Computational Fluid Dynamics
Hands on Workshop

All necessary example code, plus these instructions and supplemental information, are located on the Tuesday Workshop page of the CY902N course.

This workshop focuses on simulating Burger's equation on an interval $0 < x < \ell$ subject to "stress-free" (Neumann) boundary conditions:

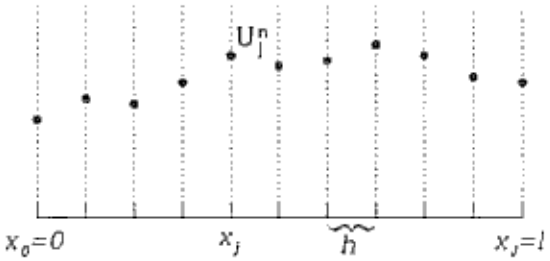
$$\text{Burger's Equation:} \qquad \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}$$

$$\text{Boundary Conditions:} \qquad \frac{\partial u}{\partial x}(0,t) = 0 \quad \text{and} \quad \frac{\partial u}{\partial x}(\ell,t) = 0$$

The initial conditions at $t = 0$ will be user specified.

The numerical methods for solving Burger's equation are like those for the full Navier-Stokes equations, but due to the fact that there is only one space dimension and no pressure term, it is possible to accomplish something within a limited amount of computer lab time.

We will consider a finite-difference method on a regular grid $x_j = jh$, with $x_0 = 0$ and $x_{N_J} = \ell$. *Note: there are $N_J + 1$ grid points: $j = 0$ through $j = N_J$.* Let U_j^n denote the numerical solution at grid point j and time step n, i.e. $U_j^n \simeq u(x_j, t_n)$. At each time the numerical solution is a $N_J + 1$ dimensional vector denoted $\mathbf{U}^n = (U_0^n, U_1^n, \ldots, U_j^n, \ldots, U_{N_J}^n)^T$.



The solutions will be time stepped with the Crank-Nicolson/Adams-Bashforth method:

$$\mathbf{U}^{n+1} = (\mathbf{I} - \frac{\nu\triangle t}{2}\mathbf{D}_2)^{-1}\left\{(\mathbf{I} + \frac{\nu\triangle t}{2}\mathbf{D}_2)\mathbf{U}^n + \frac{\triangle t}{2}\left(3\mathbf{N}(\mathbf{U}^n) - \mathbf{N}(\mathbf{U}^{n-1})\right)\right\} \tag{1}$$

where $\mathbf{D_2}$ and $\mathbf{N}(\mathbf{U}) = -\mathbf{U}\mathbf{D_1}\mathbf{U}$ are finite-difference approximations to $\frac{\partial^2}{\partial x^2}$ and $-u\frac{\partial u}{\partial x}$, respectively. $\triangle t$ is the size of time step.

A working program is provided containing all the required high-level code for initialization, I/O, graphics, etc. You are required to write the low-level code to provide the matrices $\mathbf{D_1}$ and $\mathbf{D_2}$ and the numerical linear algebra necessary for implicit time stepping. Such low-level code is the guts of full CFD and dominates the computation time. Hence, efficient implementation, via performance libraries, is essential.

Specific Aims of the Workshop are:

- To construct and apply derivative matrices for first and second derivative operators (using BLAS).
- To program implicit time-stepping (Crank-Nicolson) using banded LU decomposition (via LAPACK).
- To verify the program by performing basic tests of the numerical linear algebra.
- To investigate stability and other numerical issues.
- To guide you towards more advanced topics: pseudo-spectral techniques, multiple dimensions, full CFD using *Semtex*.

A: Initial Setup

Some initial setup is required to install the project code that you will modify to create a working simulation of Burger's equation. Also, a simple X11-based graphics library is provided for real-time graphics of the numerical solutions.

