TTGU - A Package for Solving Time Varying Partial Differential Equations on a Union of Rectangles

L. Kaufman

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

A formulation is presented for partial differential equations on a union of rectangles 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 TTGU (Transient Tensor Galerkin for partial differential equations on a Union of rectangles), is described. It solves the same general type of partial differential equation as TTGR[15], but TTGR restricts the domain to a rectangle or domains can be easily mapped into rectangles. 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 TTGU is given along with many examples.

The examples are available through electronic mail.

There are 5 examples and the **fortran** for each is available in single or double precision. For example, the command

mail research!netlib
send only ttgux1 from port
send only dttgux5 from port
.

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

October 29, 1990

TTGU - A Package for Solving Time Varying Partial Differential Equations on a Union of Rectangles

L. Kaufman

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 union of rectangles, 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. Currently, the package demands that the interfaces between rectangles be either a point of a whole side of the rectangle. The mathematical formulation is given in section 2.

This package extends the functionality of TTGR[15] and users, who have a problem on one rectangle or have a problem that can be mapped onto one rectangle, should be using TTGR and reading [15].

Getting Started if you have not read TTGR[15]:

If you have not read this document before and are not familiar with TTGR[15], 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 7.
- Skim section 5, Software, pp 11-19.
- Read Appendix 1, 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 1 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.

Getting started if you are familiar with TTGR[15]:

If you are familiar with TTGR and want to solve a **pde**, you should do or know the following:

- Read pp. 10-11 indicating the calling sequence of TTGU. There are three extra parameters, not needed in TTGR, which permit the user to specify the mesh.
- The subroutines AF and BC used for specifying the partial differential equations and the boundary conditions have not been changed from TTGR.
- If you have been using TSL2W to specify nonconstant boundary conditions or now have non constant boundary conditions, you will want to read about the subroutine ICON on p. 18.
- The default options are the same, but in the current implementation only banded direct solvers are

available to solve the underlying matrix problem and the whole Jacobian must be computed. One cannot ask that only certain pieces of the Jacobian be computed as one could in TTGR.

- Read Appendix 1, Programs, Example 1, pp.1-9.

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-7.
- A coupled system of **pde**s, see Example 2, p 7.
- A material interface, see Example 3, p 8
- A nonconstant initial conditions problem, see Example 4, p 9
- A static problem, see Example 5, pp 9-10

A Principle.

The guiding principle used during the design of `TTGU` 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 `TTGU` very useful. However, people wanting to solve the same problem many times in a production environment may find `TTGU`, with the default settings, slow for their needs; see section 6 for ways to speed `TTGU` 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 `TTGU` for Transient Tensor product Galerkin for partial differential equations on a Union of rectangles. Section 6 describes ways of making `TTGU` run faster than the default settings allow and also describes alternative ways of entering and using the package.

Appendix 1 presents the programs used to solve the examples discussed in section 4.

Appendix 2 summarizes the basic procedures available in `TTGU`, along with their arguments, and gives a list of error states and problems that may arise when using `TTGU`, along with the common causes of such difficulties.

Appendix 1 and 2 of [15] give brief tutorials on B-splines and extrapolation respectively, and the interested reader should peruse that document for background information.

A Warning.

This software is in an infant state. If the user finds bugs or unexpected behavior, please contact the author. 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 the author, and let the author know generally how things go with it. My electronic mail address is

research!lck

The examples are available through electronic mail.

There are 5 examples and the **fortran** for each is available in single or double precision. For example, the command

```
mail research!netlib
send only ttgux1 from port
send only dttgux5 from port
.
```

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

2. Statement of the Problem.

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

The pde-bc Formulation.

On each rectangle, the general **pde**-**bc** form that can be solved with the approach used in `TTGU` 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

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

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).

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.

The number of **pde** variables, n_u, is assumed to be the same for all rectangles.

`TTGU` demands that the interface between rectangles be either a point or a whole side of the rectangle. Thus the following situation

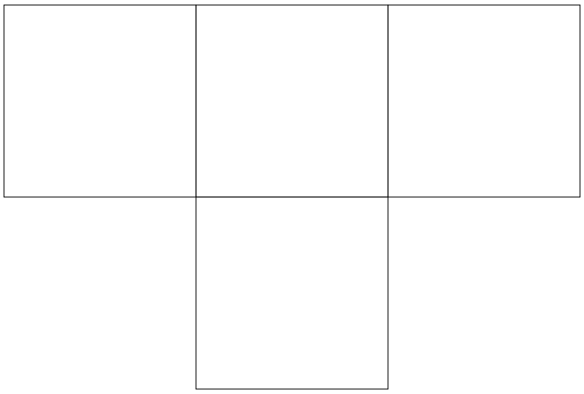


Fig 2.1

is valid, but the following configuration

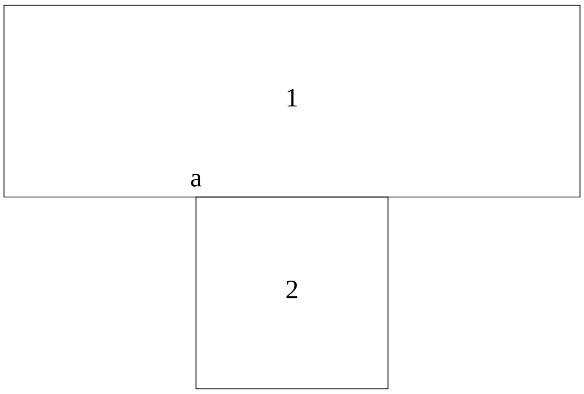


Fig 2.2

is not valid. The main reason for this restriction is to insure that when the user specifies a knot sequence (see the next section), an error in the multiplicity of the knots is not made which would preclude high order convergence. This restriction might be eliminated in future versions of this code if requested by users. In rectangle 1 if one is using a cubic B-spline the knot at "a" in Fig. 2.2 should have multiplicity 3 to get fourth order convergence.

3. General Method of Solution.

On rectangle j let the solution $\mathbf{u}(t,x,y)$, for a given instant in time, be approximated by a tensor-product of B-splines [21,1,2,3] of **order** $k_{x,j}$ and $k_{y,j}$ on **meshes** $X(1) \le \cdots \le X(NX_j)$, and $Y(1) \le \cdots \le Y(NY_j)$. 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 $k_{x,j}$, with $k_{x,j}-2$ continuous derivatives, where $k_{x,j} \ge 2$ 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 for each rectangle

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

If we set $h_x = \max_i | X(i+1) - X(i) |$ and h_y similarly, then the error, $\| u_i - \hat{u}_i \|_\infty \equiv \max_{x,y} | u_i(t,x,y) - \hat{u}_i(t,x,y) |$ 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 [25,19] 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 [10,25] 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" [7,8], 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 [16], or an exponentially-fitted technique [13], or even an explicit method such as Gragg's modified mid-point rule [12,13], could be used. `TTGU` uses backwards-Euler as the default time discretization scheme. See [20] 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 δ/2, the results of these two computations can be combined using extrapolation [5,12] 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 $O(\delta^{\gamma})$, $O(\delta^{2\gamma}), \cdots,$ $O(\delta^{P\gamma}), \cdots$.

A step-size and order monitor is available [17,18] 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 `TTGU` 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.

At the bottom level one must solve a linear system. For `TTGU` the user may specify that for the interior of each rectangle a banded solver, with or without pivoting, be used. At this point iterative methods and sparse direct methods are not an option. Section 6 gives a somewhat more detailed outline of the spatial discretization process and the various parameters that describe the solution procedure.

Mesh restrictions

When two rectangles share a common side the order of the meshes and the mesh themselves must be the same. For example, if two rectangles meet at $x = 0$, then the y meshes for these two rectangles must agree as well as k_y. Figure 3.1 is **not permitted**, while Figure 3.2 is permitted. Even if $k_{x,1} = 2$ and $k_{x,2} = 4$, Figure 3.2 would be fine.

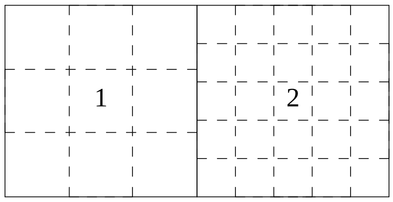


Figure 3.1: This mesh is not permitted

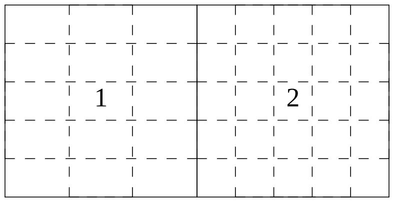


Figure 3.2: Permitted mesh with $k_{y,1} = k_{y,2}$

Currently the solution on interfaces are the same independent of the rectangle used to compute the solution from the B spline coefficients. With Figure 3.1 the value of the solution on the interfaces might depend on which B-spline coefficients were used, those from rectangle 1 or those from rectangle 2. Figure 3.1 entails a level of approximation that the author did not wish to consider with the first version of the code. Increased functionality was the primary objective and efficiency for certain special problems was considered secondary. Perhaps in the next version of the program Figure 3.1 will be permitted.

4. Examples - Formulation.

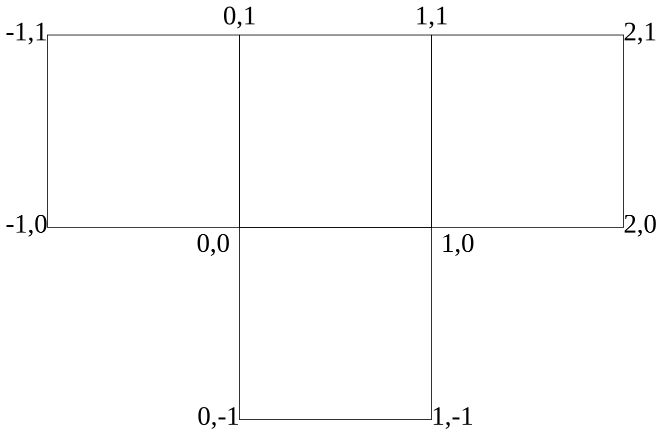
This section gives the formulation of many examples in terms of (2.1)-(2.2). See Appendix 1 for programs that solve these problems using `TTGU`.

