Fortran 90+ Intensive Workshop

Dr Stephen So
(s.so@griffith.edu.au)

Griffith School of Engineering

Brief Outline

- Fortran versus MATLAB / C
- Fortran program development and syntax
- Intrinsic data types and operators
- Arrays and array manipulation
- File reading and writing
- Subprograms – functions and subroutines
- Modules
- Derived data types
- Function and operating overloading
- gfortran compiler optimisation
- Calling LAPACK and FFTPACK
- Calling Fortran 90 subprograms from MATLAB

Introduction

- Essential elements of the Fortran 90+ (90/95/2003) programming language will be covered
- Prior programming experience (e.g. MATLAB) is assumed
- Working in the Windows environment
- See http://maxwell.me.gu.edu.au/sso/fortran

What do I need in order to write Fortran programs?

- Fortran compiler
  - converts your source coded into an executable problem
- Text editor or integrated development environment (IDE)
  - used to write your source code into a text file
- Some numerical libraries (LAPACK, BLAS, FFTPACK, RKSUITE, etc.)
  - contains subprograms written (and tested) by others

Fortran compilers

- There are many commercial Fortran compilers on the market (cost a lot of $$$)
  - Intel Visual Fortran
  - PGI Visual Fortran
  - NAG Fortran
  - Lahey Fortran
  - Absoft Fortran
  - Compaq Visual Fortran (discontinued, replaced by Intel VF)
- We will use a free and open-source compiler
  - GNU Fortran (gfortran)

IDEs

- Commercial IDEs
  - Microsoft Visual Studio (used by Intel VF, PGI VF, Lahey, Compaq VF)
  - Custom IDEs (used by Absoft and older version of Lahey Fortran 95)
- Free and open-source IDEs
  - CodeBlocks
  - Eclipse (Photran plugin)
  - NetBeans

Numerical Libraries

- Commercial maths libraries
  - Intel Math Kernel Library (MKL)
  - International Mathematics and Statistics Library (IMSL)
- Free maths libraries
  - AMD Core Math Library (ACML) (no source code)
  - BLAS/ATLAS, LAPACK, FFTPACK, etc. from netlib.org
  - FFTW

Fortran versus MATLAB

- IBM Mathematical Formula Translation System (Fortran II, III, IV, 66, 77, 90, 95, 2003, 2008)
- Advantages:
  - Very fast (compiled versus interpreted)
  - Efficient with memory (can deallocate arrays)
  - More control over data representation and formatting
  - Very popular in scientific and engineering community
  - Lots of legacy Fortran 77 code available (BLAS, LAPACK, etc.)

Fortran versus MATLAB

- Disadvantages:
  - Tedious and difficult to learn
  - More verbose
  - Very limited matrix manipulation
  - No built-in routines or toolboxes
  - No built-in visualisation tools

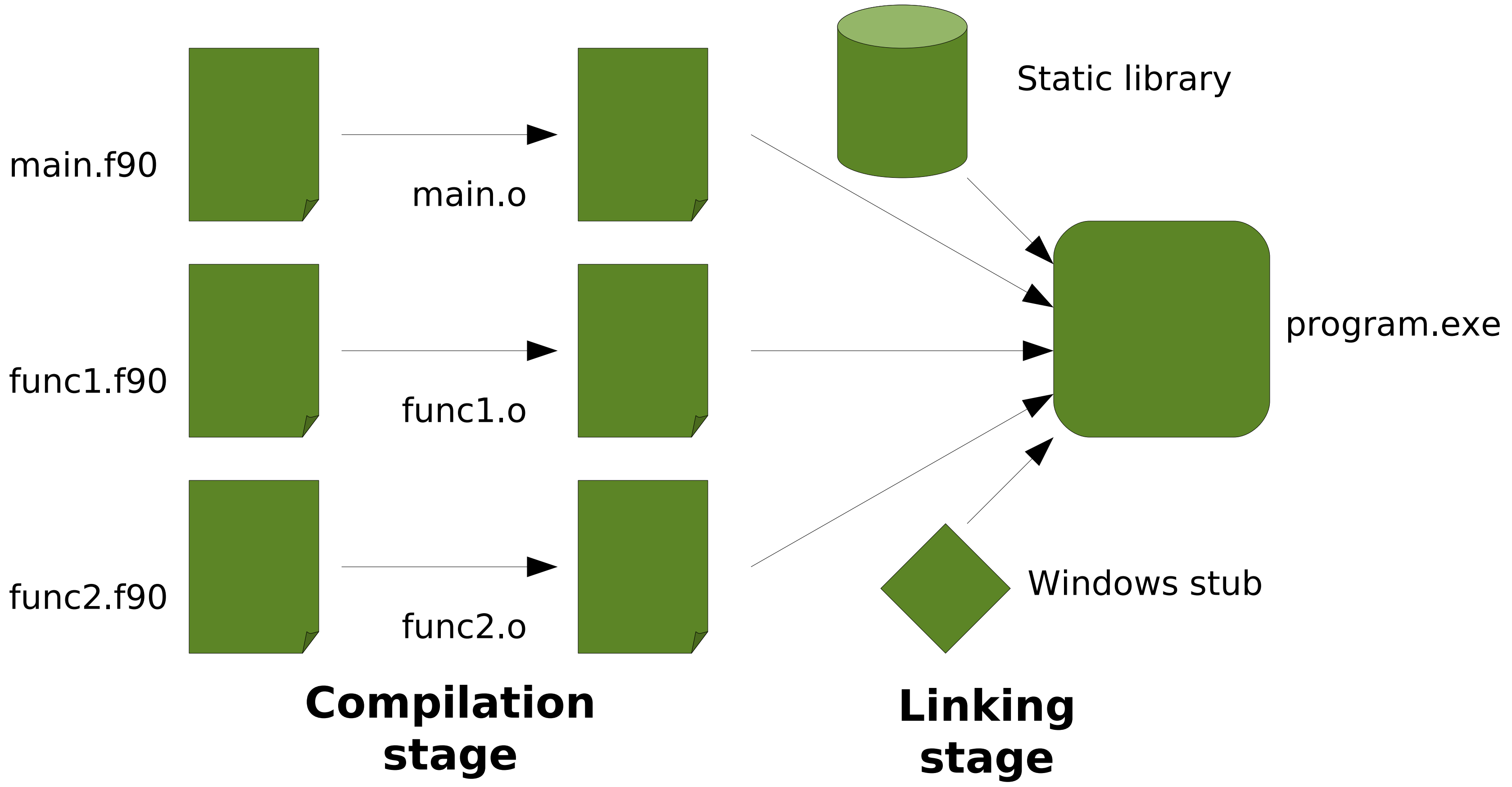
Fortran versus C

- C strengths are in systems programming
- Fortran's strengths are in number crunching and numerical data processing
- Fortran compilers are more sophisticated than C compilers for numerical optimisation
- Modern Fortran compilers have support for parallel processing and GPU processing (CUDA) as well