TO DO:

1. First, download and install the graphics library *ezplot* used by the example program to provide real-time graphics. If you do not have directories $HOME/lib and $HOME/include you need to make these directories:

   ```
   prompt> cd $HOME
   prompt> mkdir lib
   prompt> mkdir include
   ```

   Using a web browser, download ezplot.tar.gz from the workshop site to $HOME/lib. By default it will probably be downloaded to $HOME/Downloads (or possibly $HOME/Desktop) and you will need to move it to $HOME/lib. Then untar the compressed tar file:

   ```
   prompt> cd $HOME/lib
   prompt> tar -zxvf ezplot.tar.gz
   ```

   Then make and install the library:

   ```
   prompt> cd ezplot
   prompt> make
   prompt> make install
   ```

2. Install the project code tree. Create a subdirectory for the this case study, e.g.

   ```
   prompt> cd $HOME
   prompt> mkdir cfd
   ```

   Download cfd_example.tar.gz, move to $HOME/cfd. Then untar

   ```
   prompt> cd $HOME/cfd
   prompt> tar -zxvf cfd_example.tar.gz
   ```

   This will create a further 3 subdirectories: Burger, C, Fortran. Burger will contain a full working code for time-stepping Burger's equation. C and Fortran contain fragments for you to work on either in C or Fortran.

   NOTE, the low-level code you will be editing in this workshop is virtually identical whether you are a C, C++ or a Fortran programmer. The C version is C++ code written primarily in procedural C style so as to be easily accessible to both C and C++ programmers and to facilitate a direct translation into Fortran.

3. Before proceeding, you should verify that everything is installed correctly. This also gives you an idea of what your final program should do. Go to the subdirectory Burger with the full example and make and run the finite-difference program.

   ```
   prompt> cd $HOME/cfd/Burger
   prompt> make
   prompt> ./burger
   ```

   For the numerical parameters input

   ```
   1000 200 1000 10000
   ```

   and for the viscosity ν input

   ```
   0.1
   ```

   You should see a window open with the initial condition. With the pointer in this window hit the space bar. The simulation will run until the final time (1000). Pressing the escape key will terminate the simulation. If you type the letter p with the pointer in the window, the simulation will pause until you hit another space.

NOTE: Within the code tree you will find partially completed versions of matrix.cpp *or* matrix.f90 *corresponding to completed Parts B, C, and D of this project, e.g.* matrix_PartB.cpp *and* matrix_PartB.f90 *are versions with Part B of the workshop completed.*

Part B: Derivative Matrices

At the lowest level of almost all CFD, one needs derivative matrices to represent differential operators appearing in the equation. For Burger's equation one needs matrices corresponding to first and second derivatives. The form of these matrices depends on the discretization (finite-difference, spectral, etc). Here we use:

$$\mathbf{D_1} = \frac{1}{2h}\begin{pmatrix} 0 & 0 & & & & & & & \\ -1 & 0 & 1 & & & & & & \\ & \ddots & \ddots & \ddots & & & & & \\ & & -1 & 0 & 1 & & & & \\ & & & -1 & 0 & 1 & & & \\ & & & & -1 & 0 & 1 & & \\ & & & & & \ddots & \ddots & \ddots & \\ & & & & & & -1 & 0 & 1 \\ & & & & & & & 0 & 0 \end{pmatrix}$$

and

$$\mathbf{D_2} = \frac{1}{12h^2}\begin{pmatrix} -42 & 48 & -6 & & & & & & \\ 12 & -24 & 12 & 0 & & & & & \\ -1 & 16 & -30 & 16 & -1 & & & & \\ & \ddots & \ddots & \ddots & \ddots & \ddots & & & \\ & -1 & 16 & -30 & 16 & -1 & & & \\ & & -1 & 16 & -30 & 16 & -1 & & \\ & & & -1 & 16 & -30 & 16 & -1 & \\ & & & & \ddots & \ddots & \ddots & \ddots & \ddots \\ & & & & -1 & 16 & -30 & 16 & -1 \\ & & & & & 0 & 12 & -24 & 12 \\ & & & & & & -6 & 48 & -42 \end{pmatrix}$$

Matrices $\mathbf{D_1}$ and $\mathbf{D_2}$ have constant entries along the diagonals, except near the ends (due to boundary conditions on the PDE). While $\mathbf{D_1}$ is nearly anti-symmetric, and $\mathbf{D_2}$ is nearly symmetric, neither is exactly so due to the approximations at the boundaries. Both matrices are banded: $\mathbf{D_1}$ is tridiagonal and $\mathbf{D_2}$ is penta-diagonal. The bandwidth is closely tied to the order of the approximation. Even though $\mathbf{D_1}$ is tridiagonal, we will treat it as a general banded matrix, since this makes changing the order of the approximation approximation simple if one wishes to do so.