Example 1 - A Simple Heat Equation.

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

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

on the T-shaped region



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

$$u(t,x,y) = t \cdot x \cdot y \qquad (4.2)$$

with initial conditions

$$u(0,x,y) = 0\,. \qquad (4.3)$$

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

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

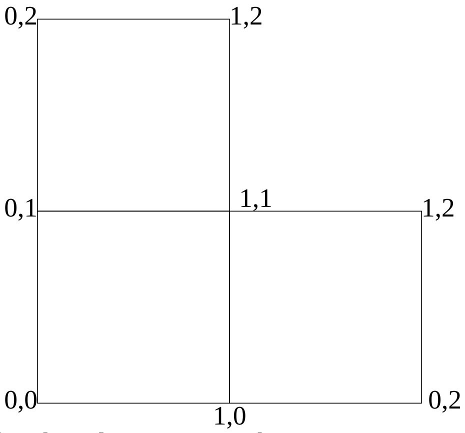
See example 1 in Appendix 1 for code solving this problem. The only difference between this problem and example 1 in the documentation of TTGR[15] is the domain. Both examples use the same pde, but for the first example in TTGR[15] the domain is a rectangle.

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 L shaped region



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

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

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 1 for code solving this problem. The only difference between this example and example 2 in the documentation’of TTGR[15] is the domain. In [15] the domain is a simple rectangle.

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 [19,25]. That method has the property that the term $\mathbf{a}(\cdot)$ in (2.1) is forced to be continuous everywhere, see [19]. 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 `TTGU` is important in their formulation.

Example 3 - Interfaces

This is example 3 in the documentation of TTGR[15]. It 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 1 for code solving this problem. In the documentation of TTGR, this example was treated as one rectangle and to take into consideration the material interfaces, multiple knots were inserted at the interfaces. For TTGU the problem is treated as being defined on 3 unit squares stacked on top of each other.

Example 4 - Non constant boundary conditions

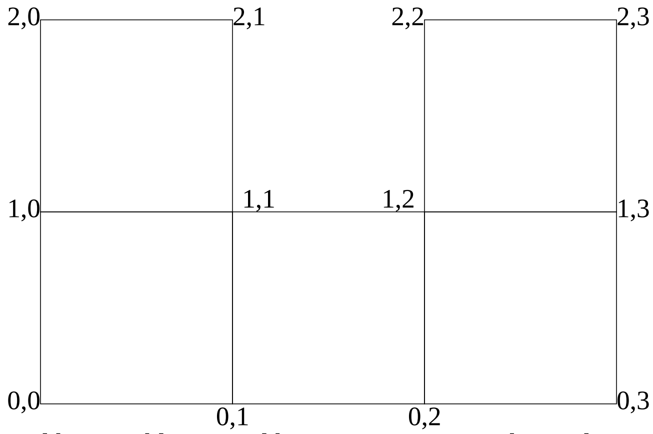
This example has the same spatial domain and the same pde as in Example 2. However in Example 2 the initial time is 0, so that the solution can be represented initally by the constant 1, and in this example the intial time is set to 1, so that initally *u* is not a constant and the B-spline coefficients have to be determined before the PDE is solved.

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



This domain was suggested by a problem posed by G.C.Scott. Since the Real part of any analytic function solves Laplace's equation, a good choice for u is the Real part of z log (z), where $z = x + i\,y$. Dirichlet **bc**s on the left, right and top of the domain, consistent with that choice for u can be stipulated. For the bottom Neumann data $u_y = 0$ can be chosen. This gives the boundary conditions

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

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 `TTGU` does not require it.

The logarithmic singularity will result in slow convergence unless a non-uniform grid is used. See example 5 in Appendix 1 for code solving this problem. Again the only difference between this problem and example 5 in the documentation of TTGR[15] is the domain. In [15] the domain is a simple rectangle.

5. Software for the pde-bc Problem.

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

Before invoking `TTGU` the user must

- Make B-spline meshes for x and y on each rectangle.
- 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).
  - `HANDLU` - to output (print) the solution results.

Each of these preparatory steps are illustrated in Appendix 1 and will not be described here.

The B-spline coefficients are stored in such a way that a user with one rectangle can use either TTGR[15] or TTGU without changing data structures. Moreover, with the structure defined below it is easy to use other available software, e.g. TSD1 which evaluates the solution from one rectangle. Because each rectangle is stored using a different mesh, one cannot simply add an index stipulating the rectangle. The following storage map for the B-spline coefficients may look cumbersome, but it works.

Assume that there are n_r rectangles and let

$$s_j = 1 + \sum_{m}^{j-1} (NX_i - k_{x,m}) \times (NY_i - k_{y,m}), \quad \text{for } 0 \le j \le n_r .$$

Then the B-spline coefficients $\mathbf{b}_{p,q,i}$ for rectangle j in (3.1) is stored in a matrix U dimensioned $(s_{r+1} - 1, n_u)$ at position $\mathrm{U}(s_j - 1 + p + (q-1) \times (NX_j - k_{x,j}), i)$ The subroutines AF and BC will only be given the U coefficients for one rectangle at a time so the above notation will be irrelevant to them. The use of the above data structure is shown in HANDLU and with a B-spline evaluator in Appendix 1. It is not as strange as it looks on the surface.

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

```
Call TTGU (U,nu,nr,kx,X,nx,ixb,ky,Y,ny,iyb,
           tstart,tstop,dt,
           AF,BC,
           errpar,
           HANDLU)
```

The input to TTGU is

`U`	-	The B-spline coefficients (3.1) for the initial values of the **pde** variables **u** on all rectangles. See above description for the storage of U. See below for ways to get the initial conditions, `ICON`.
`nu`	-	The number n_u of **pde** variables **u**.
`nr`	-	The number n_r of rectangles.
`kx`	-	An array of length n_r where `kx(j)` gives the B-spline order to be used in x for rectangle j. `kx` ≥ 2 is necessary.
`X`	-	The B-spline mesh to be used in x. Elements `X(IXB(J))` through `X(IXB(J)+NX(J)-1)` correspond to the mesh in x for rectangle j. The multiplicity of `X(IXB(J))` and `X(IXB(J)+NX(J)-1)` must be at least `kx(j)` for $J = 1, \ldots$ `nr`. The Port Library routines for making uniform meshes, `UMB` and `LUMB`, guarantee the first and last mesh points have multiplicity `kx`.

`nx` - An array of length n_r where `nx(j)` gives the length of the mesh array `X` for rectangle *j* beginning at `ixb(j)`.

`ixb` - An array of length n_r where `ixb(j)` points to the beginning of the *x* mesh for rectangle *j* in the array `X`.

`ky` - An array of length n_r where `ky(j)` gives the B-spline order to be used in *y* for rectangle *j*. `ky` ≥ 2 is necessary.

`Y` - The B-spline mesh to be used in *y*. Elements `Y(IYB(J))` through `Y(IYB(J)+NY(J)-1)` correspond to the mesh in *y* for rectangle *j*. The multiplicity of `Y(IYB(J))` and `Y(IYB(J)+NY(J)-1)` must be at least `ky(j)` for $J = 1, \cdots$ `nr`. The Port Library routines for making uniform meshes, `UMB` and `LUMB`, guarantee the first and last mesh points have multiplicity `ky`.

`ny` - An array of length n_r where `ny(j)` gives the length of the mesh array `Y` for rectangle *j* beginning at `iyb(j)`.

`iyb` - An array of length n_r where `iyb(j)` points to the beginning of the *y* mesh for rectangle *j* in the 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 `TTGU`; see output description below.

`dt` - The initial choice for the time-step. The performance of `TTGU` 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 `TTGU` 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 `TTGUP` 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

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

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

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

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.

`HANDLU` - A user-supplied subroutine that will be called by `TTGU` at the end of each time-step. `HANDLU` 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 `TTGUH` may be used in place of `HANDLU`.

The output from `TTGU` 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 `HANDLU`. If an error state exists on return (see Appendix 2), `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 `TTGU`.

`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. `TTGU` raises the time-step rapidly when solving a static problem, but it is wasteful to solve a problem more than once.

A "restart" in `HANDLU`, see below, for a static problem is a disaster − the user should **STOP** right there. `TTGU` 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 [9] is, neglecting lower order terms, when the default setting are used,

$$n_u(\max_i(n_{x,i}n_{y,i}(3H_i-1)))+2n_u{}^2(\sum_{i=1}^{n_r}n_{x,i}n_{y,i}(n_{x,i}-k_{x,i}+n_{y,i}-k_{y,i})+(\sum_{i=1}^{n_r}(n_{x,i}-k_{x,i}+n_{y,i}-k_{y,i}))^2)$$

Real words (storage units), where $H_i \equiv n_u\ (\ k_{x,i}\ +\ (\ n_{x,i}\ -\ k_{x,i}\ )\ (k_{y,i}\ -\ 1\ )\ )$ is the half-band-width of the Jacobian, on the i^{th} rectangle, n_u is the number of **pde**s, $n_{x,i}$ is the number of points in the spatial mesh for x, $k_{x,i}$ is the B-spline order for the mesh x, $n_{y,i}$ is the number of points in the spatial mesh for y and $k_{y,i}$ is the B-spline order for the mesh in y for the i^{th} rectangle.

All scratch space for `TTGU` is taken from the stack. Solving **pde**s is a non-trivial process, requiring substantial work space. For virtually all problems solved by `TTGU` 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 1.

Run-time.

The run-time of TTGU is proportional to $n_u \sum_{i=1}^{n_r} n_{x,i}\, n_{y,i}\, (H_i - 1)^2$ for the default settings. See section 6 for ways to make TTGU run much faster.

The storage and run-time of TTGU 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 TTGU is called DTTGU. The calling sequence for DTTGU is precisely the same as that for TTGU, with *all* floating-point arguments Double Precision, *except* errpar, which remains Real.