Compiling and linking

- Fortran 90 source code is typed into a text file (*.f90, *.95)
- To run the Fortran program:
  - Step 1: Compile source code => object file (*.o)
  - Step 2: Link in other object files (other subprograms) and stub => executable file (*.exe)

Compiling and linking



Using gfortran (from command-line)

- In a command line, to compile and link a single Fortran source file (hello.f90)
  - `gfortran hello.f90 -o hello.exe`
- To perform compiling only (no linking)
  - `gfortran hello.f90 -c`
- To link multiple object files
  - `gfortran hello.o func1.o func2.o -o hello.exe`
- To include static libraries
  - `gfortran hello.o liblapack.a libblas.a -o hello.exe`

Fortran program structure

```
program program_name
    specification statements
    executable statements
[contains]
    [internal subprograms]
end program program_name
```

- Note that [ ] means this section is optional

Simple Fortran program

```fortran
program hello
    implicit none
    ! This is a comment
    print *, 'Hello, world'
end program hello
```

This statement should always be included

- Save this into a text file called hello.f90
- Compile in command line
  - gfortran hello.f90 -o hello.exe

Comments and continuation

- Any line of code after ! is treated as a comment
- A statement can be broken into multiple lines by appending a & (in MATLAB it is ...)

```
print *, 'Length of the two sides are', &
         side_1, 'and', side_2, &
        'metres'
```

- Multiple statements can be put onto a single line by separating them with semi-colons (;)

```
side_1 = 3; side_2 = 4; hypot = 10
```

Printing to the screen

- We can see the **print** statement allows us to display text on the screen
- Syntax:
  - **`print`** *`fmt, list`*
- *fmt* specifies how the text is to be formatted (more about this later)
- If *fmt* is an asterisk (*), formatting is automatic (list-directed I/O)
- *list* can consist of strings (text) or variables

Reading from keyboard

- To read data from the keyboard, we use the **read** statement
- Syntax:
  - **read** *fmt*, *list*
- If *fmt* is an asterisk (*), the format of input data is automatic
- *list* consists of variables to store the data

Variables

- Like MATLAB, variables are memory locations used to store data
- Unlike MATLAB:
  - we must declare **every variable** that we use in the program (unlike MATLAB)
  - Variable names are not case-sensitive (e.g. pi, pI, Pi, PI are the same name)
- Syntax for declaring variables
  - `data_type [, specifiers ::] variable_name`

Examples of variable declarations

- **integer** :: counter, i, j
- **real** :: mass
- **real**, **parameter** :: pi = 3.1416
- **logical** :: flag = **.false.**
- **complex** :: impedance
- **character**(20) :: name

Intrinsic numeric types

- **Integer** (4 bytes / 32 bits)
  - A whole number that has no fractional part or decimal point e.g. 3, -44, 120
  - Largest: 2147483647
- **Real** (4 bytes / 32 bits)
  - A number that can have a fractional part or decimal point e.g. 3.0, 2.34, -98.12, 3.3e8
  - IEEE 754 single precision: 6 decimal digits
  - Smallest, Largest: 1.17549435E-38, 3.40282347E+38

Intrinsic numeric types

- **Double precision** (8 bytes / 64 bits)
  - A real number with twice the precision e.g. 3.3d8
  - IEEE 754 double precision: 15 decimal digits
  - Smallest: 2.2250738585072013 8E-308
  - Largest: 1.7976931348623157 1E+308
- **Complex**
  - Complex number consisting of a real part and imaginary part
  - e.g. 3.2 – 4.98 i => (3.2, 4.98)

Intrinsic non-numeric types

- **Character**
  - ASCII character or text e.g. 'a', 't', '$'
  - character(*N*) to declare string of *N* chars
  - Can use either “ or ' to delimit strings
  - // for concatenation   e.g. 'abc'//'def'
  - **len**() function returns the number of characters
- **Logical**
  - Either **.true.** or **.false.**

Kind parameter

- The range of an integer (max 2147483647) may not be enough
- The precision of double precision (15) may not be enough
- Fortran 90 allows us to specify the kind of integer and the kind of real
- The kind number is *usually* (but not always) the number of bytes
  - **integer**(**kind** = 8) or **integer**(8) – 8 byte integer
  - **real**(8) – 8 byte real (double precision)

Kinds for different compilers

- Compiler specific (refer to kinds.f90 and kindfind.f90)
- gfortran supports:
  - integer(1), integer(2), integer(4), integer(8), integer(16) (64 bit only)
  - real(4), real(8) (double precision),

    real(10) – precision of 18 (extended precision)
- Lahey Fortran 95 and Intel Visual Fortran supports:
  - integer(1), integer(2), integer(4), integer(8)
  - real(4), real(8) (double precision),

    real(16) – precision of 33 (quad precision)

Intrinsic kind functions

- **kind**(a) will return the kind of variable *a*
- Use the following functions for portability (since the kind number may not be equal to the number of bytes):
  - **selected_int_kind**(p) returns the kind of integer with *p* significant digits
  - **selected_real_kind**(p, r) returns the kind number of a real with p digit precision and decimal exponent range of -*r* and +*r*
- These functions return -1 if not supported by the compiler, generating a compile error

Specifying the kind of literal constant

- In order to specify the kind for a literal constant, we use an underscore followed by the kind number
- 231_2 – integer(2) literal constant
- 23.13_8 – real(8) literal constant
- More convenient to use parameters

```
integer, parameter :: sp = selected_real_kind(6)
integer, parameter :: dp = selected_real_kind(15)
```

- Then to specify a double precision literal, we type: 3.1416_dp

Kind number example

```
program kind_example
    implicit none
    integer, parameter :: dp = selected_real_kind(15)
    integer, parameter :: ep = selected_real_kind(18)
    real :: a
    double precision :: b, c
    real(ep) :: d
    a = 1.234567890123456789012345678901234567879
    b = 1.234567890123456789012345678901234567879
    c = 1.234567890123456789012345678901234567879_dp
    d = 1.234567890123456789012345678901234567879_ep
    print *, 'real(4) ', a
    print *, 'real(4) ', b
    print *, 'real(8) ', c
    print *, 'real(10) ', d
end program kind_example
```

Intrinsic functions for complex variables

- **real**(z) – returns the real part of z
- **aimag**(z) – returns the imaginary part of z
- **conjg**(z) – return complex conjugate of z
- **abs**(z) – returns magnitude of z
- z = **cmplx**(a, b) – form complex number from two real variables a and b