TO DO:
You need to write the following low-level subroutines: Build_D1, Build_D2, Eval_D. These routines will build the (banded) matrices in suitable BLAS storage and perform matrix-vector multiplication, calling suitable BLAS routines

1. Go to the appropriate subdirectory depending on whether you will use C or Fortran. The file matrix.cpp or matrix.f90 contains dummy versions of all the subroutines you need to write with all necessary calling the arguments specified. The top 3 subroutines in this file are the ones you need to complete in this part of the lab. Before doing anything however, you should make the example program as it is:

   ```
   prompt> cd $HOME/cfd/yourcase
   prompt> make
   ```

   The code should compile and run, but it will not actually do anything yet.

   From the comments in these files and the BLAS man pages (see supplemental material), complete the three required subroutines. The hard part is getting the matrix elements into the correct positions for banded matrix storage. You may want to write out on paper first how you will store the above banded matrices matrices in BLAS format.

   Note to C/C++ programmers: I strongly suggest that you use the Fortran interface to BLAS, rather than CBLAS, in this workshop. You will need to use the Fortran interface when you get

to LAPACK in the next parts of the project. Two header files, blas.h and lapack.h, are already in the source tree and are included in matrix.cpp. These files contain prototypes for blas and lapack routines. A _f77 suffix is given to all name of BLAS routines, e.g. the BLAS routine DDOT is called ddot_f77. You should look at the comments at the top of these files.

Make and debug as needed until your code works (or at least compiles).

2. Testing is an important topic that requires more discussion than this lab permits. In the case of finite-difference approximations, we can at least check that Eval_D produces the correct output from specific input and converges to correct order. There is basic testing code already included inside the analysis part of the program (analyze.cpp). What you need to do is the following:

   (a) Edit the file parameter.cpp. Even if you are not familiar with C++, you can find the line near the top of this file containing:
   compute_error = 0;
   Change this to:
   compute_error = 1;
   and make.

   (b) When you now run the code, U will be set to a test function (cosine) and your derivative matrices will be tested against analytic derivatives. You should try the following:

   ```
   prompt> make
   prompt> ./burger
   ```

   For the numerical parameters input

   ```
   100 200 1 1
   ```

   and for the viscosity ν input (the value is irrelevant)

   ```
   0.1
   ```

   You should see a window open with the test function. With the pointer in this window hit the space bar (twice). The window will disappear (because the "simulation" is finished) and in your terminal window you should see output like:

   ```
   WARNING: Setting U in Analyze_ini() for testing
   ```

D1 error:	0.000830144	0.000927878	0.00131616	69
D2 error:	5.25345e-06	2.56785e-05	0.000163807	0

   Your output may not be exactly the same as this, but if you do not see small errors then your matrices are wrong. The last column gives the grid point with the largest error.

   (c) You can try other values of nj, but do not spend too long on this. Once you are finished testing, edit parameter.cpp to put compute_error = 0;

Part C: Time-stepping Operators

Consider now time stepping scheme (1). In order to take a time step you need three subroutines:

- A subroutine that computes $\mathbf{A}_+\mathbf{U}$ from input $\mathbf{U}$, where $\mathbf{A}_+ \equiv (\mathbf{I} + \frac{\nu\triangle t}{2}\mathbf{D_2})$.
- A subroutine that solves $\mathbf{A}_-\mathbf{U} = \mathbf{b}$ for $\mathbf{U}$ from input $\mathbf{b}$, where $\mathbf{A}_- \equiv (\mathbf{I} - \frac{\nu\triangle t}{2}\mathbf{D_2})$.
- A subroutine that computes the nonlinear term $\mathbf{N}(\mathbf{U})$ from input $\mathbf{U}$.