AF **and** BC **Descriptions.**

The user-supplied subroutines AF and BC, which define the **pde** -**bc** problem to be solved, are now described. There is a distinct invocation for each rectangle .i.e., one is asked for information for only one rectangle at a time. The subroutines AF and BC have the same calling sequence as they do for TTGR[15]. When TTGU 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 TTGU 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 TTGU 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 TTGU 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`, $q=1,\ldots,$ `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`, $q=1,\ldots,$ `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`, $q=1,\ldots,$ `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`, $q=1,\ldots,$ `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.

Although the calling sequence of `AF` has not been changed for historical reasons the user may wish to know that actually `nye` is currently always set to 1 but that `nxe` gives all the quadrature points in the x direction for any given rectangle. Thus `nxe` is always larger the `nye`. Also in the current version the user may determine the current rectangle by inserting the common statement

```
common /ttguc/ ir
```

The variable `ir` indicates the rectangle under current investigation and should not be changed by the user.

When `TTGU` 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 `TTGU` 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 `TTGU` 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 `TTGU` 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.

`HANDLU` Description.

The user-supplied output and control subroutine `HANDLU` 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. When `TTGU` 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, `TTGU` will

```
Call HANDLU(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" `HANDLU` subroutine `TTGUH` may be used.

`TTGU` also invokes `HANDLU` 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 `TTGU` failed at time `t`. Such things are called "restarts" and are typically expensive and worth knowing about.

The input provided by `TTGU` to `HANDLU` 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 1, the needed values of `nx`, `ny` and `Nu` are obtained, magically, from `Common` regions internal to `TTGU`. This botch will probably be fixed in the next edition of `TTGU`.

The output from `HANDLU` 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.

For each rectangle to evaluate the solution created by `TTGU`, 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 `TTGU` to make evaluation easier for both users and the authors. Example 1 in Appendix 1 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 ICON(u, nu, nr, kxr, x, nxr, ixb, kyr, y, nyr, iyb, ic)
```

The input parameters nu, nr, kxr, x, nxr, ixb, kyr, y, nyr, and iyb have the same meaning as they do on the invocation of `TTGU`. The output parameter u may be passed directly into `TTGU` without any modification. The input parameter `ic` is the name of a user written subroutine which must declared external in the program calling `ICON`. The user written subroutine `ic` is called once per rectangle and supplies the initial values of the solution at specified points in the rectangle. When `ICON` needs to compute the initial conditions it will

```
CALL IC(nu, ir, xq, nxq, yq, nyq, ui)
```

and provides as input values

`nu` - The number n_u of **pde** variables **u**.

`ir` - The rectangle on which data is to be supplied

`xq` - A list of points x where u is to be evaluated. This `xq` is *not* the B-spline mesh X. The points `xq` at which u is desired are determined by the quadrature rule used by `ICON` to implement Galerkin's method.

`nxq` - The length of xq.

`yq` - A list of points y where u is to be evaluated. This `yq` is *not* the B-spline mesh Y. The points `yq` at which u are desired are determined by the quadrature rule used by `ICON` to implement Galerkin's method.

`nyq` - The length of yq.

`IC` must return as *output*

`ui` - The value of u initially at the `xq`(p) and `yq`(q). `ui`(p,q,j) = u_j(`xq`(p),`yq`(q)), for $p = 1, \cdots,$ `nxq`, $q = 1,\ldots,$ `nyq` and $j = 1, \cdots,$ `nu`.

The double precision version of `ICON` is `DICON` with all Real arguments typed double precision.

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

There are many "knobs" in `TTGU`; 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 TTGU through the named Common region

```
Common / TTGUF / Failed ; Logical Failed .
```

Before TTGU calls AF or BC, it sets Failed = .False. . Thus, if the user doesn't use or even know of the existence of TTGUF, TTGU assumes that everything has been correctly computed on return from those subroutines. If, however, the user has a problem, uses TTGUF, and sets Failed = .True., TTGU 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 TTGUF is DTTGUF.

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 as the documentation for TTGR [15] indicates.

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 TTGU. Several examples of knob twiddling that can increase the speed of TTGU considerably are also given.

An exceedingly brief outline of the organization of TTGU 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

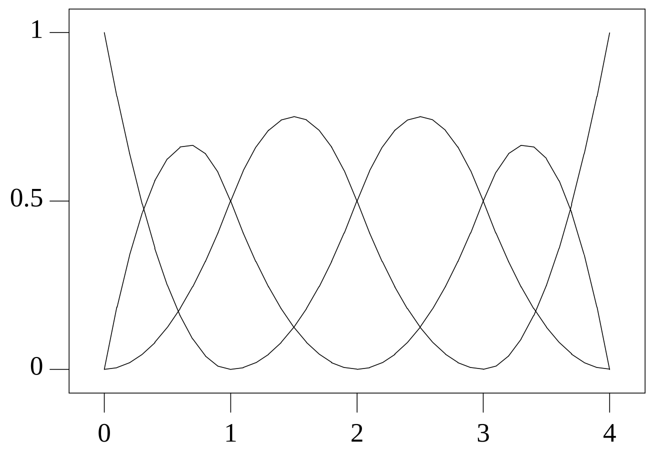
Get integrals for matrix (`AF`), mesh interval by mesh interval,
Get the **bc**s (`BC`).
Get **pde** (`AF`) on the boundary.
Solve the **pde** system.

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`).

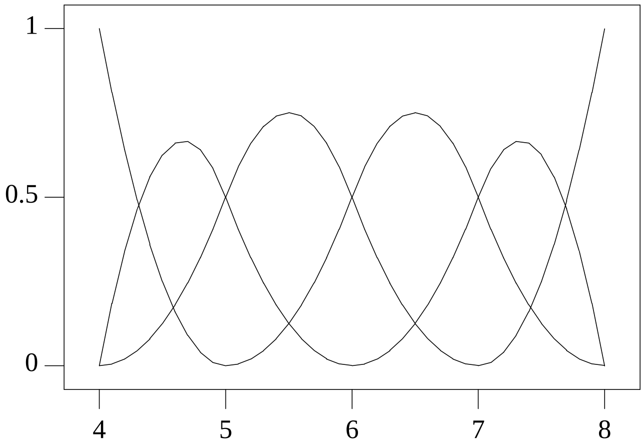
The Linear System

At the bottom level we must solve the linearized Galerkin equations, a linear system. The linear system for each rectangle is a banded system. If one interchanges the x and y coordinates for a rectangle, the resulting system would still be banded, and in general we would like to choose that orientation that would minimize the bandwidth. When one knits together the linear systems for several rectangles, sometimes the band structure of the composite system is preserved; other times it is destroyed. In general while processing an individual rectangle we will take advantage of its band structure and we will arrange the order in which the rectangles are processed to preserve band structure and still keep narrow band widths. Sometimes the geometry of the system is such that we are forced to form a matrix for the equations on the interfaces between rectangles and treat it as a dense matrix.

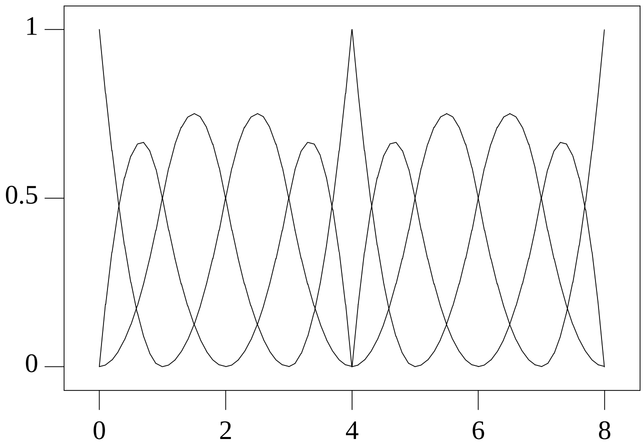
On the interface between rectangles, one would like a multiple knot and the B-splines "add". More explicitly, assume rectangles 1 and 2 meet at $x = 4$. The B splines for rectangle 1 for $k = 3$ would look like



and for rectangle 2 would look like



The B-splines representing the union of the rectangles with a multiple knot of order 2 at $x = 4$ would look like:



Algebraically, the linear system of rectangle 1 either looks like or can be permuted to look like

$$\begin{bmatrix} \mathbf{a} & \mathbf{b} \\ C & \mathbf{d} \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{e} \\ \mathbf{f} \end{bmatrix} \tag{6.1}$$

and the one for rectangle 2 can be permuted to look like

$$\begin{bmatrix} G & H \\ J & K \end{bmatrix} \begin{bmatrix} \mathbf{u}_3 \\ \mathbf{u}_4 \end{bmatrix} = \begin{bmatrix} \mathbf{m} \\ \mathbf{n} \end{bmatrix}. \tag{6.2}$$

The linear system representing the union would look like

$$\begin{bmatrix} \mathbf{a} & \mathbf{b} & \\ C & D+G & H \\ & J & K \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \mathbf{u}_4 \end{bmatrix} = \begin{bmatrix} \mathbf{e} \\ \mathbf{f}+\mathbf{m} \\ \mathbf{n} \end{bmatrix} \tag{6.3}$$

and $\mathbf{u}_3$ is set to $\mathbf{u}_2$, i.e. the solution is forced to match on the interface.

