s on the bottom

$$\mathbf{b} = u_y \tag{A4.10a}$$

and on the other sides

$$\mathbf{b} = \mathbf{u} - s(t,x,y) \tag{A4.10b}$$

where s is chosen so that the solution is $u = Real(z\ log(z))$.

The following program solves the **pde**-**bc** combination (A4.9)-(A4.10), using cubic B-splines over a mesh consisting of 3 non-uniformly spaced, distinct points on $(0,1)$, with the time-evolution carried out to 10^{-2} absolute accuracy. The non-uniform mesh used is $x_i \equiv (\frac{i-1}{n-1})^k$, for $i = 1, \cdots, n$. Since the solution has a $\log(z)$ singularity at $z = 0$, this grading of the mesh is sufficient to give $O(n^{-k})$ convergence where k is the order of the spline used. Without the grading, the convergence would only be $O(n^{-1})$ and it would require many more points to get comparable accuracy. The error at each time-step is printed out to confirm the accuracy of the computed solution. The main program is

```
# Main

# To solve Laplaces equation with Real ( z*log(z) ) as solution.

  Real tstart,tstop,dt,Lx,Rx,Ly,Ry
  Real errpar(2)
  Integer Nu,kx,ix,nx, ky,iy,ny,ISTKGT,iU,ndx,ndy,i
  External AF,BC,HANDLE


  Common / CSTAK / Ds(350000); Double Precision Ds
  Real Ws(1000)    # The PORT Library stack and its aliases.
  Real Rs(1000) ; Integer Is(1000) ; Complex Cs(500) ; Logical Ls(1000)
  Equivalence ( Ds(1),Cs(1),Ws(1),Rs(1),Is(1),Ls(1) )

  Call ISTKIN(350000,4)    # Initialize the PORT Library stack length.

  Call ENTER(1)

  Nu = 1

  Lx = 0; Rx = +1
  Ly = 0; Ry = +1

  kx = 4; ky = 4

  ndx = 2; ndy = 2

  tstart = 0; tstop = 1; dt = 1

  errpar(1) = 1e-2; errpar(2) = 1e-4

  nx = ndx+2*(kx-1); ix = ISTKGT(nx,3)    # Space for x mesh.
  Do i = 1, kx
    { Ws(ix+i-1) = 0; Ws(ix+nx-i) = Rx }     # 0 and Rx mult = kx.
  Do i = 1, ndx-1 { Ws(ix+kx-2+i) = Rx*((i-1)/(ndx-1e0))**kx }

  ny = ndy+2*(ky-1); iy = ISTKGT(ny,3)    # Space for y mesh.
  Do i = 1, ky
    { Ws(iy+i-1) = 0; Ws(iy+ny-i) = Ry }     # 0 and Ry mult = ky.
  Do i = 1, ndy-1 { Ws(iy+ky-2+i) = Ry*((i-1)/(ndy-1e0))**ky }

  iU = ISTKGT(Nu*(nx-kx)*(ny-ky),3)    # Space for the solution.
  Call SETR(Nu*(nx-kx)*(ny-ky),0e0,Ws(iU))

  Call TTGR (Ws(iU),Nu,kx,Ws(ix),nx, ky,Ws(iy),ny, tstart,tstop, dt,
             AF,BC,
             errpar,
             HANDLE)

  Call LEAVE

  Call WRAPUP
```

```
Stop

End
```

The body of the subroutine AF for (A4.9) is

```
Do i = 1, Nu
  {
  Do q = 1, ny
    {
    Do p = 1, nx
      {
      a(p,q,i,1) = Ux(p,q,i); a(p,q,i,2) = Uy(p,q,i)
      AUx(p,q,i,i,1) = 1; AUy(p,q,i,i,2) = 1
      }
    }
  }
```

The body of the subroutine BC for (A4.10) is

```
Do j = 1, ny
  {
  Do i = 1, nx
    {
    If ( y(j) == Ly )    # Neumann data on bottom.
      {
      b(i,j,1) = Uy(i,j,1)
      BUy(i,j,1,1) = 1
      }
    Else    # Dirichlet data.
      {
      r = Sqrt(x(i)**2+y(j)**2)
      If ( x(i) > 0 ) { theta = Atan(y(j)/x(i)) }
      Else            { theta = 2*Atan(1e0) }

      b(i,j,1) = U(i,j,1) - r*(Cos(theta)*Log(r) - theta*Sin(theta))
      BU(i,j,1,1) = 1
      }
    }
  }
```