In this part of the project the matrices $\mathbf{A}_+$ and $\mathbf{A}_-$ will be considered. The nonlinear term will be addressed in Part D.

The matrices $\mathbf{A}_+$ and $\mathbf{A}_-$ have the same bandwidth as $\mathbf{D_2}$ and should be constructed from $\mathbf{D_2}$ in such a way that if $\mathbf{D_2}$ is changed (to a higher order approximation for example) no code for the construction of $\mathbf{A}_+$ and $\mathbf{A}_-$ needs changing. Since $\mathbf{A}_-$ is a fixed matrix (for a given $\triangle t$ and ν), it should be factorised once at the beginning (known as a pre-processing step) and the banded-LU decomposition should be used at each time step. *Note, the banded-LU decomposition of* $\mathbf{A}_-$ *requires more storage than* $\mathbf{D_2}$ *or* $\mathbf{A}_+$.

TO DO:
You need to write the following subroutines: Build_A, Eval_A_plus, and Solve_A_minus. The file matrix.cpp or matrix.f90 contains further dummy versions of the subroutines you need to write with all necessary calling arguments specified. From the comments in these files and the BLAS and LAPACK man pages, complete the three required subroutines.

1. The routine Build_A is the most complicated. It will construct $\mathbf{A}_+$ and $\mathbf{A}_-$ from $\mathbf{D_2}$ using appropriate storage and then compute the LU decomposition of $\mathbf{A}_-$. The storage for $\mathbf{A}_-$ will be different from $\mathbf{A}_+$ because of the LU decomposition.

2. The routines Eval_A_plus and Solve_A_minus are relatively easy and require only a few calls to BLAS/LAPACK routines. (There should be no *for* or *do* loops in these subroutines.)

3. Testing. Again, this is a rich subject which we can only touch upon here. The routine Nonlinear should not have been modified yet. In the code provided, this subroutine returns zero for all components of the nonlinear term (equivalent to removing the nonlinear term from the equation). In this form, basic testing of the linear parts of the code can be done.

   (a) A very simple and strong test of the construction $\mathbf{A}_+$ and $\mathbf{A}_-$ and factorization of $\mathbf{A}_-$ is the following. In the time stepping scheme (1), if $\mathbf{N(U)} = 0$ and $\triangle t$ in $\mathbf{A}_-$ is replaced by $-\triangle t$ then (1) becomes:

   $$\mathbf{U}^{n+1} = (\mathbf{I} + \frac{\nu \triangle t}{2}\mathbf{D_2})^{-1}(\mathbf{I} + \frac{\nu \triangle t}{2}\mathbf{D_2})\mathbf{U}^n = \mathbf{U}^n \tag{2}$$

   This means that if you go into your code and (temporarily) change the appropriate sign where you construct $\mathbf{A}_-$ but leave everything else as is, when you run the program, the solution will not change since the $\mathbf{A}_+$ and the modified $\mathbf{A}_-^{-1}$ are (or should be) exact inverses of one another. Try this. If $\mathbf{U}$ changes you have not programmed $\mathbf{A}_+$ and $\mathbf{A}_-$ consistently. After you have finished this test be sure to change back your code so that $\mathbf{A}_-$ is correct.

   (b) A second test is provided by exact solutions. Since with the nonlinear term zero the equation reduces to the heat or diffusion equations, exact solutions are available. You should again edit parameter.cpp to put compute_error = 1; as in Part B. This will invoke testing code not only for $\mathbf{D_1}$ and $\mathbf{D_2}$, but also for time stepping the heat equation. Make and run burger with the following input:

   ```
   1000 200 1000 10000
   ```

   and for the viscosity ν input

   ```
   0.1
   ```

   After the simulation you will find a file error.txt in the working directory. It contains error estimates, in different norms, for the numerical solution compared with the exact solutions at each time step. If everything is working correctly you should see lines such as:

   ```
   10000    4.88247e-08    1.12019e-07    4.18387e-07    1000
   ```

Part D: Nonlinear term

Finally the nonlinear term $\mathbf{N(U)}$ in the equation is addressed.

