Do Constructs & Named Constructs

- So far we have mainly used the **counter-controlled** do
  
  ```
  do k = 1, n
  ```

- We have used **nested loops**.

- We have also used the **do-exit** construct.

  ```
  do
    statements
    if( · · · ) EXIT
  end do
  ```

- Here the **EXIT** statement causes repetition of the loop to terminate by transferring control to the statement following the `end do`.

- Sometimes it is necessary to terminate only the current repetition and jump ahead to the next one. Fortran 90 provides the **CYCLE** statement to accomplish this.
• Suppose we were writing a code to calculate the load on a column given some value of the parameter $s$. We don’t read in a fixed number of values of $s$ (which must be between 0 and 200) but rather query the user to see if there is any more data; if the response is “no”, then we terminate loop using the **exit**. However, if the user enters data outside the valid range for $s$ then we use the **cycle** command so we don’t calculate a load for these values.

```fortran
if ( s < 0.0 .or. s > 200.0) then
   print *, "invalid value for s"
   cycle
end if
```

Then no load will be calculated for this value and the control goes back to the start of the **do** loop. Of course this can be accomplished in other ways.

• Care must be used for non-counter controlled loops because it’s easy to get into an infinite loop.

• Another type of **do** construct is the **do while** which continues the loop as long as a specified condition is satisfied. For example
do while ( error >= tolerance )
    statements
end do

The specified condition uses the syntax of an if construct.

• Once again care must be taken because you can get into an infinite loop.

• Oftentimes we can have several nested loops. For ease of reading and debugging we may want to label the loops and the end do. This can be done with comment statements or with a named loop.

• The syntax is

    name: DO · · ·
    statements
    end do name

• When we have named nested loops, then we can exit from a particular loop; for example,
x_loop: do
  print *, "enter x"
  read *, x

  n_loop: do
    print *, "enter n"
    read *, n
    if (n == 0) exit n_loop
    if (x < 0.0 .and. n < 0) exit x_loop
    statements
  end do n_loop
end do x_loop

• We can also have named conditionals, including the case construct. The syntax is analogous. For example,

  error_check: if (error < tolerance) then
    statements
  end if error_check
grade:  case select (grade)
         case ('A')
            statements
         case ('B')
            statements
end select grade
Current time and date

- First of all, Fortran 90 has an intrinsic routine called

\[
\text{date\_and\_time(date, time, zone, values)}
\]

which returns a date stamp. All four arguments are optional. It determines the date in the format CCYYMMDD (century, year, month, day), the time in hhmmss.ss (hours, minutes, seconds); the last two arguments have to do with Greenwich Mean time.

- This is useful because if your code generates output, you can put a date stamp on your output files. This is good for archival reasons.

- For example, we could have

\[
\text{call date\_and\_time (date, time )}
\]
\[
\text{print *, date; print *, time}
\]
where we have declared
character (len=8) date
class (len=10) time

• The output is
  20100323 → March 23, 2010
  124211.034 → 12:42 and 11.034 seconds

• Of course a better way to output this is to write your own routine called, e.g., `timestamp` which calls this intrinsic routine but prints out the result in a nicer way. If you need to know if the time is a.m. or p.m. this can be obtained using the other variables.

• The variables must be declared as a character string at least the length indicated above; they can be declared longer also.

• Output for your remaining homework should include a time stamp.
Timing a Job

- To compare two approaches for solving the same problem one often compares the time it takes for each method.

- In other situations you might want to determine which part of your code is creating the long execute time so you can try to make that section more efficient.

- For these reasonings one often wants to time a job or a section of a job.

- There are two options for timing. One gives “real” time, i.e., the wall clock timing. Of course this can vary slightly from run to run.

- The other option is to determine the actual CPU time. Fortran 90 does not have an intrinsic procedure for measuring CPU time but a utility function called `etime` can be used but it can possibly cause a loss in portability of your code.