Initialising variables

- There are three ways to initialise variables with starting values:
  - During the declaration:

```
real :: mass1 = 50.3, mass2 = 100.0
```

  - In a **data** statement (version 1):

```
real :: mass1, mass2,
data mass1, mass2 /50.3, 100.0/
```

  - In a **data** statement (version 2):

```
real :: mass1, mass2,
data mass1 /50.3/, mass2 /100.0/
```

Repeating initialising data

- If a list of variables need to be initialised with the same value, we can repeat them
- Instead of:

```
integer :: a, b, c, d
data a, b, c, d /0, 0, 0, 0/
```

- We can use:

```
integer :: a, b, c, d
data a, b, c, d /4 * 0/
```

- Works for initialising arrays too

Arithmetic operators

- Addition +
- Subtraction -
- Multiplication *
- Division /
- Exponentiation **
  - e.g. 3^{12} => 3 ** 12
  - For integer powers, it is faster to multiply manually

Relational operators

- Greater than **.gt.** or **>**
- Greater than or equal **.ge.** or **>=**
- Less than **.lt.** or **<**
- Less than or equal **.le.** or **<=**
- Equal **.eq.** or **==**
- Not equal **.ne.** or **/=**

Logical operators

- Not (unary) **.not.**
- And **.and.**
- Or **.or.**
- Logical equivalence **.eqv.**
- Logical non-equivalence **.neqv.**

Simple example program

```
program age_program
    implicit none
    integer :: year, age, present_year = 2011
    character(10) :: my_name ! string of 10 chars

    print *, 'What is your name?'
    read *, my_name
    print *, 'What year were you born?'
    read *, year
    age = present_year – year
    print *, 'Hello', my_name, 'you are of age', age
end program age_program
```

Data type conversions

- Need to be careful when mixing different data types in expressions
- Fortran will *sometimes* automatically convert the variables to preserve precision (but still need to check)

```
integer :: a = 3, c = 7
real :: b = 4.3, d
c = a + b ! truncation will occur
d = a + b ! no truncation
print *, c, d
```

Explicit data type conversion

- We can explicitly force data type conversions, just to be certain
- **int**(x) => converts to integer
- **real**(x) => convert to real
- **dble**(x), **dfloat**(x) => convert to double precision
- **cmplx**(x) => converts real/integer to complex (no imaginary part)
- **cmplx**(x, y) => converts real/integer to complex (real x and imaginary y)

Example

```
integer :: a = 3, c = 7
real :: b = 4.3, d
complex :: e
c = a + b
d = real(a) + b
e = cmplx(b, c) ! not e = (b, c)
print *, c, d, e
```

Do loop

- Syntax:

```
do [var = start, end, step]
    statements
end do
```

- *var* must be an integer
- Example:

```
  do i = 1, 10, 2
      total = total + i
  end do
```

Do loop (no counter)

- Infinite do (no var) can be stopped with **exit**

```
do
    if (condition) then
        exit
    end if
end do
```

Implied Do loop

- Can be used for:
  - Initialising arrays
  - Reading a list of values (useful for 2D arrays)
  - Printing a list of values
- Syntax: `(`*`statement`*`, `*`var`*` = `*`start`*`, `*`stop`*`, `*`step`*`)`
- `print *, (i, i = 0, 20, 2)`
- `read *, (table(i), i = 1, 10)`

-
  ```
  real :: values(20)
  values = (/ (0.2 * i, i = 1, 20) /)
  ```

Reading with implied do

- Compare this code:

```
do i = 1, 10
    read *, value(i)
end do
```

- With this code:

```
read *, (value(i), i = 1, 10)
```

- Is there a difference?

While loop

- Introduced in Fortran 90 (some Fortran 77 compilers supported it)
- Syntax:

```
do while (expression)
    statements
end do
```

- Example:

```
do while (total .lt. 100)
    total = total + 1
end do
```

If then else…

- Example:

```
if (discrim .gt. 0) then
    print *, 'There are two roots'
else if (discrim .eq. 0) then
    print *, 'There is a single root'
else
    print *, 'The roots are complex'
end if
```

Select statement

```
select case (option)
    case(1)
        print *, 'Gardening'
    case(2:10)
        print *, 'Clothes'
    case(11, 12, 15, 20:30)
        print *, 'Restaurant'
    case default
        print *, 'Carpark'
end select
```

Arrays

- Arrays are variables that consist of multiple elements of the same type
- Fortran supports up to 7 dimensions
- By default, array indices start at 1
- Examples:

```
integer :: table1(20), vector(5, 5)
complex, dimension(3, 3) :: matrix1, matrix2
table1(1) = 3
vector(2, 3) = -10
matrix1(1, 2) = (-2.4, 9.12)
```

Custom array indices and initialisation

- We can redefine the index range
  - `integer :: number(0 : 10), cost(-10 : 20, 3 : 9)`
- We can initialise (and set) 1D arrays using array literal constants (/ … /)

```
real :: values(3) = (/3.2, 4.7, -1.0/)
integer :: table(2)
table = (/2, 4/)
```

- Note that the number of elements (on the right) must match the array dimensions

Array initialisation using a data statement

- We can also initialise arrays using a data statement (from Fortran 77)

```
real :: readings(5), count(3), voltage(8)
data readings /1.2, 0.9, -2.0, 56.0, -0.89/, &
     count /2.0, 6.0, 7.0/, voltage /8 * 0/
```

- 2D arrays are internally stored as 1D arrays

```
integer :: matrix(2, 2)
data matrix /2, 5, 3, 10/
```

- You can also **reshape**() function (see later)

How multidimensional arrays are stored

- Unlike C, Fortran stores multidimensional arrays using *column-order*
- indices to the left change fastest
- Imagine a 2D array: **integer** :: a(2, 3)

a(1,1)	a(1,2)	a(1,3)
a(2,1)	a(2,2)	a(2,3)

- In Fortran, this array is stored in this order:
  [a(1,1), a(2,1), a(1,2), a(2,2), a(1,3), a(2,3)]

Array slicing

- Like MATLAB, we can select sub-sections or slices of an array
- Syntax: *`array_name`*`(`*`start`*`:`*`end`*`:`*`stride`*`)`

```
integer :: a(10), b(5), total
data a /1, 2, 3, 4, 5, 6, 7, 8, 9, 10/
b = a(3:7) ! a(3),a(4),a(5),a(6),a(7)
b = a(:5)  ! a(1),a(2),a(3),a(4),a(5)
b = a(6:)  ! a(6),a(7),a(8),a(9),a(10)
b = a(1:10:2) ! a(1),a(3),a(5),a(7),a(9)
b = a((/ 2, 5, 7, 8, 9 /)) ! a(2),a(5),a(7),a(8),a(9)
total = sum(a(2:7)) ! Sum is an intrinsic function
```