TO DO:
There is a final subroutine Nonlinear in matrix.cpp or matrix.f90. You need to modify this to compute the nonlinear term.

First note that the example routine Nonlinear returns zero for all components of the nonlinear term. This is always useful for later testing and so you should leave this code as an option (maybe comment it out but don't delete the code).

The computation of the nonlinear term $\mathbf{N(U)}$ requires calling Eval_D. From the result of this the j^{th} component is given by:

$$\mathbf{N(U)}|_j = \mathbf{U}_j\,(\mathbf{D_1 U})_j.$$

where $(\mathbf{D_1 U})_j$ is the j^{th} component of $\mathbf{D_1 U}$. This nonlinear product between $\mathbf{U}$ and $\mathbf{D_1 U}$ should be carried out with a simple *for* or *do* loop. Even though it can actually be accomplished with a call to a BLAS routine (can you find which one), this is not natural - the term is nonlinear and BLAS addresses linear computations.

Part E: Example Runs

Now that you have a working code for simulating Burger's equations, let's look at a few examples. First make sure that have put compute_error = 0; in parameter.cpp. Now make and run burger with the following input:

1000 200 1000 10000

and for the viscosity ν

0.1

With the initial condition already in the code you should see shocks form with the largest shock to the left which the propagates to the right and overtakes all others.

Now run exactly as before but change the viscosity to 1.

Now run exactly as before but change the viscosity to 0.01. In this case the numerical solution will be unstable due to a violation of the CFL condition. This can be fixed by upwind differencing.

The way the code is written, the initial conditions are set in driver.cpp. There are a few other cases there. You might want to give these a try (by editing driver.cpp and re-making). Try running with the standard values:

1000 200 1000 10000

and for the viscosity ν

0.1

Part F: Basic Modifications and Extensions

Here are recommendations for further work that can be done later, in your own time, if you are interested.

- The numerical instability illustrated in Part E can be cured with *upwind differencing*. Modify the subroutine Nonlinear to perform upwind differencing of the first derivative rather than calling Eval_D.
- Modify your existing program to use Fourier pseudo-spectral method on the interval $0 < x < \ell$ subject to periodic boundary conditions. There is an example of such a code in the directory Burger. You should be able to do this only changing the stepper code. The driver and analyzer should be unaffected. The fft library is not provided and it is up to you to find one and to call the ffts appropriately.
- Write a program, which could later be turned into a subroutine, to solve the 2D Helmholtz/Poisson equation. You could either do this in finite differences or pseudo-spectrally.
- Download *Semtex* from the web site: *http://users.monash.edu.au/~bburn/semtex.html*. Install it, read the manual pages and run some of the examples. *Semtex* is a free, state-of-the-art spectral-element CFD code. It uses a splitting scheme that is similar to, though more advanced than, the scheme we have used for Burger's equation in this workshop.

Appendix: Finite-difference formulas

For the first derivatives, we use the second-order approximation to $\frac{\partial u}{\partial x}$:

$$\frac{\partial u}{\partial x}(x_j) = \frac{-U_{j-1} + U_{j+1}}{2h} + O(h^2).$$

This can be verified by writing U_{j+1} in terms of Taylor series about point x_j. Since the boundary conditions are $\frac{\partial u}{\partial x} = 0$ at $x = 0$ and $x = \ell$ we have:

$$\frac{\partial u}{\partial x}(x_0) = \frac{\partial u}{\partial x}(x_{N_J}) = 0.$$

For points $1 < j < N_J - 1$, the following is a fourth-order approximation to $\frac{\partial^2 u}{\partial x^2}$:

$$\frac{\partial^2 u}{\partial x^2}(x_j) = \frac{-U_{j-2} + 16U_{j-1} - 30U_j + 16U_{j+1} - U_{j+2}}{12h^2} + O(h^4).$$

This can be verified by writing U_{j+1} in terms of Taylor series about point x_j.