The body of the subroutines HANDLE and GERR for computing and printing the error is same as before. The only change in the subroutine EWE for computing u in the previous example is the code for computing u.

```
Do p = 1, Nu
  {
  Do i = 1, nx
    {
    Do j = 1, ny
      {
      r = Sqrt(x(i)**2+y(j)**2)
      If ( x(i) > 0 ) { theta = Atan(y(j)/x(i)) }
      Else            { theta = 2*Atan(1e0) }

      If ( r > 0 ) { U(i,j,p) = r*(Cos(theta)*Log(r) - theta*Sin(theta)) }
      Else         { U(i,j,p) = 0 }
      }
    }
  }
```

The output of this program is

```
error in U(.,  1.00e+00) =  3.44e-02
used     2654 /   700000 of the stack allowed.
```

The run-time for the above example was 8.7 seconds.

Example 6 - Error Estimation.

We consider estimating the error in the solution to example 5 above, without using any information about the exact solution to do it.

The error estimation is done according to the scheme outlined in Example 6 of section 4. The solution is obtained as in example 5 above, then the solution is obtained on a finer grid (in this case, with twice the number of mesh points), and finally the solution is obtained on the crude mesh with the time evolution carried out to one-tenth the accuracy.

Since the problem is static, we expect that the error estimate in time should be very small and that for the spatial error will dominate.