In our code we do not explicitly form (6.3). We know that eventually we wish to form the LU decomposition of (6.3) and that the blocks that make up (6.3) are banded. The LU decomposition of (6.3) is given by

$$\begin{bmatrix} L_{\mathbf{a}} & & \\ C & L_{G'} & \\ & J & L_{K'} \end{bmatrix} \begin{bmatrix} \mathbf{u}_{\mathbf{a}} & L_{\mathbf{a}}^{-1}\mathbf{b} & \\ & \mathbf{u}_{G'} & L_{G'}^{-1}H \\ & & \mathbf{u}_{K'} \end{bmatrix} \tag{6.4}$$

where

$$L_{\mathbf{a}} \mathbf{u}_{\mathbf{a}} = \mathbf{a} \tag{6.5}$$

$$L_{G'} \mathbf{u}_{G'} = G' \equiv \mathbf{d} + G + C L_{\mathbf{a}}^{-1} \mathbf{b} \tag{6.6}$$

$$L_{K'} \mathbf{u}_{K'} = K + J L_{G'}^{-1} H.$$

We can formed the LU decomposition of the banded matrix **a** and then form

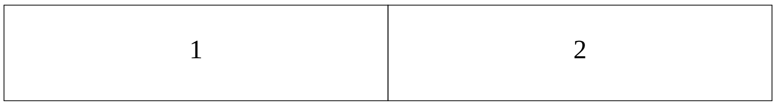
$$\hat{G} = \mathbf{d} + C L_{\mathbf{a}}^{-1} \mathbf{b}. \tag{6.7}$$

The matrix $\hat{G}$ will be dense but the matrix

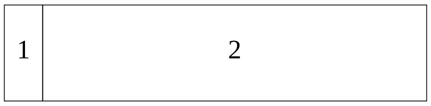
$$\begin{bmatrix} \hat{G} + G & H \\ J & K \end{bmatrix} \tag{6.8}$$

will have the same band width as (6.2) and we treat it the same way.

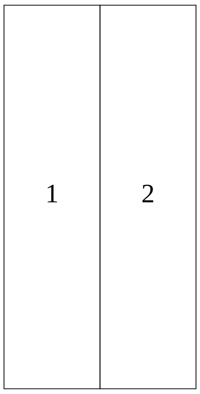
We would like to permute our matrices to minimize band width. The banded form of (6.3) is valid if rectangles 1 and 2 look like



If, in fact, the rectangles looked like



then we may wish to permute the linear system corresponding to rectangle 1 to minimize band width which would mean that the unknown variables representing the solution on the interface will not be at the bottom of the matrix but regularly scattered throughout the matrix. This complicates the programming but the resulting matrix of (6.8) still has a band structure with as narrow a band width as one may wish. However if the union of rectangles had the form



then we would like to permute both matrices and for the union of rectangles we would arrive at a system that looks like

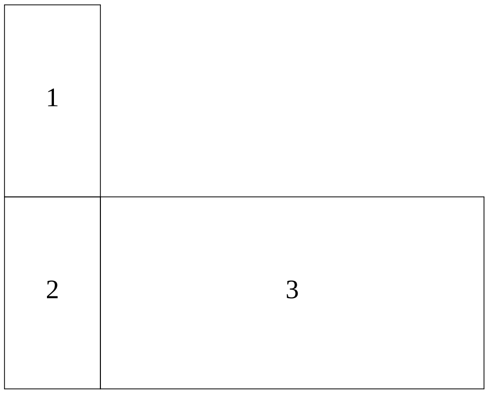
$$\begin{bmatrix} \mathbf{a} & & \mathbf{b} \\ & K & J \\ C & H & \mathbf{d} + G \end{bmatrix}. \tag{6.9}$$

One could use a banded LU decomposition routine to find the LU decompositions of **a** and K, but in order to find the LU decomposition of (6.9) one would also need the LU decomposition of

$$D + G + CL_{\mathbf{a}}^{-1}\mathbf{b} + HL_K^{-1}J \tag{6.10}$$

which is not likely to be banded.

Now let us consider a more complicated geometry. Consider



Processing the above boxes in the order given would mean that we would always be working with banded matrices. However, if we chose to process box "3" first, we would either be forced to separate out the interface variables or permute the equations for box "2" so that they would be banded but would be of wide band width. Because of this, we first perform an analysis to determine a good ordering of the rectangles to preserve narrow band widths and to keep banded structure throughout the computation. When we find that keeping small bandwidth banded systems is impossible we construct a general matrix as in (6.10) to handle all those variables on the interface that do not fit in the banded structure.

Twiddling Procedure Knobs.

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

```
Call TTGUR(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 `TTGU`, is `TTGUE` 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 `TTGU` 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 `TTGUE`.

The error control provided in the subroutine `TTGUE` 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 TTGUE 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

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

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 TTGUR is the same as that for TTGU with the additional possibility that the user may alter errpar through his subprogram ERROR.

The double precision version of TTGUR is DTTGUR, with all Real arguments typed double precision, except errpar, which remains Real. Similarly for TTGUE and TTGUP.

Twiddling non-Procedure Knobs.

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

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

just before calling TTGU will do the trick. The input to TTGUV 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 TTGUV any number of times. The input to TTGUV 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 TTGUV 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 TTGU and Double Precision for DTTGU.
- 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 TTGUV 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 TTGUV.

- theta - The time-discretization parameter, see section 3 and Appendix 2 of [15]. 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 ≠ 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 `TTGU`) / `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 `TTGU` 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 `TTGU` 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.

Default: `LA` = +1. `j` = 2010. `LA` = `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 $N(i) \equiv (\sqrt{2})^{i-1} N(1)$. 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 TTGUV

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
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 TTGUV is DTTGUV, with all Real arguments typed Double precision, except hfract and egive which remain Real.

Run-time Statistics.

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

```
Call TTGUX
```

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

Initial Conditions

The subroutine `ICON` determines the initial values of `U` by calling the underlying static solver in `TTGU` with the problem

$$u - \mathrm{u}(\mathrm{data}) = 0.$$

For one rectangle calling `TSL2W` by E.H. Grosse, as suggested in [15], is appropriate. However, if one called `TSL2W` for more than one rectangle, it is highly unlikely that the initial conditions would agree along interfaces. One really has to consider the total geometry of the system. Calling the bottom level subroutines of `TTGU` with a problem that has no derivative means that some extra work is done in setting up the underlying matrix problem, but unless the order of the approximation is say greater than 4, the most time consuming portion of the computation involves solving the linear system and not constructing the system. Thus using `TTGU` to determine the initial values of the B-spline coefficients of the solution is an easy, not too time-consuming solution. In fact, this is a good way of generally approximating data on a union of rectangles.

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

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

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

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

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

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

[13] 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.

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

[15] L. Kaufman and N.L. Schryer "TTGR-A Package for Solving Partial Differential Equations in Two Space Variables," Bell Laboratories Computing Science Technical Report #135, 1985.

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

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

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

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

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

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

[22] 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).

[23] 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).

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

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

Appendix 1

Examples - Programs

The program examples given below are intended to both illustrate the use of `TTGU` and provide prototypes for a prospective user. Anyone contemplating using `TTGU` would be well advised to pick an example program that invokes those capabilities of `TTGU` the intended problem will require, and type it in (or obtain a copy of the example code from the author). 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 `TTGU` software before proceeding in this appendix, otherwise the reading will be dark and obscure.

Before invoking `TTGU` 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 [9] 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 completely on a uniform mesh, but it would require roughly **105,625** grid points to do it for $k = 2$. Using the combination of non-uniform mesh and uniform mesh scheme given for that problem, 3125 grid points do the job nicely. Remember that the run-time and memory requirements of `TTGU` are proportional to a power of 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 `ICON` available, see section 4.

The Examples

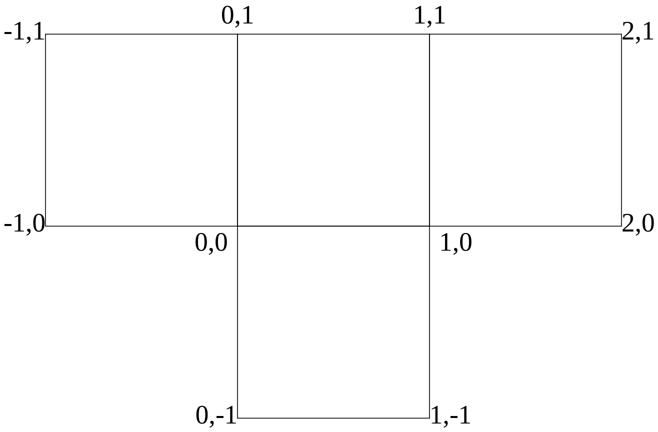
All examples reported here were run on a VAX 11/8550 using double-precision arithmetic, under the UNIX® operating system, Research Tenth Edition.

Example 1 - A Simple Heat Equation.

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

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

on the T-shaped domain



with **bc**s (4.2)

$$\mathbf{b} = u(t,x,y) - t\,x\,y. \tag{A.2}$$

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, solves (A.1)-(A.2) using `TTGU`, with a linear B-spline ($k = 2$) over a spatial mesh consisting of 3 equally spaced, distinct points on each side of each rectangle in the domain with the time evolution carried out to 10^{-2} absolute accuracy.

The main program uses several PORT [9] 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`.
- `IDUMB` 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,4)` sets the pointer `iU` so that locations `Ws(iU), ..., Ws(iU+L-1)` are available for the B-spline coefficients of the solution.

- `SETD` sets an array to a given Double precision constant. `SETD` provides the constant **ic**'s (A.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.

The PORT Library stack is used in a rather interesting way to setup the meshes. Because we have the array IXB, the *x* meshes for all the rectangles do not have to occupy consecutive locations in one long array. One can place a mesh for the first rectangle in an array, skip some spaces, insert the mesh for the second rectangle, etc. The subroutine IDUMB makes uniform meshes in the PORT stack and returns as its value a pointer to the mesh in the stack. In our example with 8 uniform meshes we can use 8 calls to IDUMB to make the meshes and put their beginning locations in IXB and IYB. Since our meshes are now in the stack, when we call DTTGU we use as the X array and Y array, the name of the PORT stack, namely WS.

At the end of each time-step the solution is printed out at at the corners, center, and midpoint of each side for each rectangle in the domain. The main program is

