Numerical Modelling in Fortran day 2

Paul Tackley, 2019
Goals for today

• Review main points in online materials you read for homework

• More details about loops

• Finite difference approximation

• Introduce and practice
  – subroutines & functions
  – arrays
program miscellaneous_things
  implicit none
  integer i,j
  real a
  logical equal

  i=2; j=5    ! multiple statements on same line

  ! continuing statements over several lines
  a = 2*i +   &
    3*j

  ! careful with integer constants!
  print*,2/3, 2./3, 2/3. , 2.0/3.0

  ! example use of logical variables
  equal = (i==j)
  print*,equal

  ! use of mod(), min() and max() functions
  do i = 1,10
    print*,mod(i,3),min(i,j),max(i,j)
  end do

  ! real->integer conversion functions
  do i = -8,8
    a = real(i)/4.
    print*,a,int(a),nint(a),floor(a),ceiling(a)
  end do
end program miscellaneous_things
Miscellaneous things

• Continuing lines:
  – f95 use ‘&’ at the end of the line
  – f77: put any character in column 6 on next line

• Formats of constants:
  – Use ‘.’ to distinguish real from integer (avoid 2/3=0 !)
  – 1.234x10^{-13} is written as 1.234e-13

• logical variables have 2 values: .true. or .false.

• Variable naming rules:
  – start with letter
  – mix numbers, letters and _
  – no spaces
Character/string definitions

- `character :: a`  (single character)
- `character(len=10) :: a`  (string of length 10)
- `character :: a*10, b*5`
- `character*15:: a,b`  (fortran77 style)
- `character(len=*) , parameter :: Name= ‘Paul’`
  
  – automatic length, otherwise strings will be truncated or padded with spaces to fit declared length
Initialising variables

• Always initialise variables! Don’t assume they will automatically be set to 0!
• Either
  – when defined, e.g., `real:: a=5.`
  – in the program, e.g., `a=5.0`
  – read from keyboard or file
Arithmetic operators

• e.g., what does a + b * c ** 3 mean?
  – ((a + b) * c) ** 3?  a + (b * c) ** 3?  etc.
  – No! correct is:  a + (b * (c ** 3))
  – priority is **, (* or /) then (+ or -)

• Also LOGICAL operators, in this priority:
  – .not., .and., .or., (.eqv. or .neqv.)

• Be careful mixing variable types (e.g. integer & real) in the same expression!
Some intrinsic mathematical functions

- **abs** (absolute value, real or integer)
- **sqrt** (square root)
- **sin**, **cos**, **tan**, **asin**, **acos**, **atan**, **atan2**: assume angles are in **radians**
- **exp** and **log**: **log** is **natural log**, use **log10** for base 10.
- also **cosh**, **sinh**, **tanh**
- for full list see a manual
Some conversion functions

- **Int**(a): round to smaller # (4.7->4; -4.6->-4)
- **Nint**(a): nearest integer (4.7->5; -4.6->-5)
- **Floor**(a): (4.7->4; -4.6->-5)
- **Ceiling**(a): (4.7->5; -4.6->-4)
- **Float**(i): integer -> real
- **Real**(c): real part of complex
- **mod**(a,b) is remainder of x-int(x/y)*y
- **max**(a,b,c,....), **min**(a,b,c,....)
read(*,*) and write(*,*)

- read(*,*) and write(*,*) do the same thing as read* and print* but are more flexible:
  - The 1st * can be changed to a file number, to read or write from a file
  - The 2nd * can be used to specify the format (e.g., number of decimal places)
- More about this later in the course!
program more_loops
  implicit none
  integer :: j

  print*,'first loop'
  do j = 0,10,2       ! 0 to 10 in steps of 2
    write(*,*), j     ! this does the same as print*
  end do

  print*,'second loop'
  do j = 10,0,-1      ! steps of -1
    print*,j
  end do

  print*,'third loop'
  do ! an infinite loop, unless you EXIT
    print*,'input 1 to exit'
    read(*,*), j       ! does the same as read*,
    if (j==1) exit     ! single line if statement
  end do

  print*,'fourth loop'
  do while (j==1)
    print*,'input something other than 1 to exit'
    read*, j
  end do

end program more_loops
functions and subroutines

• Useful for performing tasks that are performed more than once in a program and/or

• Modularising (splitting up into logical chunks) the code to make it more understandable

• A function returns a value, a subroutine doesn’t (except through changing its arguments)
example functions

integer function sum3(a,b,c)
    implicit none
    integer,intent(in):: a,b,c ! intent is optional
          ! but avoids bugs
    sum3 = a+b+c
end function sum3