The main program is

```
# Main

# To get error estimates for Laplaces equation with Real ( z*log(z) ) as solution.

  Real tstart,tstop,dt,Lx,Rx,Ly,Ry, EERR,errE,errR
  Real errpar(2)
  Integer Nu,kx,ix,nx, ky,iy,ny,ISTKGT,iU,ndx,ndy,i,I1MACH,
          ixr,iyr,nxr,nyr,iUr,iUe
  External AF,BC,HANDLE


  Common / CSTAK / Ds(350000); Double Precision Ds
  Real Ws(1000)    # The PORT Library stack and its aliases.
  Real Rs(1000) ; Integer Is(1000) ; Complex Cs(500) ; Logical Ls(1000)
  Equivalence ( Ds(1),Cs(1),Ws(1),Rs(1),Is(1),Ls(1) )

  Call ISTKIN(350000,4)    # Initialize the PORT Library stack length.

  Call ENTER(1)

  Nu = 1

  Lx = 0; Rx = +1
  Ly = 0; Ry = +1

  kx = 4; ky = 4

  ndx = 2; ndy = 2

  tstart = 0; tstop = 1; dt = 1

  errpar(1) = 1e-2; errpar(2) = 1e-4

  nx = ndx+2*(kx-1); ix = ISTKGT(nx,3)    # Space for x mesh.
  Do i = 1, kx
    { Ws(ix+i-1) = 0; Ws(ix+nx-i) = Rx }     # 0 and Rx mult = kx.
  Do i = 1, ndx-1 { Ws(ix+kx-2+i) = Rx*((i-1)/(ndx-1e0))**kx }

  ny = ndy+2*(ky-1); iy = ISTKGT(ny,3)    # Space for y mesh.
  Do i = 1, ky
    { Ws(iy+i-1) = 0; Ws(iy+ny-i) = Ry }     # 0 and Ry mult = ky.
  Do i = 1, ndy-1 { Ws(iy+ky-2+i) = Ry*((i-1)/(ndy-1e0))**ky }

  iU = ISTKGT(Nu*(nx-kx)*(ny-ky),3)    # Space for the solution.
  Call SETR(Nu*(nx-kx)*(ny-ky),0e0,Ws(iU))

  iwunit = I1MACH(2)
  Write(iwunit,9000)
 9000 Format(" Solving on crude mesh.")

  Call TTGR (Ws(iU),Nu,kx,Ws(ix),nx, ky,Ws(iy),ny, tstart,tstop, dt,
             AF,BC,
             errpar,
             HANDLE)
```

```
 dt = 1

 ndx = 2*ndx-1; ndy = 2*ndy-1    # Refine mesh.

 nxr = ndx+2*(kx-1); ixr = ISTKGT(nxr,3)    # Space for x mesh.
 Do i = 1, kx
   { Ws(ixr+i-1) = 0; Ws(ixr+nxr-i) = Rx }     # 0 and Rx mult = kx.
 Do i = 1, ndx-1 { Ws(ixr+kx-2+i) = Rx*((i-1)/(ndx-1e0))**kx }

 nyr = ndy+2*(ky-1); iyr = ISTKGT(nyr,3)    # Space for y mesh.
 Do i = 1, ky
   { Ws(iyr+i-1) = 0; Ws(iyr+nyr-i) = Ry }     # 0 and Ry mult = ky.
 Do i = 1, ndy-1 { Ws(iyr+ky-2+i) = Ry*((i-1)/(ndy-1e0))**ky }

 iUr = ISTKGT(Nu*(nxr-kx)*(nyr-ky),3)    # Space for the solution.
 Call SETR(Nu*(nxr-kx)*(nyr-ky),0e0,Ws(iUr))

 iwunit = I1MACH(2)
 Write(iwunit,9001)
9001 Format(" Solving on Refined mesh.")

 Call TTGR (Ws(iUr),Nu,kx,Ws(ixr),nxr, ky,Ws(iyr),nyr, tstart,tstop, dt,
            AF,BC,
            errpar,
            HANDLE)

 dt = 1; errpar(1) = errpar(1)/10; errpar(2) = errpar(2)/10

 iUe = ISTKGT(Nu*(nx-kx)*(ny-ky),3)    # Space for the solution.
 Call SETR(Nu*(nx-kx)*(ny-ky),0e0,Ws(iUe))

 iwunit = I1MACH(2)
 Write(iwunit,9002)
9002 Format(" Solving with errpar/10.")

 Call TTGR (Ws(iUe),Nu,kx,Ws(ix),nx, ky,Ws(iy),ny, tstart,tstop, dt,
            AF,BC,
            errpar,
            HANDLE)

 errR = EERR(kx,ix,nx, ky,iy,ny, Ws(iU),Nu, ixr,nxr,iyr,nyr,Ws(iUr), tstop)

 errE = 0
 Do i = 1, Nu*(nx-kx)*(ny-ky)
   {
   errE = Max( errE, Abs( Ws(iU+i-1)-Ws(iUe+i-1) ) )
   }

 iwunit = I1MACH(2)
 Write(iwunit,9003) errE
9003 Format(" U error from U and Ue =",1p4e10.2)

 iwunit = I1MACH(2)
 Write(iwunit,9004) errR
```

```
9004 Format(" U error from U and Ur =",1p4e10.2)
 Call LEAVE
 Call WRAPUP
 Stop
 End
```