At the boundary grid points $j = 0, 1$ and $j = N_J - 1, N_J$, it is not possible to maintain the fourth-order approximation without increasing the bandwidth of the matrix. Rather than do this, we use a lower-order approximation near the boundaries:

$$\frac{\partial^2 u}{\partial x^2}(x_1) = \frac{U_0 - 2U_1 + U_2}{h^2} + O(h^2),$$

$$\frac{\partial^2 u}{\partial x^2}(x_0) = \frac{-7U_0 + 8U_1 - U_2}{2h^2} + O(h^2),$$

and similarly at grid points $N_J - 1$ and N_J. The approximation at x_0 relies on the assumed boundary condition $\frac{\partial u}{\partial x}(x) = 0$.

SUPPLEMENTAL MATERIAL FOR CFD WORKSHOP:

Summary of banded storage in BLAS and LAPACK

The following information is available from the BLAS and LAPACK documentation as well as various web sources. To assist with the workshop the following diagrams may help. Given a banded matrix A

$$A = \begin{pmatrix} 11 & 12 & 13 & & & & & & \\ 21 & 22 & 23 & 24 & & & & & \\ 31 & 32 & 33 & 34 & 35 & & & & \\ & 42 & 43 & 44 & 45 & 46 & & & \\ & & 53 & 54 & 55 & 56 & 57 & & \\ & & & 64 & 65 & 66 & 67 & 68 & \\ & & & & 75 & 76 & 77 & 78 & 79 \\ & & & & & 86 & 87 & 88 & 89 \\ & & & & & & 97 & 98 & 99 \end{pmatrix}$$

where blanks indicate zero values, 11 means a_{11} etc, then the BLAS format for the banded storage of A is:

$$A_{blas} = \begin{pmatrix} * & * & 13 & 24 & 35 & 46 & 57 & 68 & 79 \\ * & 12 & 23 & 34 & 45 & 56 & 67 & 78 & 89 \\ 11 & 22 & 33 & 44 & 55 & 66 & 77 & 88 & 99 \\ 21 & 32 & 43 & 54 & 65 & 76 & 87 & 98 & * \\ 31 & 42 & 53 & 64 & 75 & 86 & 97 & * & * \end{pmatrix}$$

Elements indicated with $*$ are never accessed and need not be set. In this case LDA = 5, KL = 2, LU = 2. Banded storage can be thought of as storing diagonals along rows (note the location of the diagonal of A when stored in A_{blas}) or as up-shifting the rows of the matrix. Note that the column locations remain unaltered.

The LU decomposition of a banded matrix requires and additional KL rows. So the above matrix would be store as (before factorization)

$$A_{lapack} = \begin{pmatrix} * & * & * & * & + & + & + & + & + \\ * & * & * & + & + & + & + & + & + \\ * & * & 13 & 24 & 35 & 46 & 57 & 68 & 79 \\ * & 12 & 23 & 34 & 45 & 56 & 67 & 78 & 89 \\ 11 & 22 & 33 & 44 & 55 & 66 & 77 & 88 & 99 \\ 21 & 32 & 43 & 54 & 65 & 76 & 87 & 98 & * \\ 31 & 42 & 53 & 64 & 75 & 86 & 97 & * & * \end{pmatrix}$$

Elements indicated with $*$ are never accessed and need not be set. Elements indicated with $+$ are not set, but are filled in by the LU decomposition and are used in later back substitutions.

NAME

DGBMV - perform one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A'*x + beta*y,

SYNOPSIS

```
SUBROUTINE DGBMV ( TRANS, M, N, KL, KU, ALPHA, A, LDA, X, INCX, BETA,
                   Y, INCY )

    DOUBLE        PRECISION ALPHA, BETA

    INTEGER       INCX, INCY, KL, KU, LDA, M, N

    CHARACTER*1   TRANS

    DOUBLE        PRECISION A( LDA, * ), X( * ), Y( * )
```

PURPOSE

DGBMV performs one of the matrix-vector operations

where alpha and beta are scalars, x and y are vectors and A is an m by n band matrix, with kl sub-diagonals and ku super-diagonals.

PARAMETERS

TRANS - CHARACTER*1.
On entry, TRANS specifies the operation to be performed as follows:

TRANS = 'N' or 'n' y := alpha*A*x + beta*y.

TRANS = 'T' or 't' y := alpha*A'*x + beta*y.