Array operators

- Note that the arrays must conform with each other
- Addition and subtraction

```
integer :: a(10), b(10), c(10)
c = a + b
```

  - adds each corresponding element
- Multiplication and division
  - c = a * b
  - multiplies each corresponding element
- Exponentiation
  - c = a ** 4.4
  - Raises every element of a to power of 4.4

Where...elsewhere...end where

- Allows us to perform an operation only on certain elements of an array (based on a masking condition)

  e.g. **`where`** `(a > 0.0) a = 1.0 / a`
- This performs a reciprocal only on the elements of array *a* that are positive

Where…elsewhere…end where

- More general form:

```
where (logical expression)
    array operations
elsewhere
    array operations
end where
```

- For example:

```
where (a > 0.0)
    a = log(a) ! log of positive elements
elsewhere
    a = 0.0 ! set negative elements to zero
end where
```

Intrinsic vector/matrix functions

- Transpose
  - a = **transpose**(b)
- Dot product
  - c = **dot_product**(a, b)
- Matrix multiplication (matrices must conform)
  - c = **matmul**(a, b)

Matrix example

- Given two matrices A and B

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \qquad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

- We want to write a Fortran program that computes and prints the following:

$$C = ABA^T$$

Array terminology

- The number of dimensions is called the **rank**
- The number of elements along a single dimension is called the **extent** in that dimension
- Sequence of extents is the **shape** of the array
- Example: `integer :: a(3, 4, 10:15)`
  - Rank is 3
  - Extent of 1st dim = 3, 2nd dim = 4, 3rd dim = 6
  - Shape is `(/ 3, 4, 6 /)`

Inquiring and reshaping arrays

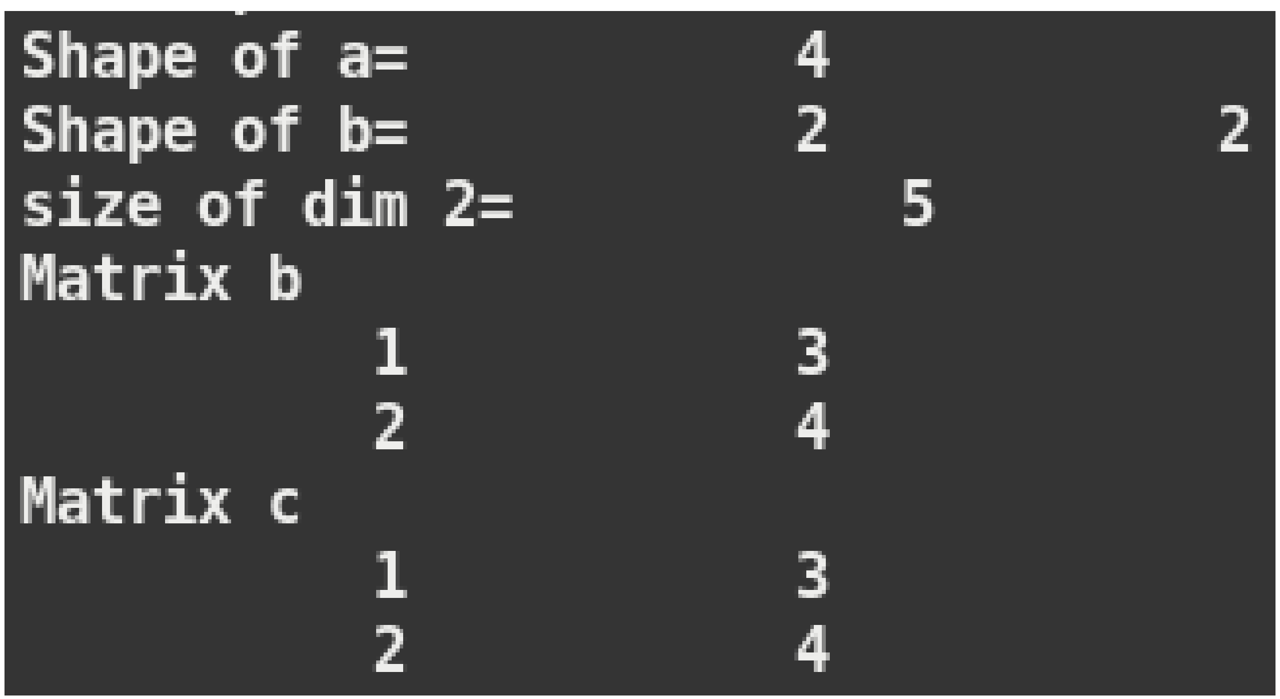
- s = **shape**(a) returns the shape of array *a*
- n = **size**(a) returns the total number of elements in array *a*
- n = **size**(a, i) returns the total number of elements along dimension *i* of array *a*
- c = **reshape**(b, s) function to change shape of array *b* to shape of *a* (stored as *s*)
- e.g. `c = reshape(b, (/3, 4/))`

Example array program

```
program reshape1
    implicit none
    integer :: a(4), b(2, 2), c(2, 2), i, j, d(3, 5)
    data a /1, 2, 3, 4/, b /1, 2, 3, 4/

    print *, 'Shape of a=', shape(a)
    print *, 'Shape of b=', shape(b)
    print *, 'size of dim 2=', size(d, 2)
    c = reshape(a, (/2, 2/))
    print *, 'Matrix b'
    do i = 1, 2
        print *, (b(i, j), j = 1, 2)
    end do
    print *, 'Matrix c'
    do i = 1, 2
        print *, (c(i, j), j = 1, 2)
    end do
end program reshape1
```

Output



```
Shape of a=           4
Shape of b=           2           2
size of dim 2=           5
Matrix b
           1           3
           2           4
Matrix c
           1           3
           2           4
```

Allocatable arrays

- Normal arrays need to have their size fixed and known at compile-time
- Use allocatable arrays if their size is not known

```
real, allocatable :: array(:)
```

- In the code, to allocate this array to *n* elements

```
allocate(array(n))
```

- When not needed, we can free the memory

```
deallocate(array)
```

- You can allocate multidimensional arrays

```
real, allocatable :: array(:, :, :)
```

Formatting screen output

- We have used **print** *, var1, var2, var3
- The asterisk * means use list-directed output (compiler automatically formats the variables)
- For more fine formatting controls, there are two methods:
  - Including a format specifier string
  - Using the **format** statement

Formatting screen output

- Format specifier string:
  - `print '(1x, i5, 5x, f10.8)', index, density`
- Format statement:

```
        print 10, index, density
    10 format(1x, i5, 5x, f10.8)
```

- These examples both use the following formatting for each line:
  - One space (1x), then
  - Integer number with width of 5 places (i5), then
  - Five spaces (5x), then
  - Real number with width of 10 places and 8 decimal places (f10.8)