and the subroutine for obtaining the error estimate is

```
  Real Function EERR(kx,ix,nx, ky,iy,ny, U,Nu, ixr,nxr, iyr,nyr,Ur, t)

# To get and print the error estimate at each time-step.

  Real U(1),Ur(1),t    # U(nx-kx,ny-ky,Nu), Ur(nxr-kx,nyr-ky,Nu).
  Integer kx,ix,nx, ky,iy,ny,Nu, ixr,nxr,iyr,nyr

  Common / CSTAK / Ds(500); Double Precision Ds
  Real Ws(1000)    # The PORT Library stack and its aliases.
  Real Rs(1000) ; Integer Is(1000) ; Complex Cs(500) ; Logical Ls(1000)
  Equivalence ( Ds(1),Cs(1),Ws(1),Rs(1),Is(1),Ls(1) )

  Real errU
  Integer i,I1MACH,ISTKGT, iFA,iFAr,
          KA(2),ITA(2),NTA(2),IXA(2),NXA(2),MA(2), ILUMD,ixs,iys,nxs,nys

  Call ENTER(1)

# Find the error in the solution at 2*kx * 2*ky points / fine mesh rectangle.

  ixs = ILUMD(Ws(ixr),nxr,2*kx,nxs)    # x search grid.
  iys = ILUMD(Ws(iyr),nyr,2*ky,nys)    # y search grid.

  KA(1) = kx; KA(2) = ky; ITA(1) = ix; ITA(2) = iy; NTA(1) = nx; NTA(2) = ny
  IXA(1) = ixs; IXA(2) = iys; NXA(1) = nxs; NXA(2) = nys
  MA(1) = 0; MA(2) = 0    # Get solution.

  iFA = ISTKGT(nxs*nys,3)    # Approximate solution values.

  Call TSD1(2,KA,Ws,ITA,NTA,U, Ws,IXA,NXA,MA, Ws(iFA))    # Evaluate them.

  KA(1) = kx; KA(2) = ky; ITA(1) = ixr; ITA(2) = iyr; NTA(1) = nxr; NTA(2) = nyr
  IXA(1) = ixs; IXA(2) = iys; NXA(1) = nxs; NXA(2) = nys
  MA(1) = 0; MA(2) = 0    # Get solution.

  iFAr = ISTKGT(nxs*nys,3)    # Approximate solution values.

  Call TSD1(2,KA,Ws,ITA,NTA,Ur, Ws,IXA,NXA,MA, Ws(iFAr))    # Evaluate them.

  errU = 0    # Error in solution values.
  Do i = 1, nxs*nys
    {
    errU = Max(errU,Abs(Ws(iFAr+i-1)-Ws(iFA+i-1)))
    }

  Call LEAVE

  EERR = errU; Return

  End
```

The rest of the subroutines are the same as in example 5.

The output of this program is

```
Solving on crude mesh.
error in U(.,  1.00e+00) =  3.44e-02
Solving on Refined mesh.
error in U(.,  1.00e+00) =  1.50e-02
Solving with errpar/10.
error in U(.,  1.00e+00) =  3.44e-02
U error from U and Ue =  0.00e+00
U error from U and Ur =  4.27e-02
used     3426 /   700000 of the stack allowed.
```

The run-time for the above example was 37.7 seconds.

The output shows that, just as we expected, the time error estimate is small, and the spatial error dominates. Also, the error estimates are quite good, and over-estimates.

If a uniform grid had been used in the above run, the errors would have been `3.44e-02` and `2.68e-02`, or substantially worse than achieved above. A convergence rate of $O(n^{-k})$ is a *lot* better than $O(n^{-1})$, even when n is only 2 or 3.

Notice that the run-time for example 5 is 4 or 5 times less than example 6. This is typical and indicates that obtaining error estimates is expensive. It is a *really* good idea to apply the above scheme whenever working on a new problem and concern about the accuracy of the computed solution is reasonable. Once the mesh and `errpar` have been chosen appropriately and confirmed, production runs can dispense with the error estimates and their expense. But some error estimation is a good idea. The alternative is a pile of cheap numbers that may be garbage. Good piles of number cost more to obtain.

We have from the above estimates that

$$\| \mathbf{u}_{true} - \mathbf{u} \|_\infty \le 4.27\times10^{-2} \tag{A4.11}$$

and from that, and the assumption that the spatial error is $O(n^{-k})$, it's tempting to say that

$$\| \mathbf{u}_{true} - \mathbf{u}_r \|_\infty \le \| \mathbf{u}_{true} - \mathbf{u} \|_\infty \times 2^{-k} \approx 1.6\times10^{-3} \tag{A4.12}$$

which underestimates the actual error by an order of magnitude! This is typical of the estimates (A4.11) and (A4.12). (A4.11) estimates the error in the crude mesh solution and is quite reliable because it *only* assumes that $\mathbf{u}_r$ is more accurate than $\mathbf{u}$. (A4.12) estimates the error in the refined solution and is not so reliable, because it also assumes that the spatial error is $O(n^{-k})$. That assumption is valid for large n, but not necessarily for small n. So we can reliably estimate the error in the crude solution, but not the refined one.

The above error estimation scheme can be applied to the solution at any point in time, not just `tstop` as was done above.

Example Summary

We summarize the memory and time usage of each of the examples on two widely disparate machines, the Cray-1 in single precision under COS and a Vax 11/750 in double precision, under Unix and using a floating-point accelerator. These two tables provide a rough guide to the resource utilization expected when running `TTGR` on various machines.