!----------------------------------------

integer function factorial(n)
    implicit none
    integer,intent(in) :: n    ! the argument
    integer :: i,a             ! local variables

    a = 1
    do i=1,n ; a=a*i; enddo    ! multiple statements
          ! "enddo" or "end do" same
    factorial = a
end function factorial
same thing as subroutines (less elegant)

```fortran
subroutine sum3(a,b,c,result)
  implicit none
  real,intent(in):: a,b,c   ! intent is optional
  real,intent(out)::result   ! but avoids bugs

  result = a+b+c
end subroutine sum3

!-----------------------------------------------

subroutine factorial(n,result)
  implicit none
  integer,intent(in) :: n   ! the arguments
  integer,intent(out):: result
  integer :: i,a             ! local variables

  a = 1
  do i=1,n ; a=a*i; enddo    ! multiple statements
    ! "enddo" or "end do" same

  result = a
end subroutine factorial
```
Internal vs. external functions

- **Internal** functions (f90-) are contained within the program, and therefore the compiler can link them easily.

- **External** functions are defined outside the main program, so the calling routine must declare their type (e.g., integer, real).
  - In f90 it is also possible to specify the type of all the arguments, using an `explicit interface block`, which has various advantages.
Example internal function

program funcdemo1
    implicit none
    integer :: n=0

    do while (n<1) ! repeats until input is valid
        print*, 'Input a positive integer:
        read*, n
    end do
    print*, n, '! = ', factorial(n)
contains ! this is a key statement

    integer function factorial(n)
        implicit none
        integer, intent(in) :: n
        integer :: i,a
        a = 1
        do i=1,n
            a=a*i
        enddo
        factorial = a
    end function factorial
end program funcdemo1
...and as an external function

```fortran
program funcdemo1
  implicit none
  integer :: n=0
  integer, external :: factorial    ! note this!

  do while (n<1)    ! repeats until input is valid
    print*, 'Input a positive integer:'
    read*, n
  end do
  print*, n, '! = ', factorial(n)
end program funcdemo1

integer function factorial(n)
  implicit none
  integer, intent(in) :: n
  integer :: i, a
  a = 1
  do i=1,n
    a = a * i
  enddo
  factorial = a
end function factorial
```
program array_declarations
  implicit none

  real,dimension(5,5) :: a,b    ! good if several the same size
  real :: c(3,5,7), d(-5:5), e(0:1) ! good if different sizes
  integer, allocatable :: f(:), g(:,:,:), n(3), i

  write(*,'(a,$)') 'Input 3 array dimensions:'
  read*, (n(i), i=1,3)     ! implicit do loop
  allocate( f(n(1)), g(n(1),n(2),n(3)) )

  ! main body of the program goes here

  deallocate (f, g) ! free up memory
end program array_declarations

!---------------------------------------------------------------------

real function sum1DArray (a,n)
  implicit none
  integer, intent(in):: n     ! arguments
  real, intent(in) :: a(n)
  integer i
  ! local variables
  real :: sum=0

  do i=1,n
    sum = sum + a(i)
  end do
  sum1DArray = sum
end function sum1DArray
Notes

• Indices start at 1 and go up to the declared value, e.g., if declare $a(5)$ then it has components $a(1), a(2), \ldots a(5)$

• To get a different lower index, e.g., $a(-5:5)$

• In subroutines & functions an argument can be used to dimension arrays

• In allocate statements other variables can be used

• Use of the $(a,\$)$ format in write avoids carriage return at the end

• Note implicit do loop $n(j), j=1,3$
Homework

• At the ‘Fortran 90 Tutorial’ at http://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/fortran.html

• Read through the sections
  – Selective Execution
  – Repetitive Execution
  – Functions (not modules - yet)

• Do the exercises on the next slides and hand in by email (.f90 files)
Exercise 1: statements & loops

• Write statements to
  – Declare a string of length 15 characters
  – Declare an integer parameter $= 5$
  – Declare a 1-dimensional array with indices running from -1 to +10
  – Declare a 4-dimensional allocatable array
  – Convert a real number to the nearest integer
  – Calculate the remainder after a is divided by b

• Write loops to
  – Add up all even numbers from 12 to 124
  – Test each element of array a(1:100) starting from 1; if the element is positive print a message to the screen and leave the loop
Exercise 2: Mean and standard deviation

- Convert your mean & stddev program from last week into a **function** or **subroutine** that operates on a 1-D array passed in as an argument.