Formatting specifiers

- *N***x** – N blank spaces
- **i***W* – integer number with W places
- **f***W.D* – real number with W total places (includes decimal point) and D decimal places
- **e***W.DeE* – real number in exponential format with W places, D decimal places, E exponential places (e.g. 0.xxxezz)
- **en***W.D* – engineering notation (e3, e6, e9,…)
- **es***W.D* – scientific notation

More format specifiers

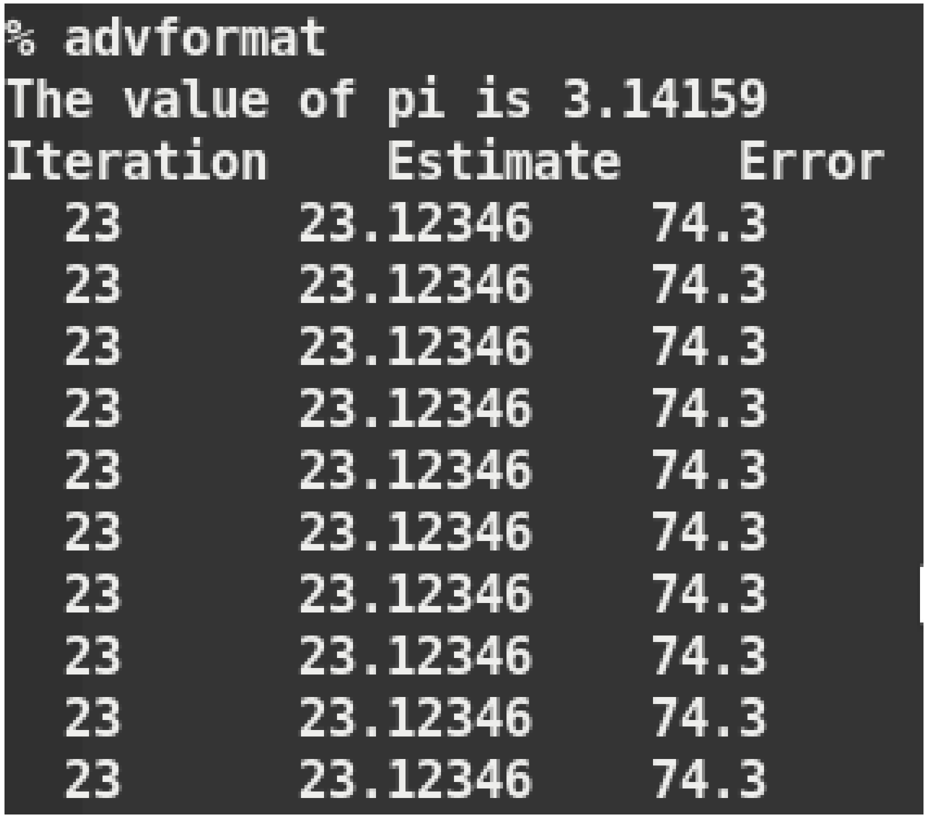
- **a***W* – character with W places
- **t***N* – move to absolute screen position N
- **tl***N* – move N spaces left (relative)
- **tr***N* – move N space right (relative)
- / - new line (end of record)
- *N*() - repeat the format inside () *N* times

Example

```fortran
program advformat
    implicit none
    integer :: iteration = 23, i
    real :: estimate = 23.1234567889776, error = 74.324
    real, parameter :: pi = 3.141592654

    print '("The value of pi is", 1x, f7.5)', pi
    print 5 ! print column headings
    do i = 1, 10
        print 10, iteration, estimate, error
    end do
5   format('Iteration', 4x, 'Estimate', 4x, 'Error')
10  format(i4, 4x, f10.5, 4x, f4.1)
end program advformat
```

Output


```
% advformat
The value of pi is 3.14159
Iteration     Estimate     Error
  23        23.12346     74.3
  23        23.12346     74.3
  23        23.12346     74.3
  23        23.12346     74.3
  23        23.12346     74.3
  23        23.12346     74.3
  23        23.12346     74.3
  23        23.12346     74.3
  23        23.12346     74.3
  23        23.12346     74.3
```

Opening text files for reading/writing

- To open a text file for reading/writing:

```
open(unit = num, file = filename,
     status = mode, [options])
```

- Each file opened has associated with it a unit number (unit = is optional)
- The status sets the mode of the file:
  - **old** – the file already exists and is not to be replaced
  - **new** – create a new file (the file must not exist)
  - **replace** – overwrite existing file (if it exists) or create a new one (if it does not exist)
  - **scratch** – writing to temporary file that is automatically deleted when closed

File opening [*options*]

- **position** = 'append' (appends data to 'old' existing file)
- **action** = read/write/readwrite
  - read – cannot perform writes on this file
  - write – cannot perform reads on this file
  - readwrite – can both write and read

File opening examples

- To open an existing file 'data.dat'

```
open(unit = 10, file = 'data.dat', status = 'old')
```

- To open an existing file 'data.dat' (appends written data)

```
open(10, file = 'data.dat', status = 'old',
position = 'append')
```

- To write data to a new file 'output.txt' that does not exist

```
open(2, file = 'output.txt', status = 'new')
```

- Replace an existing file 'output.dat' or create it if it does not exist

```
open(5, file  'output.dat', status = 'replace')
```

Closing files that are open

- When a file is not used anymore, it is recommended that it be closed
- When writing, closing a file will ensure data in the memory buffer is written to disk
- Syntax:

  `close``(`*`num`*`)`

Writing to a file

- Syntax:

  write(*num*, *fmt*, [*options*]), *var1*, *var2*, ...
- *num* is the unit number of the file (* is the terminal e.g. **write**(*, *) is equivalent to **print** *,)
- *fmt* is the format specifier string (can use *)
- [*options*] include rec, err, end, advance
- If using format specifier string and want to use minimal spacing, we can use a '0 width' for i, a, f (but not e, en, es)

  e.g. **i**0, **a**0, **f**0.10

Writing example

```
    integer :: iter, i
    real :: est, error
       ...
    open(5, file = 'out.dat', status = 'new')
    write(5, 10) ! write column headings
10  format('Iter', 4x, 'Estimate', 4x, 'Error')
    do i = 1, 100
        write(5, 15) iter, est, error
    end do
15  format(i4, 4x, f7.4, 4x, f5.2)
    close(5)
```

Opening a 'new' file that already exists

- Fortran will give a runtime error if opening a 'new' file that already exists
- This prevents the program from overwriting existing files (so it is a safety precaution)
- For example:

```
At line 7 of file simpleWrite.f90
(unit = 5, file = '')
Fortran runtime error: File
'simple.dat' already exists
```

Opening an 'old' file that does not exist

```
At line 7 of file simpleWrite.f90 (unit
= 5, file = '')
Fortran runtime error: File 'simple.dat'
does not exist
```

Reading from a file

- Syntax:

  read(*num*, *fmt*, [*options*]), *var1*, *var2*, ...