```
c  main program
c to solve the heat equation with solution u == t*x*y,
c   grad . ( u + ux + .1 * uy, u + uy + .1 * ux ) = ut + ux + uy +g(x,t)
      common /cstak/ ds
      double precision ds(350000)
      integer ixb(4), iyb(4), nxr(4), nyr(4), kxr(4), kyr(4)
      external handlu, bc, af
      integer ndx, ndy, istkgt, is(1000), iu, nu, kx, ky, idumb
      real errpar(2), rs(1000)
      logical ls(1000)
      complex cs(500)
      double precision tstart, dt, tstop, ws(500)
c the port library stack and its aliases.
      equivalence (ds(1), cs(1), ws(1), rs(1), is(1), ls(1))
c initialize the port library stack length.
      call istkin(350000, 4)
      call enter(1)
      nu = 1
      kx = 2
      ky = 2
      ndx = 3
      ndy = 3
      nr=4
      tstart = 0.d0
      tstop = 1.d0
      dt = 1
      errpar(1) = 1e-2
      errpar(2) = 1e-4
c uniform grid.
c
c make grid for t-shaped region
c
      ixb(1) = idumb(-1.0d0, 0.0d0, ndx, kx, nxr(1))
      ixb(2) = idumb(0.0d0, 1.0d0, ndx, kx, nxr(2))
      ixb(3) = idumb(1.0d0, 2.0d0, ndx, kx, nxr(3))
      ixb(4) = idumb(0.0d0, 1.0d0, ndx, kx, nxr(4))
      iyb(1) = idumb(0.0d0, 1.0d0, ndy, ky, nyr(1))
      iyb(2) = idumb(0.0d0, 1.0d0, ndy, ky, nyr(2))
      iyb(3) = idumb(0.0d0, 1.0d0, ndy, ky, nyr(3))
```

```
      iyb(4) = idumb(-1.0d0, 0.0d0, ndy, ky, nyr(4))
      nnu =nu*((nxr(1)-kx)*(nyr(1)+nyr(3)-2*ky)+
     1  (nxr(2)-kx)*(nyr(2)-ky)+
     4  (nxr(4)-kx)*(nyr(4)-ky))
      nr=4
c space for the solution.
      iu = istkgt(nnu, 4)
      do 1 i=1,nr
         kxr(i)=kx
         kyr(i)=ky
1     continue
c initial conditions for u.
      call setd(nnu, 0.d0,ws(iu))
c since idumb places the meshes in the port stack, the name of
c the port stack, ws, is used as the x and y arrays
      call dttgu(ws(iu),nu,nr,kxr,ws,nxr,ixb,kyr,ws,nyr,iyb,tstart,
     1   tstop, dt, af, bc, errpar, handlu)
      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 TTGU 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** (A.1) is exactly that given in example 1 for TTGR[15], namely:

```
      subroutine af(t, x, nx, y, ny, nu, u, ut, ux, uy, uxt, uyt
     1   , a, au, aut, aux, auy, auxt, auyt, f, fu, fut, fux, fuy, fuxt,
     2   fuyt)
      integer nu, nx, ny
      double precision t, x(nx), y(ny), u(nx, ny, nu), ut(nx, ny, nu),
     1   ux(nx, ny, nu)
      double precision uy(nx, ny, nu), uxt(nx, ny, nu), uyt(nx, ny, nu),
     1   a(nx, ny, nu, 2), au(nx, ny, nu, nu, 2), aut(nx, ny, nu, nu, 2)
      double precision aux(nx, ny, nu, nu, 2), auy(nx, ny, nu, nu, 2),
     1   auxt(nx, ny, nu, nu, 2), auyt(nx, ny, nu, nu, 2), f(nx, ny, nu)
     2   , fu(nx, ny, nu, nu)
      double precision fut(nx, ny, nu, nu), fux(nx, ny, nu, nu), fuy(nx,
     1   ny, nu, nu), fuxt(nx, ny, nu, nu), fuyt(nx, ny, nu, nu)
      integer i, p, q
      do  3 i = 1, nu
         do  2 q = 1, ny
            do  1 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)
 1          continue
 2       continue
 3    continue
      return
      end
```

Note that since the arrays B, ... , BUyt are set to zero by TTGU 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 (A.2) is exactly the same as in Example 1 for TTGR[15], namely:

```
      subroutine bc(t, x, nx, y, ny, lx, rx, ly, ry, u, ut, ux,
     1   uy, uxt, uyt, nu, b, bu, but, bux, buy, buxt, buyt)
      integer nu, nx, ny
      double precision t, x(nx), y(ny), lx, rx, ly
      double precision ry, u(nx, ny, nu), ut(nx, ny, nu), ux(nx, ny, nu)
     1   , uy(nx, ny, nu), uxt(nx, ny, nu)
      double precision uyt(nx, ny, nu), b(nx, ny, nu), bu(nx, ny, nu,
     1   nu), but(nx, ny, nu, nu), bux(nx, ny, nu, nu), buy(nx, ny, nu
     2   , nu)
      double precision buxt(nx, ny, nu, nu), buyt(nx, ny, nu, nu)
      integer i, j
      do  2 j = 1, ny
         do  1 i = 1, nx
            bu(i, j, 1, 1) = 1
            b(i, j, 1) = u(i, j, 1)-t*x(i)*y(j)
 1          continue
 2       continue
      return
      end
```

The following output subroutine simply evaluates and prints $u(t,x,y)$, for three rows and three columns in each rectangle of the domain 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).
- DTSD1 evaluates a spline, given the mesh and the coefficients. See section 4.
- IDLUMD 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 TTGU 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. More specifically we put the information for the KXR, X, NXR, KYR, Y, and NYR arrays into the PORT Library stack and KXP, IX, NXP, KYP, IY, and NYP point to the beginnings of the respective arrays in the stack. NXNYT gives the total number of mesh points in the domain, NR indicates the number of rectangles in the domain and IUP points to a location in the stack which starts an array whose elements indicate the position in the U array where the coefficients are stored for successive rectangles.

```
      subroutine handlu(t0, u0, t, u, nv, dt, tstop)
```

```
      integer nv
      double precision t0, u0(nv), t, u(nv), dt, tstop
      common /d7tgup/ errpar, nu, mxp, myp
      integer nu
      real errpar(2)
      common /d7tgum/  kxp,ix,nxp,kyp,iy,nyp,nxnyt,nr,iup
      integer kx, ix, nx, ky, iy, ny
      common /cstak/is
      integer is(1000)
      iwrite=i1mach(2)
      if (t0 .ne. t) goto 2
         write (iwrite,  1) t
   1     format (16h restart for t =, 1pe10.2)
         return
c get and print the error.
   2    continue
        write(iwrite, 3)t
   3    format(6h at t=,1pe10.2)
        ius=1
        do 5 inu = 1, nu
           iyr=iy
           ixr=ix
           do 4 ir=1,nr
              ir1=ir-1
              nx=is(nxp+ir1)
              ny=is(nyp+ir1)
              kx=is(kxp+ir1)
              ky=is(kyp+ir1)
              call gerr(kx, ixr, nx, ky, iyr, ny, u(ius), inu, t, ir)
              ixr=ixr+nx
              iyr=iyr+ny
              ius=ius+(nx-kx)*(ny-ky)
   4         continue
   5  continue
      return
      end
```

where the procedure `GERR` is used to print out the results

```
      subroutine gerr(kx, ix, nx, ky, iy, ny, u, inu, t, ir)
c to print the solution at each time-step
      integer kx, ix, nx, ky, iy, ny
      integer inu, ir
      double precision u(1), t
      common /cstak/ ds
      double precision ds(500)
      integer ifa, ita(2), ixa(2), nta(2), nxa(2), idlumd
      integer ixs, iys, nxs, nys, istkgt, i
      integer  ka(2), ma(2), is(1000), i1mach
      real rs(1000)
      logical ls(1000)
      complex cs(500)
      double precision  ws(500)
      integer temp
```

```
      equivalence (ds(1), cs(1), ws(1), rs(1), is(1), ls(1))
c u(nx-kx,ny-ky).
c the port library stack and its aliases.
      call enter(1)
c x search grid.
c find the solution at 2 * 2 points / mesh rectangle.
      ixs = idlumd(ws(ix), nx, 2, nxs)
c y search grid.
      iys = idlumd(ws(iy), ny, 2, nys)
c u search grid values.
      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
c get solution.
c approximate solution values.
      ifa = istkgt(nxs*nys, 4)
c evaluate them.
      call dtsd1(2, ka, ws, ita, nta, u, ws, ixa, nxa, ma, ws(ifa))
      temp = i1mach(2)
      write(temp,9001)ir,inu,(ws(i),i=iFA,IFa+nxs*nys-1)
9001  format(" for rect",i3," u(.,",i2,")=",
     1((1p5e10.2/20x,1p4d10.2)))
      call leave
      return
      end
```

The output of the above program unit is

```
 at t=  1.00E+00
 for rect  1 u(., 1)=  1.58E-19  5.25E-35 -1.31E-35 -5.00E-01 -2.50E-01
                      6.85E-19 -1.00E+00 -5.00E-01  3.49E-19
 for rect  2 u(., 1)= -1.31E-35 -3.00E-19 -1.37E-19  6.85E-19  2.50E-01
                      5.00E-01  3.49E-19  5.00E-01  1.00E+00
 for rect  3 u(., 1)= -1.37E-19  1.89E-34 -3.15E-19  5.00E-01  7.50E-01
                      1.00E+00  1.00E+00  1.50E+00  2.00E+00
 for rect  4 u(., 1)=  0.00E+00 -5.00E-01 -1.00E+00  0.00E+00 -2.50E-01
                     -5.00E-01 -1.31E-35 -3.00E-19 -1.37E-19
 USED     5078 /   700000 OF THE STACK ALLOWED.
```