- Fortran 90 provides the intrinsic `system_clock` which returns elapsed wall-clock time (not CPU).
- There are three arguments of `system_clock` which are all optional.
  - `count`, `intent(out)` an integer
  - `count_rate`, `intent(out)` an integer
  - `count_max`, `intent(out)` an integer
- Consider the following call to `system_clock` with write statements
  ```fortran
  call system_clock ( clock_count1, clock_rate, clock_max )
  
  write ( *, '(a)' )' SYSTEM_CLOCK with INTEGER arguments
  write ( *, '(a,i12)' )' The current clock count is ', clock_count1
  write ( *, '(a,i12)' )' The clock count per second is ', clock_rate
  write ( *, '(a,i12)' )' The maximum clock count is ', clock_max
  
  The output is:
SYSTEM_CLOCK with INTEGER arguments reports:
The current clock count is 202523129
The clock count per second is 1000
The maximum clock count is 2147483647

- The clock rate says that the minimum measurable time in seconds is 0.001.
- The maximum measurable time is given by \( \text{clock\_max} \).
- Now consider the following section of code which computes the amount of time to determine random numbers.

\[
\begin{align*}
call \ \text{system\_clock} & ( \ \text{clock\_count1}, \ \text{clock\_rate}, \ \text{clock\_max} ) \\
do \ i = 1, \ n \\
& \quad \text{call \ random\_number} ( \ x ) \\
end \ do \\
call \ \text{system\_clock} & ( \ \text{clock\_count2}, \ \text{clock\_rate}, \ \text{clock\_max} ) \\
write ( \ *, \ '(a)' ) & ' ' \\
write ( \ *, \ '(a, g14.6, a)' ) & \&
' Estimated time to compute random numbers is = ', &
real ( clock_count2 - clock_count1, kind = 8 ) &
/ real ( call_num * clock_rate, kind = 8 ), ' seconds'

– The first call to system_clock initializes the count which we save as
 clock_count1; after we have computed a set number of random variables
we call it again to compute the final time which we store in clock_count2.
To get the total time, we just subtract. The timing for this routine using
n=1,000,000 is still negligible.

– The basic idea is just put a call before the section of code you want to time
using one variable for the clock and then another call after the section
using a different variable; then we just subtract the two.

– One warning is that input/output can really effect your timings so it’s
better not to include those.

– If you want to compute CPU time then you can use a gnu utility called
 etime; see the gnu website for information on this utility.
Using Standard Library Routine

- Sometimes you don’t have the time (or inclination) to write a routine to perform a basic calculation. For example, suppose you want to solve a full $n \times n$ linear system of equations $A\vec{x} = \vec{b}$ using Gaussian elimination but you don’t have time to write your own code (although everyone should do it once).

- In this case you can use a standard library routine.

- There are standard linear algebra packages and a good source for finding these in the netlib repository which is managed by University of Tennessee, Knoxville and Oak Ridge National Laboratory (ORNL).

- You can search or browse to find what you want and then download the source codes.

- Today I’m going to take you through the steps to find a code for solving $A\vec{x} = \vec{b}$ using Gaussian elimination from the standard linear algebra package lapack. You will see that with what you already know, it’s quite easy. Also
we will use the Fortran 77 versions to see that this doesn’t matter.

- Basically, we will download the necessary files, then I like to concatenate them because there can be a lot of them due to dependencies, compile them to create a .o file, then use this as we do with modules on the command line `gfortran`.

- The only tricky part can be making sure your statement to call the library routine is exactly what it needs to be. Also sometimes the routine may need work arrays which you need to allocate.

- You should be aware that even if you use a Fortran 90 version of a Lapack routine, it probably is just a straightforward change in syntax, i.e., no modules, OOP, etc. are used. Because linear algebra routines have been well known for a long time, the methods were originally implemented in Fortran 77.
Solving a linear algebraic system of equations using Gauss elimination

Suppose we have $n$ linear equations in $n$ variables $x_i, \ i = 1, n$ where each equation is of the form

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n = b_i$$

This system can be written in matrix form $A\vec{x} = \vec{b}$ where

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \quad \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

and

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1,n} \\ a_{21} & a_{22} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{n,n} \end{pmatrix}$$

- Typically your code creates the entries of $A$ and $\vec{b}$ or alternately you could input them.
• When you took algebra, you saw a technique for solving linear systems of equations where you eliminate one variable ($x_1$) from the second equation, two variables from the third ($x_1, x_2$), etc. until the last equation has only one variable, $x_n$. You solve for $x_n$ and then use that value to calculate $x_{n-1}$, etc. This method is called Gauss elimination.

• We won’t go into the way the algorithm works (or the similar technique of $LU$ decomposition), we’ll just trust that the person who wrote the code implemented it correctly. In this way, we will use the code as a black box - not something one likes to do though.

• We now go to netlib and look for the lapack linear algebra package.