- *fmt* = * (list-directed input) is good for most cases
- The format specifiers allow very fine control when numbers are not separated by spaces or decimal points
- For example, if the data file has a real, an integer and a real with no spaces:

  ```
  23.4562433.1416
  ```

- We can use the following format specifier:

  ```
  format(f7.4, i2, f6.4)
  ```

Reading real numbers

- Another example:

  314159 with **f**7.5 would transfer as 3.14159

  f7.4 would transfer as 31.4159
- When reading real numbers with a decimal point and a space after fractional part, *the D part of **f**W.D is overridden and W determines precision*
- For example (note *b* is a blank space) *b*9.3729*b* with **f**8.3 would transfer as 9.3729 (the .3 is overridden)

Reading real numbers

- For example, given the following number

  2.71823453243
- If we read using **f**10.1, we transfer 2.7182345 (.1 is overridden to fill up to 10 places)
- If we read using **f**3.1, we transfer 2.7 (.1 is overridden but still get truncation because only 3 places)
- This only applies to reading, not writing

Reading until end of file

- If we do not know how many values to read, we need to detect the end of file
- `read(unit, fmt, end=num) var1, var2`
- When we reach end of file, program jumps to *num*

```
     do
          read(5, *, end = 20), value
     end do
20   print *, “Reached end of file”
```

- Checking **iostat** (see next slide) is preferred over using **end**

Error handling

- To handle exceptions in read/write, we pass an integer as iostat

```
integer :: ios
read(unit, fmt, iostat = ios), var1, var2
write(unit, fmt, iostat = ios), var1, var2
```

- If the integer `ios` is:
  - negative – end of record or end of file
  - positive – error detected
  - 0 – no problems

Error handling example

```fortran
integer :: ios
real :: a
do
    read(5, *, iostat = ios), a
    if (ios .gt. 0) then
        print *, 'Read error'
        stop  ! abort program
    else if (ios .lt. 0) then
       print *, 'End of file reached'
       exit ! exit the do loop
  end if
end do
```

Backspace and rewind

- **backspace** *unit_num*
  - after each read/write, the file pointer advances to the next record or line
  - backspace moves the pointer back to the previous record or line after advancing
- **rewind** *unit_num*
  - repositions pointer back to the beginning of the file

Backspace example

- Let us say we have an open file (5) that contains:

  1

  2

  3

  4
- read(5, *), value => 1
- read(5, *), value => 2
- read(5, *), value => 3
- backspace 5
- read(5, *), value => 3 (not 2)

Subprograms

- It is good practice to split a large program into smaller subprograms
- Promotes readability and re-usability
- We pass data as arguments to subprograms (dummy variables)
- By default, arguments are *passed by reference* in Fortran (unless explicitly specified)
- Fortran supports two types of subprograms
  - Functions
  - Subroutines

Functions

- Functions accept data as arguments and directly return one value
- e.g. x = sin(theta), z = cmplx(re, im)
- We need to explicitly state the data type of the arguments and return value
- By default, the function name is the return variable
- Or we can use the **result** clause
- There are several ways to declare the return value of a function

Return type of a function

- Method 1:

```
function add(x, y)
    implicit none
    real, intent(in) :: x, y
    real :: add
    add = x + y
end function add
```

- To call this function, c = add(a, b)

Return type of a function

- Method 2:

```
real function add(x, y)
    implicit none
    real, intent(in) :: x, y
    add = x + y
end function add
```

- To call this function, c = add(a, b)

Return type of a function

- Method 3:

```
function add(x, y) result(z)
    implicit none
    real, intent(in) :: x, y
    real :: z
    z = x + y
end function add
```

- To call this function, c = add(a, b)

Return type of a function

- Method 4:

```
real function add(x, y) result(z)
    implicit none
    real, intent(in) :: x, y
    z = x + y
end function add
```

- To call this function, c = add(a, b)

Intent of arguments

- Though optional, it is recommended to state the intent of each argument to a function or subroutine
- Allows compiler to pick up call errors
- There are three types of intent:
  - intent(in)
  - intent(out)
  - intent(inout)
- Arguments with intent(out) and intent(inout) reflect any changes back to calling program (pass by reference)

Example

```
real function estimate(x, error)
    implicit none
    real, intent(in) :: x
    real, intent(out) :: error
    ....
    estimate = [some code]
    error = [approx estimation error]
end function estimate
```

- This function returns the estimate directly but also the estimate error via the arguments

Subroutines

- Subroutines do not return a value directly
- They can indirectly return values via the arguments (intent of **out** or **inout**)
- Though optional, all arguments should have their intent specified
- Use the **call** statement to call a subroutine

Subroutine example

```fortran
subroutine add(x, y, result)
    implicit none
    real, intent(in) :: x, y
    real, intent(out) :: result
    result = x + y
end subroutine add
```

- To call this subroutine:

```fortran
    call add(a, b, c)
```

Alternate returning

- When executation reaches the **end** statement of a function or subroutine, we return back to the calling program
- If we want to return back via an alternate means, we can use the **return** statement

Assumed size arrays and strings

- No need to specify size of arrays and strings *when used as dummy arguments*
- For assumed length strings, use an asterisk *
- For assumed sized arrays, use a colon :
- Example:

```
subroutine process(name, table)
    character(*), intent(in) :: name
    real, intent(inout) :: table(:)
    ....
```

Internal subprograms

- We can define subprograms to be known and used only by the main program (often called utilities)
- These subprograms are **internal** and cannot be called by other main programs
- Used by functions/subroutines that are only useful for the current main program

Syntax for internal subprograms

```
program program_name
    implicit none
    [executable statements]

    contains
        [function or subroutine]
end program program_name
```

Examples of internal subroutines

- First example requires the size of the array to be passed to subroutine
- Second example uses the shape() function so we only need to pass the array
- Subroutine in second example is more generic

```
program internal
    implicit none
    integer :: a(3, 3)
    data a /1, 2, 3, 4, 5, 6, 7, 8, 9/
    call printMatrix(a, 3, 3)

    contains  ! internal subroutine
        subroutine printMatrix(a, numRows, numCols)
            implicit none
            integer, intent(in) :: numRows, numCols, &
               a(numRows, numCols)
            integer :: i, j
            do i = 1, numRows
                print *, (a(i, j), j = 1, numCols)
            end do
        end subroutine printMatrix
end program internal
```

```fortran
program internal
    implicit none
    integer :: a(3, 3)
    data a /1, 2, 3, 4, 5, 6, 7, 8, 9/
    call printMatrix(a)

    contains  ! internal subroutine
        subroutine printMatrix(a)
            implicit none
            integer, intent(in) :: a(:, :)
            integer :: i, j, s(2)

            s = shape(a)
            do i = 1, s(1)
                print *, (a(i, j), j = 1, s(2))
            end do
        end subroutine printMatrix
end program internal
```