TRANS = 'C' or 'c' y := alpha*A'*x + beta*y.

Unchanged on exit.

M - INTEGER.
On entry, M specifies the number of rows of the matrix A. M must be at least zero. Unchanged on exit.

N - INTEGER.
On entry, N specifies the number of columns of the matrix A. N must be at least zero. Unchanged on exit.

KL - INTEGER.
On entry, KL specifies the number of sub-diagonals of the matrix A. KL must satisfy 0 .le. KL. Unchanged on exit.

KU - INTEGER.
On entry, KU specifies the number of super-diagonals of the matrix A. KU must satisfy 0 .le. KU. Unchanged on exit.

ALPHA - DOUBLE PRECISION.
On entry, ALPHA specifies the scalar alpha. Unchanged on exit.

A - DOUBLE PRECISION array of DIMENSION (LDA, n).
Before entry, the leading (kl + ku + 1) by n part of the array A must contain the matrix of coefficients, supplied column by column, with the leading diagonal of the matrix in row (ku + 1) of the array, the first super-diagonal starting at position 2 in row ku, the first sub-diagonal starting at position 1 in row (ku + 2), and so on. Elements in the array A that do not correspond to elements in the band matrix (such as the top left ku by ku triangle) are not referenced. The following program segment will transfer a band matrix from conventional full matrix storage to band storage:

```
DO 20, J = 1, N K = KU + 1 - J DO 10, I = MAX( 1, J - KU ), MIN(
M, J + KL ) A( K + I, J ) = matrix( I, J ) 10     CONTINUE 20
CONTINUE
```

Unchanged on exit.

LDA - INTEGER.
On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. LDA must be at least (kl + ku + 1). Unchanged on exit.

X - DOUBLE PRECISION array of DIMENSION at least
(1 + (n - 1)*abs(INCX)) when TRANS = 'N' or 'n' and at least (1 + (m - 1)*abs(INCX)) otherwise. Before entry, the incremented array X must contain the vector x. Unchanged on exit.

INCX - INTEGER.
On entry, INCX specifies the increment for the elements of X. INCX must not be zero. Unchanged on exit.

BETA - DOUBLE PRECISION.
On entry, BETA specifies the scalar beta. When BETA is supplied as zero then Y need not be set on input. Unchanged on exit.

Y - DOUBLE PRECISION array of DIMENSION at least
(1 + (m - 1)*abs(INCY)) when TRANS = 'N' or 'n' and at least (1 + (n - 1)*abs(INCY)) otherwise. Before entry, the incremented array Y must contain the vector y. On exit, Y is overwritten by the updated vector y.

INCY - INTEGER
On entry, INCY specifies the increment for the elements of Y. INCY must not be zero. Unchanged on exit.

Level 2 Blas routine.

-- Written on 22-October-1986. Jack Dongarra, Argonne National Lab. Jeremy Du Croz, Nag Central Office. Sven Hammarling, Nag Central Office. Richard Hanson, Sandia National Labs.

NAME

DGBTRF - compute an LU factorization of a real m-by-n band matrix A using partial pivoting with row interchanges

SYNOPSIS

```
SUBROUTINE DGBTRF( M, N, KL, KU, AB, LDAB, IPIV, INFO )

    INTEGER         INFO, KL, KU, LDAB, M, N

    INTEGER         IPIV( * )

    DOUBLE          PRECISION AB( LDAB, * )
```

PURPOSE

DGBTRF computes an LU factorization of a real m-by-n band matrix A using partial pivoting with row interchanges. This is the blocked version of the algorithm, calling Level 3 BLAS.

ARGUMENTS

M (input) INTEGER
The number of rows of the matrix A. M >= 0.

N (input) INTEGER
The number of columns of the matrix A. N >= 0.

KL (input) INTEGER
The number of subdiagonals within the band of A. KL >= 0.

KU (input) INTEGER
The number of superdiagonals within the band of A. KU >= 0.