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, inu, t, ir)
```

```
      integer kx, ix, nx, ky, iy, ny, inu, ir
      double precision u(1), t
      common /cstak/ ds
      double precision ds(500)
      integer ifa, ita(2), ixa(2), nta(2), nxa(2), idlumd
      integer ixs, iys, nxs, nys, istkgt, i
      integer iewe, ka(2), ma(2), is(1000), i1mach
      real rs(1000)
      logical ls(1000)
      complex cs(500)
      double precision dabs, erru, dmax1, ws(500)
      integer temp, temp1, temp2
      equivalence (ds(1), cs(1), ws(1), rs(1), is(1), ls(1))
c to get and print the error at each time-step.
c for variable inu for rectangle ir
c u(nx-kx,ny-ky).
c the port library stack and its aliases.
      call enter(1)
c find the error in the solution at 2*kx * 2*ky points / mesh rectangle.
c x search grid.
      ixs = idlumd(ws(ix), nx, 2*kx, nxs)
c y search grid.
      iys = idlumd(ws(iy), ny, 2*ky, nys)
c u search grid values.
      iewe = istkgt(nxs*nys, 4)
c the exact solution.
      call ewe2(t, ws(ixs), nxs, ws(iys), nys, ws(iewe), inu, ir)
      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
c get solution.
      ma(2) = 0
c approximate solution values.
      ifa = istkgt(nxs*nys, 4)
c evaluate them.
      call dtsd1(2, ka, ws, ita, nta, u, ws, ixa, nxa, ma, ws(ifa))
c error in solution values.
      erru = 0
      temp = nxs*nys
      do  1 i = 1, temp
         temp2 = iewe+i
         temp1 = ifa+i
         erru = dmax1(erru, dabs(ws(temp2-1)-ws(temp1-1)))
   1     continue
      temp = i1mach(2)
      write (temp,  2) ir, inu, erru
```

```
   2  format(9h for rect,i3,14h error in u(.,,  i2,
     1       3h) =, 1pe10.2)
      call leave
      return
      end
```

and the procedure EWE2 below evaluates the exact solution at any position in time and space.

```
      subroutine ewe2(t, x, nx, y, ny, u, inu, ir)
      integer inu, ir, nx, ny
      double precision t, x(nx), y(ny), u(nx, ny)
      integer i, j
c the exact solution.
         do  2 i = 1, nx
            do  1 j = 1, ny
               u(i, j) = t*x(i)*y(j)
   1           continue
   2        continue
      return
      end
```

The above program unit gives

```
 at t=  1.00E+00
 for rect  1 error in u(., 1) =  6.94E-18
 for rect  2 error in u(., 1) =  6.94E-18
 for rect  3 error in u(., 1) =  2.78E-17
 for rect  4 error in u(., 1) =  6.94E-18
 USED     5078 /   700000 OF THE STACK ALLOWED.
```

and we can see that (A.1)-(A.2) has indeed been solved to within rounding error.

The only differences between the user written code for Example 1 of TTGR[15] and that of Example 1 here, which solves the problem on a T-shaped domain rather than a rectangle, are several statements in the main program which defines the mesh and the subroutine HANDLU is different from HANDLE. The subroutines AF and BC have not been changed.

Example 2 - A Coupled System of pdes.

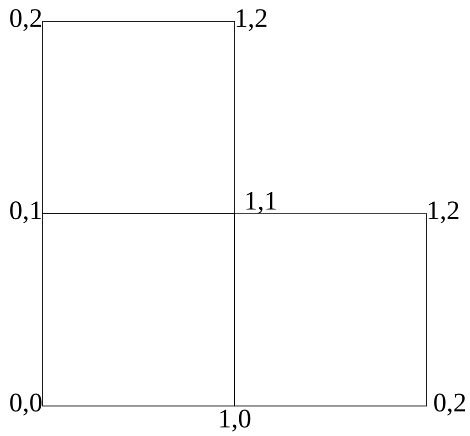
The solution of a coupled system of **pde**'s is now illustrated. The **pde** is

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

with **bc**s

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

on the L-shaped domain



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

The following program solves (A.3)-(A.4) using `TTGU`, with a cubic B-spline, over a spatial mesh consisting of 3 equally spaced, distinct points each side of each rectangle 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

```
c  main program
      common /cstak/ ds
      double precision ds(350000)
      external handlu, bc, af
      integer ndx, ndy, istkgt, is(1000), iu
      integer nu, nr, iyb(3), ixb(3), kx, ky
      integer nxr(3), nyr(3), kxr(3), kyr(3)
      integer idumb
      real errpar(2), rs(1000)
      logical ls(1000)
      complex cs(500)
      double precision tstart, dt
      double precision ws(500), tstop
      equivalence (ds(1), cs(1), ws(1), rs(1), is(1), ls(1))
c to solve two coupled, nonlinear heat equations.
c   u1 sub t = div . ( u1x, u1y ) - u1*u2 + g1
c   u2 sub t = div . ( u2x, u2y ) - u1*u2 + g2
c the port library stack and its aliases.
c initialize the port library stack length.
      call istkin(350000, 4)
      call enter(1)
      nu = 2
      kx = 4
      ky = 4
      ndx = 3
      ndy = 3
      nr = 3
      tstart = 0
      dt = 1e-2
      tstop =1.d0
      errpar(1) = 1e-2
      errpar(2) = 1e-4
c uniform grid.
      ixb(1) = idumb(0.0d0, 1.0d0, ndx, kx, nxr(1))
```

```
      ixb(2) = idumb(0.0d0, 1.0d0, ndx, kx, nxr(2))
      ixb(3) = idumb(1.0d0, 2.0d0, ndx, kx, nxr(3))
      iyb(1) = idumb(0.0d0, 1.0d0, ndy, ky, nyr(1))
      iyb(2) = idumb(1.0d0, 2.0d0, ndy, ky, nyr(2))
      iyb(3) = idumb(0.0d0, 1.0d0, ndy, ky, nyr(3))
c uniform grid.
c space for the solution.
      nnu=0
      do 1 i=1,nr
         nnu=nnu+nu*((nxr(i)-kx)*(nyr(i)-ky))
 1    continue
      iu = istkgt(nnu, 4)
      do 2 i=1,nr
         kxr(i)=kx
         kyr(i)=ky
 2    continue
      call setd(nnu, 1.d0,ws(iu))
      call dttgu(ws(iu),nu,nr,kxr,ws,nxr,ixb,kyr,ws,nyr,iyb,tstart,
     1   tstop, dt, af, bc, errpar, handlu)
      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
      double precision dexp
      do  2 q = 1, ny
         do  1 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)-(dexp(t*(x(p)-y(q)))*(x(p)-y(q)-2d0*
     1         t*t)+1d0)
            f(p, q, 2) = f(p, q, 2)-(dexp(t*(y(q)-x(p)))*(y(q)-x(p)-2d0*
     1         t*t)+1d0)
   1        continue
   2     continue
```

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

```
      double precision dexp
      do  2 j = 1, ny
         do  1 i = 1, nx
            bu(i, j, 1, 1) = 1
            b(i, j, 1) = u(i, j, 1)-dexp(t*(x(i)-y(j)))
            bu(i, j, 2, 2) = 1
            b(i, j, 2) = u(i, j, 2)-dexp(t*(y(j)-x(i)))
   1        continue
   2     continue
```

The HANDLU subroutine simply checks the accuracy of the computed solution and is the same as the one in example 1 which checks the solution. The following body for the subroutine EWE2 computes the exact solution.

```
      double precision dble, dexp
c the exact solution.
         do  2 i = 1, nx
            do  1 j = 1, ny
               u(i, j) = dexp(dble(float((-1)**(inu+1)))*t*(x(i)-y(j)))
   1           continue
   2        continue
```

The output of the above program unit is

```
 at t=  1.00E-02
 for rect  1 error in u(., 1) =  1.39E-05
 for rect  2 error in u(., 1) =  5.48E-05
 for rect  3 error in u(., 1) =  5.58E-05
 for rect  1 error in u(., 2) =  1.39E-05
 for rect  2 error in u(., 2) =  5.58E-05
 for rect  3 error in u(., 2) =  5.48E-05
 at t=  9.57E-02
 for rect  1 error in u(., 1) =  5.49E-04
 for rect  2 error in u(., 1) =  1.27E-03
 for rect  3 error in u(., 1) =  1.54E-03
 for rect  1 error in u(., 2) =  5.49E-04
 for rect  2 error in u(., 2) =  1.54E-03
 for rect  3 error in u(., 2) =  1.27E-03
 at t=  2.80E-01
 for rect  1 error in u(., 1) =  2.00E-03
 for rect  2 error in u(., 1) =  2.73E-03
 for rect  3 error in u(., 1) =  5.02E-03
 for rect  1 error in u(., 2) =  2.00E-03
 for rect  2 error in u(., 2) =  5.02E-03
 for rect  3 error in u(., 2) =  2.73E-03
 at t=  5.49E-01
 for rect  1 error in u(., 1) =  4.16E-03
 for rect  2 error in u(., 1) =  3.04E-03
 for rect  3 error in u(., 1) =  1.09E-02
 for rect  1 error in u(., 2) =  4.16E-03
```

```
for rect  2 error in u(., 2) =  1.09E-02
for rect  3 error in u(., 2) =  3.04E-03
at t=  8.92E-01
for rect  1 error in u(., 1) =  7.70E-04
for rect  2 error in u(., 1) =  2.39E-04
for rect  3 error in u(., 1) =  1.52E-03
for rect  1 error in u(., 2) =  7.70E-04
for rect  2 error in u(., 2) =  1.52E-03
for rect  3 error in u(., 2) =  2.39E-04
at t=  1.00E+00
for rect  1 error in u(., 1) =  1.98E-03
for rect  2 error in u(., 1) =  5.24E-04
for rect  3 error in u(., 1) =  7.31E-03
for rect  1 error in u(., 2) =  1.98E-03
for rect  2 error in u(., 2) =  7.31E-03
for rect  3 error in u(., 2) =  5.24E-04
USED    57108 /   700000 OF THE STACK ALLOWED.
```

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

Example 3 - Interfaces.

Consider the **pde**

$$a^{(1)} = \kappa\,(x,y)\,\mathbf{u}_x$$
$$a^{(2)} = \kappa\,(x,y)\,\mathbf{u}_y \tag{A.5}$$
$$f = \mathbf{u}_t - g$$

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

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

and those on the sides

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

The following program solves (A.5)-(A.6) using `TTGU`, with a linear B-spline ($k = 2$) over a domain that consists of 3 rectangles on top of each other each of unit height and width. The spatial meshes on each rectangle consists of 3 equally spaced points on each side of each rectangle. The time evolution is 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.

This problem is also example 3 in the documentation of TTGR[15]. For TTGR one was limited to one rectangle and to handle the interface conditions, multiple knots were inserted. For TTGU there is no reason to insert multiple knots.

The main program is