- Write a main program that
  - asks for the number of values,
  - allocates the array,
  - reads the values into the array,
  - calls the function you wrote and
  - prints the result

- A function can’t return both mean & stddev, so one of them will have to be an argument
Exercise 2: Mean and standard deviation

Input file, test case

- Download MeanStdInput.dat
- Type "a.out < MeanStdInput.dat"
- Answer should be:
- Mean & stddev: 4.03999996 1.92520082
Derivatives using finite-differences

- Graphical interpretation: $df/dx(x)$ is slope of (tangent to) graph of $f(x)$ vs. $x$

- Calculus definition:

$$
\frac{df}{dx} \equiv f'(x) \equiv \lim_{dx \to 0} \frac{f(x + dx) - f(x)}{dx}
$$

- Computer version (finite differences):

$$
f'(x) = \frac{f(x_2) - f(x_1)}{x_2 - x_1}
$$
Finite Difference grid in 1-D

- Grid points $x_0$, $x_1$, $x_2$...$x_N$
  - Here $x_i = x_0 + i \cdot h$
- Function values $y_0$, $y_1$, $y_2$...$y_N$
  - Stored in array $y(i)$
- (Fortran, by default, starts arrays at $i=1$, but you can change this to $i=0$)

\[
\left( \frac{dy}{dx} \right)_i \approx \frac{\Delta y}{\Delta x} = \frac{y(i+1) - y(i)}{h}
\]
Concept of Discretization

• True solution to equations is continuous in space and time
• In computer, space and time must be discretized into distinct units/steps/points
• Equations are satisfied for each unit/step/point but not necessarily inbetween
• Numerical solution approaches true solution as number of grid or time points becomes larger
program Deriv1
  implicit none
  integer :: n,i
  real, allocatable :: y(:), dydx(:)
  real :: x, dx

  write(*,'(a,$)') 'Input number of grid points:'; read*, n
  allocate (y(n), dydx(n))  ! allocate grid arrays

  dx = 10.0/(n-1)  ! grid spacing, assuming x from 0->10
  do i = 1, n
    x = (i-1)*dx
    y(i) = cos(x)  ! fill with cosine
  end do

  call derivative (y, n, dx, dydx)  ! calculate dydx

  do i = 1, n  ! write result, -sin(x) and error
    x = (i-1)*dx
    print*, dydx(i), -sin(x), -sin(x)-dydx(i)
  end do

  deallocate(y, dydx)  ! finish
contains

  subroutine derivative (a, np, h, aprime)  ! argument names different
    integer, intent(in) :: np  ! declare arguments
    real , intent(in) :: a(np), h
    real , intent(out) :: aprime(np)
    integer :: i  ! local variable

    do i = 1, np-1
      aprime(i) = (a(i+1)-a(i))/h  ! finite-difference formula
    end do
    aprime(np) = 0.

  end subroutine derivative
end program Deriv1
program Deriv1
  implicit none
  integer :: n,i
  real, allocatable :: y(:,), dydx(:)
  real :: x, dx

  write(*,'(a,$)') 'Input number of grid points:'; read*,n
  allocate (y(n), dydx(n)) ! allocate grid arrays

  dx = 10.0/(n-1) ! grid spacing, assuming x from 0->10
  do i = 1,n
    x = (i-1)*dx
    y(i) = cos(x) ! fill with cosine
  end do

  call derivative (y,n,dx,dydx) ! calculate dydx

  do i = 1,n ! write result, -sin(x) and error
    x = (i-1)*dx
    print*, dydx(i), -sin(x), -sin(x)-dydx(i)
  end do

  deallocate(y,dydx) ! finish
contains

contains

subroutine derivative (a,np,h,aprime) ! argument names different
   integer, intent(in) :: np                ! declare arguments
   real   , intent(in) :: a(np),h
   real   , intent(out) :: aprime(np)
   integer           :: i                   ! local variable

   do i = 1,np-1
      aprime(i) = (a(i+1)-a(i))/h           ! finite-difference formula
   end do
   aprime(np) = 0.

end subroutine derivative

end program Deriv1
Analysis

• Subroutine arguments can have different names from those in calling routine: what matters is order
• FD approximation becomes more accurate as grid spacing dx decreases
• Allocate argument arrays in the calling routine, *not* in the subroutine/function
Summary: first derivative

\[ \frac{dy}{dx} \approx \frac{\Delta y}{\Delta x} = \frac{y_i - y_{i-1}}{x_i - x_{i-1}} = \frac{y_i - y_{i-1}}{h} \]

• Second derivative

\[ \left( \frac{\partial^2 y}{\partial x^2} \right)_i = \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} \]
Exercise 3: Second derivative

• Write a subroutine that calculates the second derivative of an input 1D array, using the finite difference approximation
  • The inputs will be the array, number of points and grid spacing.
  • The resulting 1-D array can be an intent(out) argument.
  • The 2\textsuperscript{nd} derivative will be calculated at i=2…n-1
  • Assume the derivative at the end points (i=1 and n) is 0.
• Test this routine by writing a main program that calls the subroutine with two idealized functions for which you know the correct answer, e.g., sin(x), x**2. Write out the your code’s result, the correct result, and the error
• Hand in (to ETHFortran@gmail.com) your .f90 code and the results of your two tests