AB (input/output) DOUBLE PRECISION array, dimension (LDAB,N)
On entry, the matrix A in band storage, in rows KL+1 to 2*KL+KU+1; rows 1 to KL of the array need not be set. The j-th column of A is stored in the j-th column of the array AB as follows:
AB(kl+ku+1+i-j,j) = A(i,j) for max(1,j-ku)<=i<=min(m,j+kl)

On exit, details of the factorization: U is stored as an upper triangular band matrix with KL+KU superdiagonals in rows 1 to KL+KU+1, and the multipliers used during the factorization are stored in rows KL+KU+2 to 2*KL+KU+1. See below for further details.

LDAB (input) INTEGER
The leading dimension of the array AB. LDAB >= 2*KL+KU+1.

IPIV (output) INTEGER array, dimension (min(M,N))
The pivot indices; for 1 <= i <= min(M,N), row i of the matrix was interchanged with row IPIV(i).

INFO (output) INTEGER
= 0: successful exit
< 0: if INFO = -i, the i-th argument had an illegal value
> 0: if INFO = +i, U(i,i) is exactly zero. The factorization has been completed, but the factor U is exactly singular, and division by zero will occur if it is used to solve a system of equations.

FURTHER DETAILS

The band storage scheme is illustrated by the following example, when M = N = 6, KL = 2, KU = 1:

On entry: On exit:

*	*	*	+	+	+	*	*	*	u14	u25	u36
*	*	+	+	+	+	*	*	u13	u24	u35	u46
*	a12	a23	a34	a45	a56	*	u12	u23	u34	u45	u56
a11	a22	a33	a44	a55	a66	u11	u22	u33	u44	u55	u66
a21	a32	a43	a54	a65	*	m21	m32	m43	m54	m65	*
a31	a42	a53	a64	*	*	m31	m42	m53	m64	*	*

Array elements marked * are not used by the routine; elements marked + need not be set on entry, but are required by the routine to store elements of U because of fill-in resulting from the row interchanges.

NAME

DGBTRS - solve a system of linear equations A * X = B or A' * X = B with a general band matrix A using the LU factorization computed by DGBTRF

SYNOPSIS

```
SUBROUTINE DGBTRS( TRANS, N, KL, KU, NRHS, AB, LDAB, IPIV, B, LDB, INFO
                   )

    CHARACTER      TRANS

    INTEGER        INFO, KL, KU, LDAB, LDB, N, NRHS

    INTEGER        IPIV( * )

    DOUBLE         PRECISION AB( LDAB, * ), B( LDB, * )
```

PURPOSE

DGBTRS solves a system of linear equations A * X = B or A' * X = B with a general band matrix A using the LU factorization computed by DGBTRF.

ARGUMENTS

TRANS (input) CHARACTER*1
Specifies the form of the system of equations.
'N': A * X = B (No transpose)
'T': A'* X = B (Transpose)
'C': A'* X = B (Conjugate transpose = Transpose)

N (input) INTEGER
The order of the matrix A. N >= 0.

KL (input) INTEGER
The number of subdiagonals within the band of A. KL >= 0.

KU (input) INTEGER
The number of superdiagonals within the band of A. KU >= 0.

NRHS (input) INTEGER
The number of right hand sides, i.e., the number of columns of the matrix B. NRHS >= 0.

AB (input) DOUBLE PRECISION array, dimension (LDAB,N)
Details of the LU factorization of the band matrix A, as computed by DGBTRF. U is stored as an upper triangular band matrix with KL+KU superdiagonals in rows 1 to KL+KU+1, and the multipliers used during the factorization are stored in rows KL+KU+2 to 2*KL+KU+1.

LDAB (input) INTEGER
The leading dimension of the array AB. LDAB >= 2*KL+KU+1.

IPIV (input) INTEGER array, dimension (N)
The pivot indices; for 1 <= i <= N, row i of the matrix was interchanged with row IPIV(i).

B (input/output) DOUBLE PRECISION array, dimension (LDB,NRHS)
On entry, the right hand side matrix B. On exit, the solution matrix X.

LDB (input) INTEGER
The leading dimension of the array B. LDB >= max(1,N).

INFO (output) INTEGER
= 0: successful exit
< 0: if INFO = -i, the i-th argument had an illegal value