Single Precision on a Cray-1		
Example	Memory (words)	Time (secs)
1	825	0.1025
2	9398	4.1503
3	1260	0.2031
4	3388	0.3645

5	2654	0.1207
6	3426	0.5042

Double Precision on a Vax 11/750		
Example	Memory (words)	Time (secs)
1	1356	5.5
2	18324	465.8
3	2139	11.4
4	6460	30.4
5	5020	9.5
6	6530	41.7

Appendix 5

Routine, Common and Error State Summary

This appendix summarizes the calling sequences for the routines of TTGR, the contents of the relatively public Common regions and the error states. It is terse and meant to serve as a reference guide, not as a tutorial. The top level of TTGR is

```
Call TTGR(U,Nu,kx,x,nx, ky,y,ny,
          tstart,tstop,dt,
          AF,BC,
          errpar,
          HANDLE)
```

The AF procedure is of the form

```
Subroutine AF(t,x,nx,y,ny,U,Ux,Uy,Ut,Utx,Uty,Nu,
              A,AU,AUx,AUy,AUt,AUtx,AUty,
              f,fU,fUx,fUy,fUt,fUtx,fUty)
```

the BC procedure is of the form

```
Subroutine BC(t,Lx,Rx,Ly,Ry,U,Ux,Uy,Ut,Utx,Uty,Nu,
              B,BU,BUx,BUy,BUt,BUtx,BUty)
```

and the HANDLE procedure has the form

```
Subroutine HANDLE(t0,U0,V0,t,U,V,Nu,nxmk,nymk,kx,x,nx,ky,y,ny,dt,tstop)
```

The "Return-End" HANDLE procedure is TTGRH.

The "Return-End" BC procedure is TTGRP, for when $n_u = 0$.

The statistics printing procedure is TTGRX.

The basic knob twiddling routine for TTGR is

```
Call TTGRV(j,f,r,i,l)
```

The following table summarizes the values that can be set by TTGRV

Name	j	Default	Set to
theta	1	1	f
beta	2	1	f
gamma	3	1	f
delta	4	0	f
hfract	1001	1	r
egive	1002	100	r
kj	2001	0	i
minit	2002	10	i
maxit	2003	50	i
kmax	2004	10	i
kinit	2005	2	i
mmax	2006	15	i
mxq	2008	0	i

myq	2009	0	i
la	2010	1	i
pieces	2011	+2	i
pc	2012	0	i
accel	2013	0	i
xpoly	3001	False	l
erputs	3002	False	l
N(i)	4000+i	–	i

where `mxq` = 0 means that `kx` quadrature points will be used in *x*, similarly for `myq`.

The procedural knob twiddler is

```
Call TTGRR(U,Nu,kx,x,nx, ky,y,ny,
           tstart,tstop,dt,
           AF,BC,
           ERROR,errpar,
           HANDLE)
```

where the `ERROR` procedure has the form

```
Logical Function ERROR(U,Nu,nxmk,kx,x,nx, ky,y,ny,t,dt,
                       errpar,
                       erputs,
                       eU,eV)
```

The `TTGR` coordinate mapping routines for `AF` are

```
Call TTGRU(nx,ny,D, Ux,Uy,Ut, Nu)
```

to map from internal to user coordinates, and

```
Call TTGRG( nx,ny,D, Nu,
            A,AU,AUx,AUy,
            F,FU,FUx,FUy)
```

to map from user to internal coordinates.

For mapping `BC` from internal to user coordinates there is `TTGRU` as used above in `AF`, and to map `BC` from user back into internal coordinates

```
Call TTGRB( nx,ny,D, Nu, BUx,BUy,BUt )
```

Common Regions

When the user cannot evaluate any of `AF` or `BC` that fact can be signaled to `TTGR` via

```
Common / TTGRF / Failed; Logical Failed
```

Naming Conventions

The naming convention for `TTGR` is the same as that for the Port Library: all hidden (not user callable) subroutines have names beginning with a letter followed by a digit. If users avoid such names, there will be no name conflicts.

Error States.