• If you click on individual routines then we can see the naming convention that is used.

• The first letter indicates the data type
  S - real, D - double precision, C - complex, etc.

• The second two letters indicate the type of matrix
  GE - general, GB - general banded, SB - symmetric banded, etc.
• The last three letters indicate the computation performed
• We want to use single precision, and solve a general linear system, i.e., the matrix has no special properties.
• Now we click on Index of LAPACK single precision real routines
• We see that sgesv.f is the final we want. It says it solves $AX = B$. Now let’s look at the individual routine and the comment statements should tell us about the routine and its calling statement.
• We see that it has 8 arguments
  
  \[
  \text{SUBROUTINE SGESV( N, NRHS, A, LDA, IPIV, B, LDB, INFO )}
  \]
  
  We have to read the comment statements to figure out what each of these stands for.
• $N$ is the number of equations, i.e., the size of the $n \times n$ matrix.
• $\text{NRHS}$ is the number of right hand sides; the code is general so it allows you to solve $A\vec{x} = \vec{b}$ for several different vectors $\vec{b}$ but the same $A$ if you know them all at once. For us we only have one rhs.
• $A$ is the $N \times N$ matrix
• **LDA** is the first dimension in your dimension of $A$; typically it is $N$ but in Fortran 77 it was often some number greater than $N$.

• **IPIV** is an integer work array which you must set up.

• **$B$** is your rhs on input and on output it is your solution.

• **LDB** is the first dimension of $B$; remember that the code allows $B$ to be a two dimensional array holding several right hand sides; typically this is also $N$.

• **INFO** is an integer that is output by the routine (you have to print it) and if it is 0, then normal termination; otherwise it gives you an indication where the code failed (if you understand how the algorithm works).

**Dependencies**

• Lapack was written by building the algorithms on a set of standard basic linear algebra subprograms called **blas**. These include three levels of routines such as matrix times vector, norm calculations, etc. Remember that in fortran 77 there was no intrinsic for a matrix times a vector or the product of two
• Consequently, when we download our program \texttt{sgesv.f} we see that it calls other routines. These are called dependencies in the netlib repository.

• If we go back a page we can see that we can download the file plus dependencies.

• Note that we can choose to download it into a zip or tar file. It also has the warning that some files may contain additional dependencies on BLAS routines.

• I tend to download the blas routines once and compile them. That way they are always available.

• We now ask to download the files; it created a folder called \texttt{single}.

• If you open this folder it contains five files. We could compile each separately or we could combine them in one file. This is probably the best because sometimes we have twenty or thirty files downloaded.

• In UNIX you can use the concatenate command to create one file. Since the files are the only ones in that folder, I move to that folder and then use
cat *.f > solver.f
This takes all files that end in .f and combines them into a single file solver.f. Note that I used the .f extension because these are fortran 77 files.

• If you list elements in that directory, you will see that the
• Now I simply compile solver.f.
gfortran -c solver.f

• To let my program have access to this file I just include the file solver.o in the gfortran statement as we have done in the past.

Using the library routine

• Now we will write a main program to call this routine.

• We will input a random matrix; then we choose our solution (such as all one's) and generate the rhs by using the intrinsic routine for matrix multiplication, matmul.

• Now that we have the matrix and the rhs, we have a call to the solver and print out the answer.
• My call to the solver is just
  
  call sgesv ( n, nrhs, a, n, ipivot, rhs, n, info )
  
  where I have set the number of rhs to one, dimensioned the integer work array ipivot and the integer info along with the variables I needed to set up the matrix and rhs.

• Oftentimes the library routines will not generate any print statements; it is up to you to print out info (recall this indicates if we had normal execution) and whatever you want to do with the solution of your system.

• Typically I include statements like

  if ( info .ne. 0 ) write *, "solver failed, info = ", info

• Now when we run this we see that it fails. We get a list of undefined symbols:
  
  "lsame" referenced from sgetrs
  "xerbla" referenced from sgesv

• If you recall, there is a disclaimer on the netlib site that says that routines may also use standard BLAS routines. This is what is happening.

• Consequently, what I do is just download all the BLAS routines once and
create a `blas.o` file which I use on the command line (or in my makefile).

- Now when we run it we still get some undefined symbols. There are a couple of routines which have to do with timing that weren’t included in my download of BLAS. So I just go back to netlib and do a search for the routine I need. I just cut and past it to the end of my file `blas.f`, recompile it and try again until I have all undefined symbols removed.

- Now when I use the command

  ```bash
  gfortran test_sgesv solver.o blas.o
  ```

  we get the following output

  solution to linear system is
  
  1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000