```
c  main program
```

```
      common /cstak/ ds
      double precision ds(350000)
      external handlu, bc, af
      integer ndx, ndy, istkgt, is(1000), iu
      integer nu, nr, iyb(3), ixb(3), kx, ky
      integer nxr(3), nyr(3), kxr(3), kyr(3)
      integer idumb
      real errpar(2), rs(1000)
      logical ls(1000)
      complex cs(500)
      double precision tstart, dt, ws(500)
      double precision tstop
      equivalence (ds(1), cs(1), ws(1), rs(1), is(1), ls(1))
c to solve the layered heat equation, with kappa = 1, 1/2, 1/3,
c   div . ( kappa(x,y) * grad u ) = ut + g
c the port library stack and its aliases.
c initialize the port library stack length.
      call istkin(350000, 4)
      call enter(1)
      nu = 1
      nr = 3
      kx = 2
      ky = 2
      ndx = 3
      ndy = 3
      tstart = 0
      tstop = 1
      dt = 1
      errpar(1) = 1e-2
      errpar(2) = 1e-4
c uniform grid.
      ixb(1) = idumb(0.0d0, 1.0d0, ndx, kx, nxr(1))
      ixb(2) = idumb(0.0d0, 1.0d0, ndx, kx, nxr(2))
      ixb(3) = idumb(0.0d0, 1.0d0, ndx, kx, nxr(3))
      iyb(1) = idumb(0.0d0, 1.0d0, ndy, ky, nyr(1))
      iyb(2) = idumb(1.0d0, 2.0d0, ndy, ky, nyr(2))
      iyb(3) = idumb(2.0d0, 3.0d0, ndy, ky, nyr(3))
c space for the solution.
      nnu=0
      do 1 i=1,nr
         nnu=nnu+nu*((nxr(i)-kx)*(nyr(i)-ky))
 1    continue
      iu = istkgt(nnu, 4)
      do 2 i=1,nr
         kxr(i)=kx
         kyr(i)=ky
 2    continue
      call setd(nnu, 0.d0,ws(iu))
      call dttgu(ws(iu),nu,nr,kxr,ws,nxr,ixb,kyr,ws,nyr,iyb,tstart,
     1   tstop, dt, af, bc, errpar, handlu)
      call leave
      call wrapup
      stop
      end
```

The body of the AF and BC subroutines for specifying the **pde** (A.5) and the **bc** (A.7) respectively are exactly the same as those given for example 3 in the documentation of TTGR[15].

There is no change in the HANDLU or GERR subroutines of example 1. The only change in the body of the subroutine EWE2, for computing *u*, of example 2 is the code for computing *u*,

```
u(i, j) = dble(float(ir))*t*y(j)-dble(float(ir-1))*t
if(ir.eq.3) u(i,j)=u(i,j)-t
```

The output from this program is

```
 at t=  1.00E+00
 for rect  1 error in u(., 1) =  1.39E-17
 for rect  2 error in u(., 1) =  5.55E-17
 for rect  3 error in u(., 1) =  1.11E-16
 USED     4006 /   700000 OF THE STACK ALLOWED.
```

and we see that (A.5)-(A.6) has indeed been solved to rounding error.

Example 4 - Determining the initial conditions

The PDE and spatial domain of this example is exactly the same as that of Example 2. The only difference is that in example 2 we considered $0 \le t \le 1.0$, which meant the *u* initially was a constant and that in this example we wish that $1.0 \le t \le 1.01$ so that *u* is not a constant initially. To help us determine the initial B-spline coefficients we call the subroutine DICON documented in section 5 of this document. This actually entails a one line change in the main program of example 2 besides the changes to TSTART and TSTOP. In the program below after the initial B-spline coefficients are computed, we call GERR to determine the error in the pde before calling DTTGU to solve the pde.

```
c  main program
      common /cstak/ ds
      double precision ds(350000)
      external handlu, bc, af, ic
      integer ndx, ndy, istkgt, is(1000), iu
      integer nu, nr, iyb(3), ixb(3), kx, ky
      integer nxr(3), nyr(3), kxr(3), kyr(3)
      integer idumb
      real errpar(2), rs(1000)
      logical ls(1000)
      complex cs(500)
      double precision tstart, dt
      double precision ws(500), tstop
      equivalence (ds(1), cs(1), ws(1), rs(1), is(1), ls(1))
c to solve two coupled, nonlinear heat equations.
c   u1 sub t = div . ( u1x, u1y ) - u1*u2 + g1
c   u2 sub t = div . ( u2x, u2y ) - u1*u2 + g2
c the port library stack and its aliases.
c initialize the port library stack length.
      call istkin(350000, 4)
      call enter(1)
```

```
      nu = 2
      kx = 4
      ky = 4
      ndx = 3
      ndy = 3
      nr = 3
      tstart = 1.0d0
      dt = 1e-2
      tstop =1.01d0
      errpar(1) = 1e-2
      errpar(2) = 1e-4
c uniform grid.
      ixb(1) = idumb(0.0d0, 1.0d0, ndx, kx, nxr(1))
      ixb(2) = idumb(0.0d0, 1.0d0, ndx, kx, nxr(2))
      ixb(3) = idumb(1.0d0, 2.0d0, ndx, kx, nxr(3))
      iyb(1) = idumb(0.0d0, 1.0d0, ndy, ky, nyr(1))
      iyb(2) = idumb(1.0d0, 2.0d0, ndy, ky, nyr(2))
      iyb(3) = idumb(0.0d0, 1.0d0, ndy, ky, nyr(3))
c uniform grid.
c space for the solution.
      nnu=0
      do 1 i=1,nr
         nnu=nnu+nu*((nxr(i)-kx)*(nyr(i)-ky))
 1    continue
      iu = istkgt(nnu, 4)
      do 2 i=1,nr
         kxr(i)=kx
         kyr(i)=ky
 2    continue
      call dicon(ws(iu),nu,nr,kxr,ws,nxr,ixb,kyr,ws,nyr,iyb,ic)
      iu1=iu
      iwrite=i1mach(2)
      write(iwrite,3)
3     format(10h initially)
      do 5 inu=1,nu
         do 4 i=1,nr
            call gerr(kxr(i),ixb(i),nxr(i),kyr(i),iyb(i),nyr(i),
     1      ws(iu1),inu,1.0d0,i)
            iu1=iu1+(nxr(i)-kxr(i))*(nyr(i)-kyr(i))
4        continue
5     continue
      call dttgu(ws(iu),nu,nr,kxr,ws,nxr,ixb,kyr,ws,nyr,iyb,tstart,
     1   tstop, dt, af, bc, errpar, handlu)
      call leave
      call wrapup
      stop
      end
```

The subroutines AF and BC are exactly those given in example 2. To determine the initial conditions we also have to supply a subroutine IC to compute the initial conditions at specific points. This subroutine is called once per rectangle.

```
      subroutine ic(nu,ir,xq,nxq,yq,nyq,ui)
      integer nu, ir, nxq, nyq
```

```
      double precision xq(nxq), yq(nyq), ui(nxq, nyq,nu)
      double precision dble, dexp
      integer p
      do 30 p=1,nu
        do 20 j=1,nyq
           do 10 i=1, nxq
             ui(i, j, p) = dexp(dble(float((-1)**(p+1)))*(xq(i)-yq(j)))
10         continue
20      continue
30    continue
      return
      end
```

When the subroutines HANDLU, GERR, and EWE2 of example 2 were used, the following output was produced:

```
 initially
 for rect  1 error in u(., 1) =  3.70E-04
 for rect  2 error in u(., 1) =  1.39E-04
 for rect  3 error in u(., 1) =  9.90E-04
 for rect  1 error in u(., 2) =  3.70E-04
 for rect  2 error in u(., 2) =  9.90E-04
 for rect  3 error in u(., 2) =  1.39E-04
 at t=  1.01E+00
 for rect  1 error in u(., 1) =  1.98E-04
 for rect  2 error in u(., 1) =  7.21E-05
 for rect  3 error in u(., 1) =  6.54E-04
 for rect  1 error in u(., 2) =  1.98E-04
 for rect  2 error in u(., 2) =  6.54E-04
 for rect  3 error in u(., 2) =  7.21E-05
 USED    56508 /   700000 OF THE STACK ALLOWED.
```

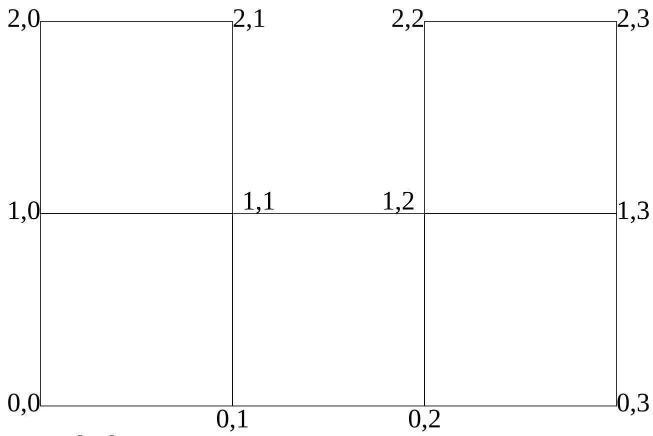
Notice that initially the B spline coefficients do not give the exact initial conditions.

Example 5 - A Static Problem.

Consider the **pde**

$$\begin{aligned} a^{(1)} &= \mathbf{u}_x \\ a^{(2)} &= \mathbf{u}_y \\ f &= 0 \end{aligned} \tag{A.9}$$

on the domain



with **bc**s on the bottom

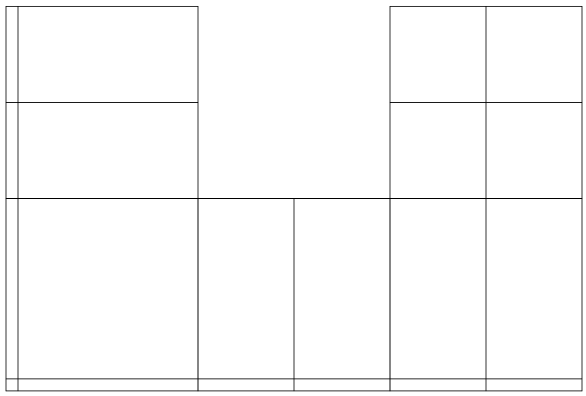
$$\mathbf{b} = u_y \tag{A.10a}$$

and on the other sides

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

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

The following program solves the **pde-bc** combination (A.9)-(A.10), using cubic B-splines over a mesh consisting of 3 non-uniformly spaced, distinct points whenever $0 \le x \le 1$ and whenever $0 \le y \le 1$ and a uniform mesh elsewhere 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 $i = 1, \cdots, n$. The non-uniform mesh is used in the x direction in rectangles 1 and 2 and in the y direction in rectangles 1, 3, and 4. A uniform mesh is used in the x direction in rectangles 3, 4, and 5 and in the y direction in rectangles 2 and 5. Thus the mesh would look like



This unusual grid has been used for several reasons. In the first place, it demonstrates that one can use different grids on different rectangles, but currently, on contiguous rectangles the grids must match. Thus in the x direction one could not have a different mesh in rectangles 1 and 2. Secondly, since the solution has a $\log(z)$ singularity at $z = 0$, the non uniform grading of the mesh in rectangle 1 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. However, as our data will indicate, the singularity has little effect on rectangle 5, so that the need for a non uniform mesh is not as great.

In the main program below the error at each time-step is printed out to confirm the accuracy of the computed solution.