This section provides a list of the error states [14] that may be encountered when using TTGR. Some interpretation of these error messages is made to aid the user in finding bugs (if they exist) in the user-supplied code AF, BC or HANDLE. For each level of (entry to) TTGR, the error message and number for a given error state is the same, always reflecting the error from the bottom layer of TTGR. The list of error states below, along with interpretation, is the complete set of error states for the TTGR package as obtained from the lowest level of TTGR.

There are many internal variables of TTGR that may be controlled by the subroutine TTGRV. Thus, there are many more ways to call TTGR with bad data than just the obvious ones involving data in the calling sequence for TTGR. The list of error states given below is complete for TTGR/TTGRV, and therefore involves variables not mentioned in the calling sequence for TTGR. If you are only using TTGR, and not TTGRV as well, then the error states mentioning variables not in the TTGR call can not occur. So if you bump into an error state below that mentions an undescribed or unknown variable, simply ignore it, it cannot happen to you.

1 - Nu .lt. 1.

2 - kx .lt. 2.

3 - nx .lt. 2*kx.

4 - ky .lt. 2.

5 - ny .lt. 2*ky.

6 - dt=0 on input. The user-chosen value for the time-step dt is too small, that is, tstart+dt ≡ tstart.

7 - dt has wrong sign on input. dt and tstop-tstart must have the same sign.

8 - mxq .lt. kx-1.

9 - myq .lt. ky-1.

10 - Abs(LA) must be 1 or 2.

11 - Pieces must be 0 or 2.

12 - PC must be 0, 1 or 2.

13 - Accel must be 0, 1 or 2.

14 - For PC .gt. 0 need Abs(LA) = 2.

15 - For PC .gt. 0 need Accel = 1 or 2.

1000 - dt from HANDLE has wrong sign. Recoverable.

1001 - Cannot raise dt in HANDLE when Failure is set. Recoverable.

1002 - E(i) .le. 0 returned by ERROR. Recoverable. The error request is too small. Having a relative error request on a variable going to 0 can cause this.

1003 - mxq=kx-1 and Order=0. Recoverable. Must have mgq=k when one of the pdes is of zero order.

1004 - pde(i) is vacuous. Recoverable. There is no i^{th} pde.

1005 - Improper BCs. Recoverable. The **bc**s and pdes do not match properly.

1006 - pde system not in minimal order form. Recoverable. The pde can have derivatives removed from it.

1007 - Too few boundary conditions. Recoverable.

1008 - Too many boundary conditions. Recoverable.

1009 - Mixed boundary conditions are overdetermined. Recoverable. There are too many mixed **bc**s.

1010 - Singular Mixed BCs. Recoverable. The mixed **bc**s were singular so frequently that dt went to 0.

1011 - Dirichlet boundary conditions are overdetermined. Recoverable. There are too many Dirichlet **bc**s.

1012 - Singular Dirichlet BCs. Recoverable. The Dirichlet **bc**s were singular so frequently that dt went to 0.

1013 - Singular Jacobian. Recoverable. The Jacobian for the `pde` was singular so frequently that dt went to 0.

1014 - Out of stack space for LU decomposition. Recoverable.

1015 - Too many iterations needed in splitting. Recoverable.

1016 - dt=0. Recoverable. The time-step has become too small. The problem may be very badly scaled, that is units like light-years and micro-grams are being used simultaneously. Another cause is too small an accuracy requirement, like errpar(2)=0 when the solution is exceedingly small.

1017 - dt=0 returned from `HANDLE`. Recoverable. Handle lowered dt and it became too small.

1018 - `AF`, `BC` failure. Recoverable. Failed = True occurred in `AF` or `BC` so often that dt went to 0.

1019 - Too many Newton iterations predicted. Recoverable. The number of Newton iterations was predicted to be too large so often that dt went to 0. Probable cause is a bad Jacobian, see next error state.

1020 - Too many Newton iterations needed. Recoverable. Too many Newton iterations were needed so often that dt went to 0. Probable cause is an incorrectly computed Jacobian. Another possible cause is that Minit and/or Maxit are too small. Yet another possible cause is a very badly conditioned Jacobian. Further possible causes: too stringent an error request or a mesh that is too non-uniform.