Using external subprograms

- We can call functions or subroutines that are external to the program
- This allows us to build up a 'library' of useful procedures for future use
- We can specify which procedures are external (mandatory for functions)
- Two methods we can use:
  - Using **external** statement/attribute
  - Declaring an explicit **interface** (highly recommended)

Example using external function

```fortran
program example1
    implicit none
    real :: x = 3, y = 4
    real, external :: calcRadius

    print *, 'Radius = ', calcRadius(x, y)
end program example1

real function calcRadius(x, y)
    implicit none
    real, intent(in) :: x, y
    calcRadius = sqrt(x ** 2 + y ** 2)
end function calcRadius
```

Example using external subroutine

```
program example2
    implicit none
    real :: x = 3, y = 4, r
    external calcRadius
    call calcRadius(x, y, r)
    print *, 'Radius = ', r
end program example2
subroutine calcRadius(x, y, r)
    implicit none
    real, intent(in) :: x, y
    real, intent(out) :: r
    r = sqrt(x ** 2 + y ** 2)
end subroutine calcRadius
```

External function (using explicit interface)

```
program example1
    implicit none
    real :: x = 3, y = 4
    interface
        real function calcRadius(x, y)
            real, intent(in) :: x, y
        end function calcRadius
    end interface

    print *, 'Radius = ', calcRadius(x, y)
end program example1
```

External subroutine (using explicit interface)

```
program example2
    implicit none
    real :: x = 3, y = 4, r
    interface
        subroutine calcRadius(x, y, r)
            real, intent(in) :: x, y
            real, intent(out) :: r
        end subroutine calcRadius
    end interface
    ...
```

Why use interfaces?

- Allows the compiler to check whether you have the correct arguments to the external function/subroutine call
- If you use **external**, the compiler cannot check the arguments (may cause runtime errors)
- Try removing one of the arguments to calcRadius and see how the compiler reacts
- **external** may be used if the function/subroutine is not directly called (see Newton's method example)

Another example of external danger

```
program main
    implicit none
    real, external :: add
    print *, 'The result is', add(3, 4)
end program main

real function add(a, b) result(c)
    implicit none
    real, intent(in) :: a, b
    c = a + b
end function add
```

OUTPUT:

```
 The result is   9.80908925E-45
```

'Save' variables

- Variables declared in subprograms are destroyed after returning (i.e. volatile)
- We can declare save variables that remain (and retain their value) after the subprogram has finished

```
integer, save :: i
```

- Initialising a variable in a subprogram makes it *automatically* a 'save' variable

```
integer :: i = 0
```

Elemental functions

- Functions with scalar dummy arguments can only be passed scalars, e.g.

```
real function add(a, b)
    real :: a, b
    add = a + b
```

- The above function only adds two scalars, a and b
- If we define it as **elemental**, the function can be used for adding conforming arrays

```
elemental real function add(a, b)
```

Full example of an elemental function

```
program elem
    implicit none
    real, dimension(3) :: a, b
    data a /1., 2., 3./, b /5., 3., 1./
    print *, add(a, b)
    contains
        elemental real function add(a, b)
            real, intent(in) :: a, b
            add = a + b
        end function add
end program elem
```

Modules

- Writing interface blocks is tedious when calling many different external subprograms
- Fortran 90 provides a way of grouping related subprograms (as well as type definitions and variables) into a single library or **module**
- Compiling a module creates a *.mod file that contains the interfaces
- By using a module, we do not need to write an interface block for each subprogram

Syntax for modules

```
module module_name
    [variable declarations]
    [type definitions]

    contains
        [functions and subroutines]

end module module_name
```

Example of a module

```
module griffith_student
    type student
        integer :: number
        character(30) :: name
        real :: gpa
    end type student
    integer :: numStudents, nextNum
    contains
        subroutine add_student(name, gpa)
            ....
        end subroutine add_student
end module griffith_student
```

Using a module

- When compiling a module, a .mod file (e.g. griffith_student.mod) will be created
- To use a module, we use the **use** statement
- e.g.

```
program studentProgram
    use griffith_student
    implicit none
    ....
```

Using modules

- When using a module, all type definitions and variables defined there will be available in the main program
- It is possible to 'hide' variables or subprograms (as private)
- All subroutines and functions can be called without the need for an explicit interface block (the compiler can still check your arguments!!)
- Therefore, it is recommended to put your own subroutines/functions into modules

Recursive functions

- Some mathematical algorithms are recursive by nature
- For example, the factorial:

  5! = 5 x 4!

  4! = 4 x 3!

  3! = 3 x 2!

  etc.
- In Fortran, only recursive functions can call themselves

Recursive functions

- Unlike normal functions, the function name cannot be used as the return variable (why?)
- We add the **result** statement
- For example:

```
recursive function factorial(x) result(res)
    real, intent(in) :: x
    real :: res
    ....
end function factorial
```

Recursive functions

- Another method of declaring the type of the result variable:

  real recursive function factorial(x) **result**(res)

  or

  recursive real function factorial(x) **result**(res)
- Remember to include a stopping condition when using recursive functions
- Use sparingly as they are slow

Example of recursive function

```
real recursive function factorial(x) result(fact)
    real, intent(in) :: x
    if (x .gt. 1) then
        fact = x * factorial(x - 1)
    else
        fact = 1  ! stopping condition
    end if
end function factorial
```

Derived data types

- Arrays can only store elements of the same data type
- Derived data types can group different data variables (known as structures in C)
- Example:

```
type student
    integer :: number
    character(20) :: name
    real :: gpa
end type student
```

Derived data types

- To declare a variable of the derived data type:

```
type(student) :: bob
```

- We use the % operator to access the members

```
bob%name = 'Bob'
bob%number = 12345
bob%gpa = 5.8
```

Example of reading into derived data types

```
type(student) :: adam, eve

adam = student(12345, 'Adam_Sandler', 5.6)
open(10, file = 'typeex.txt', status = 'replace')
write(10, *) adam     ! write using listed I/O
close(10)
open(12, file = 'typeex.txt', status = 'old')
read(12, *) eve     ! read using listed I/O
close(12)
print *, adam
print *, eve
```

Function overloading

- 'Overloading' means calling different subprograms using the same generic name
- Which subprogram is called depends on the type of the arguments
- For example, we might have two versions of the function name func(x)
  - sfunc(x) if x is a real
  - dfunc(x) if x is a double precision
- Overloading allows Fortran to automatically choose to call sfunc() or dfunc() depending on the type of x