```
c  main program
      common /cstak/ ds
      double precision ds(350000)
      external handlu, bc, af
      integer ndx, ndy, istkgt, is(1000), iu, ix, temp, temp1
      integer nu, nr, iyb(5), ixb(5), kx, ky
      integer nxr(5), nyr(5), kxr(5), kyr(5)
      integer idumb
      real errpar(2), rs(1000)
      logical ls(1000)
      complex cs(500)
      double precision tstart, dt, rx
      double precision ws(500), tstop
      equivalence (ds(1), cs(1), ws(1), rs(1), is(1), ls(1))
c to solve laplaces equation with real ( z*log(z) ) as solution.
c the port library stack and its aliases.
c initialize the port library stack length.
      call istkin(350000, 4)
      call enter(1)
      nu = 1
      kx = 4
      ky = 4
      ndx = 3
      ndy = 3
      nr = 5
      tstart = 0
      dt = 1.d0
      tstop =1.d0
      errpar(1) = 1e-2
      errpar(2) = 1e-4
      nx = ndx+2*(kx-1)
      rx=1.0d0
c space for x mesh for rectangle 1
      ix = istkgt(nx, 4)
c 0 and rx mult = kx.
      ixb(1)=ix
      do  1 i = 1, kx
         temp = ix+i
         ws(temp-1) = 0
         temp = ix+nx-i
         ws(temp) = rx
   1     continue
      temp = ndx-1
      do  2 i = 1, temp
         temp1 = ix+kx-2+i
         ws(temp1) = rx*(dble(float(i-1))/(dble(float(ndx))-1d0))**kx
   2     continue
c rectangle 2 has same grid in x direction as rectangle 1
      ixb(2)=istkgt(nx, 4)
      call dcopy(nx, ws(ix), 1, ws(ixb(2)), 1)
c uniform grid for rectangles 3,4, and 5 in x direction
      ixb(3) = idumb(1.0d0, 2.0d0, ndx, kx, nxr(3))
```

```
      ixb(4) = idumb(2.0d0, 3.0d0, ndx, kx, nxr(4))
      ixb(5) = idumb(2.0d0, 3.0d0, ndx, kx, nxr(5))
      ny = ndy+2*(ky-1)
c rectangles 1,3, and 4 use the same grid in the y direction as
c is used for the x direction in rectangle 1
c space for y mesh.
      iyb(1) = istkgt(ny, 4)
      call dcopy( nx, ws(ix), 1, ws(iyb(1)), 1)
      iyb(3) =istkgt(ny, 4)
      call dcopy( nx, ws(ix), 1, ws(iyb(3)), 1)
      iyb(4) =istkgt(ny, 4)
      call dcopy( nx, ws(ix), 1, ws(iyb(4)), 1)
c rectangles 2 and 5 use uniform mesh in y direction
      iyb(2) = idumb(1.0d0, 2.0d0, ndy, ky, nyr(2))
      iyb(5) = idumb(1.0d0, 2.0d0, ndy, ky, nyr(5))
c space for the solution.
      nnu=0
      do 3 i=1,nr
         nxr(i)=nx
         nyr(i)=ny
         nnu=nnu+nu*((nxr(i)-kx)*(nyr(i)-ky))
 3    continue
      iu = istkgt(nnu, 4)
      do 4 i=1,nr
         kxr(i)=kx
         kyr(i)=ky
 4    continue
      call setd(nnu, 0.0d0,ws(iu))
      call istkck
      write(6,25)nnu
25     format(" nnu",i5)
      call dttgu(ws(iu),nu,nr,kxr,ws,nxr,ixb,kyr,ws,nyr,iyb,tstart,
     1   tstop, dt, af, bc, errpar, handlu)
      call leave
      call wrapup
      stop
      end
```

The body of the AF and BC subroutines for specifying the **pde** (A.9) and the **bc** (A.10) respectively are exactly the same as those given for example 5 in the documentation of TTGR[15].

There is no change in the HANDLU or GERR subroutines of example 1. The body of the subroutine EWE2, for computing *u*, is changed as follows:

```
      double precision r, dcos, dlog, dsin, datan, theta
      double precision dsqrt
c the exact solution.
         do  6 i = 1, nx
            do  5 j = 1, ny
               r = dsqrt(x(i)**2+y(j)**2)
               if (x(i) .le. 0d0) goto 1
                  theta = datan(y(j)/x(i))
                  goto  2
   1              theta = 2d0*datan(1d0)
```

```
  2          if (r .le. 0d0) goto 3
                u(i, j) = r*(dcos(theta)*dlog(r)-theta*dsin(theta))
                goto  4
  3             u(i, j) = 0
  4          continue
  5          continue
  6       continue
```

The output of this program is

```
 at t=  1.00E+00
 for rect  1 error in u(., 1) =  1.48E-02
 for rect  2 error in u(., 1) =  2.12E-03
 for rect  3 error in u(., 1) =  3.20E-03
 for rect  4 error in u(., 1) =  7.18E-05
 for rect  5 error in u(., 1) =  1.64E-05
 USED    26612 /   700000 OF THE STACK ALLOWED.
```

One notices that the singularity affects rectangle 1 much more than it affects the other rectangles. After the above program was executed, the mesh was refined by changing NDX and NDY to 5 and the program was run again with the following result:

```
 for rect  1 error in u(.,  1.00E+00, 1) =  3.21E-03
 for rect  2 error in u(.,  1.00E+00, 1) =  2.93E-04
 for rect  3 error in u(.,  1.00E+00, 1) =  2.52E-04
 for rect  4 error in u(.,  1.00E+00, 1) =  2.48E-05
 for rect  5 error in u(.,  1.00E+00, 1) =  7.02E-06
 USED    75804 /   700000 OF THE STACK ALLOWED.
```

Appendix 2

Routine, Common and Error State Summary

This appendix summarizes the calling sequences for the routines of TTGU, 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 TTGU is

```
Call TTGU(U,Nu,Nr,kx,x,nx,ixb, ky,y,ny,iyb,
          tstart,tstop,dt,
          AF,BC,
          errpar,
          HANDLU)
```

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 HANDLU procedure has the form

```
Subroutine HANDLU(t0,U0,V0,t,U,V,Nu,dt,tstop)
```

The "Return-End" HANDLU procedure is TTGUH.
The "Return-End" BC procedure is TTGUP, for when $n_u = 0$.
The statistics printing procedure is TTGUX.

The basic knob twiddling routine for TTGU is

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

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

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
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 TTGUR(U,Nu,Nr,kx,x,ixb,nx,ky,y,iyb,ny,
            tstart,tstop,dt,
            AF,BC,
            ERROR,errpar,
            HANDLU)
```

where the ERROR procedure has the form

```
  Logical Function ERROR(U,Nu,t,dt,
                         errpar,
                         erputs,
                         eU)
```

Common Regions

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

```
  Common / TTGUF / Failed; Logical Failed
```

Naming Conventions

The naming convention for TTGU 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 [9] that may be encountered when using TTGU. 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 HANDLU. For each level of (entry to) TTGU, the error message and number for a given error state is the same, always reflecting the error from the bottom layer of TTGU. The list of error states below, along with interpretation, is the complete set of error states for the TTGU package as obtained from the lowest level of TTGU.

There are many internal variables of TTGU that may be controlled by the subroutine TTGUV. Thus, there are many more ways to call TTGU with bad data than just the obvious ones involving data in the calling sequence for TTGU. The list of error states given below is complete for TTGU/TTGUV, and therefore involves variables not mentioned in the calling sequence for TTGU. If you are only using TTGU, and not TTGUV as well, then the error states mentioning variables not in the TTGU 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. in some rectangle

3 nx .lt. 2*kx. in some rectangle

4 ky .lt. 2. in some rectangle

5 ny .lt. 2*ky. in some rectangle

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.

16 nr.lt.1

17 Disagreement of meshes along interfaces

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 `pde`s is of zero order.

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

1005 Improper BCs. Recoverable. The **bc**s and `pde`s 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.

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.