Example of function overloading

```fortran
interface func
    real function sfunc(x)
        real, intent(in) :: x
    end function sfunc


    double precision function dfunc(x)
        double precision, intent(in) :: x
    end function dfunc
end interface func
```

- func(x) would call sfunc(x) if x was declared as real and dfunc(x) if x was double precision

Module procedure overloading

- Functions defined in an module already have an explicit interface
- So we don't need to define another explicit interface
- To overload module functions sfunc and dfunc:

```
interface func
    module procedure sfunc, dfunc
end interface func
```

Operator overloading

- Suppose we write a function that processes two derived types (e.g. add)
- Rather than calling the add() function, Fortran 90+ allows us to 'overload' a conforming arithmetic operator
- For example, we can 'overload' the + operator to call the add function
- Makes program more readable

Example of operator overloading

- Let us define a polar coordinate derived type in a module:

```
type polar
    real :: mag
    real :: angle
end type polar
```

- In our module, we define an polarAdd function that performs an addition in rectangular co-ordinates

Example of operator overloading

```
function polarAdd(a, b) result(c)
   type(polar), intent(in) :: a, b
   type(polar) :: c
```

- We can overload the + operator via the interface

```
interface operator(+)
   module procedure polarAdd
end interface
```

- Now when we write c = a + b, it will call

  c = polarAdd(a, b)

Example of operator overloading

- See the full polarComp source code

gfortran optimisation

- We can add the following switches for more better optimisation
  - -O2 or -O3 (level 2 optimisation is default)
  - -funroll-loops
  - -mtune = *arch* (optimises for a particular processor)
    - native (CPU used to compile) , pentium3, pentium4, pentium-m, core2, etc.
  - -mfpmath=sse
  - -mmmx, -msse, -msse2, -msse3 ....
- http://gcc.gnu.org/onlinedocs/gcc-3.4.6/gcc/Optimize-Options.html
- http://gcc.gnu.org/onlinedocs/gcc-4.3.3/gcc/i386-and-x86_002d64-Options.html

Newton's method example (roots2.f90)

- The task is to write an external Fortran function to perform Newton's method of root finding (newton.f90)
- User supplies function and its derivative
- Stops when there is little change in the estimates
- Note the mixed-use of interfaces and external
  - **external** used for func() in main program is fine since it is not directly called here

Statistics module example (statistics.f90)

- This program should read a data file (reg.dat)
- This file contains some header info and then 2 columns of data (x and y)
- Functions and subroutines are written to find mean, variance, and line of best fit
- These subprograms are grouped into a single module (statistics)
- Note that when using modules, there is no need for explicit interface blocks

Static libraries

- Rather than compiling subprograms each time, we can store as object files (*.o) and link into the main program later (saves time)
- Linking lots of object files can be a hassle (esp. if you don't know which ones you are calling)
- A static library (*.a or *.lib) is an archive for storing many related object files into one file
- We simply link the static library with our main program

BLAS and LAPACK libraries

- Basic Linear Algebra Subprograms (BLAS) and Linear Algebra Package (LAPACK)
- BLAS contains routines for vector and matrix multiplication (probably redundant due to Fortran 90 capabilities)
- LAPACK contains routines for linear algebra (EVD, SVD, LU, QR, Cholesky, Schur, etc.)
- Generic BLAS and LAPACK source code can be downloaded from http://www.netlib.org/lapack/
- Commercial Fortran packages often come with optimised LAPACK libraries (Intel MKL, AMD AMCL, etc.)

LAPACK libraries for windows

- Static LAPACK libraries are provided with commercial compilers
- We can build a LAPACK static library using our own compiler
- However, the library usually only links to code compiled using same compiler
- ie. a LAPACK library made using gfortran cannot be linked into a program compiled using Intel Visual Fortran

LAPACK naming convention

- Of the form XYYZZZ
- X, indicates the data type as follows:
  - S REAL
  - D DOUBLE PRECISION
  - C COMPLEX
  - Z COMPLEX*16 or DOUBLE COMPLEX
- YY, indicate the type of matrix (or of the most significant matrix)
- ZZZ, indicate the computation performed

Naming example

- For example, SGEBRD is a
  - single precision routine (S) that;
  - performs a bidiagonal reduction (BRD);
  - of a real general matrix (GE).

Calling LAPACK from Fortran 90

- Need to declare the subroutine as either **external** or via an **interface** block (recommended)
- Read the documentation of the subroutine, copying its arguments into the interface
- Note: Sometimes the documentation will have array arguments with no dimension (*)
- The dimensions of all arrays must be specified exactly!!

Using LAPACK to calculate inverse of a matrix

- LAPACK requires two subroutine calls (in order)
  - sgetrf() - calculates the LU decomposition of A
  - sgetri() - uses LU decomp of A to find its inverse
- Read the documentation for each subroutine
  - sgetrf.pdf or sgetrf.txt
  - sgetri.pdf or sgetri.txt
- Write an interface block for both subroutines

Using LAPACK to calculate eigenvectors and eigenvalues

- We will consider the LAPACK routine **ssyevd** for finding eigenvectors and eigenvalues from a symmetric matrix
- Documentation specifies the necessary size of each matrix (lwork, liwork)
- If working with double precision matrices, use **dsyevd**

Useful links for using LAPACK

- http://www.netlib.org/lapack/lug/ (users guide containing descriptions of all LAPACK routines)

Calling Fortran 90 subprograms from MATLAB

- Combines the speed of Fortran with MATLAB
- There are two ways of doing this:
  - MEX files
  - Loading of dynamic libraries (DLLs)
- MEX files are heavily documented in MATLAB
- We will look at the second method
- Dynamic libraries are similar to static libraries, except that they are dynamically linked at runtime

Creating a dynamic library

- In Windows, the extension is *.dll (in Linux, it is *.so)
- NetBeans can create a dynamic library for you
- In command line (add.f90):
  - `gfortran -c -fPIC add.f90`
  - `gfortran -shared -o libadd.dll -fPIC add.o`

Creating a C header file

- MATLAB requires an accompanying C header file
- Consists of the function name followed by an underscore. e.g. add -> add_
- Example:

```
real function add(a, b)
    real, intent(in) :: a, b
    add = a + b
end function add
```

C header file

- The C header file (add.h) would consist of
  - `extern float add_(float *a, float *b);`
- The corresponding data types in C
  - integer => int
  - real => float
  - double precision => double
  - character => char
- If using array arguments, the size of the arrays must be passed

Loading and calling the DLL in MATLAB

- We use the loadlibrary function in MATLAB

```
loadlibrary('libadd', 'add.h')
```

- Once this has been loaded, we call the function

```
a = calllib('libadd', 'add_', 2, 3)
```