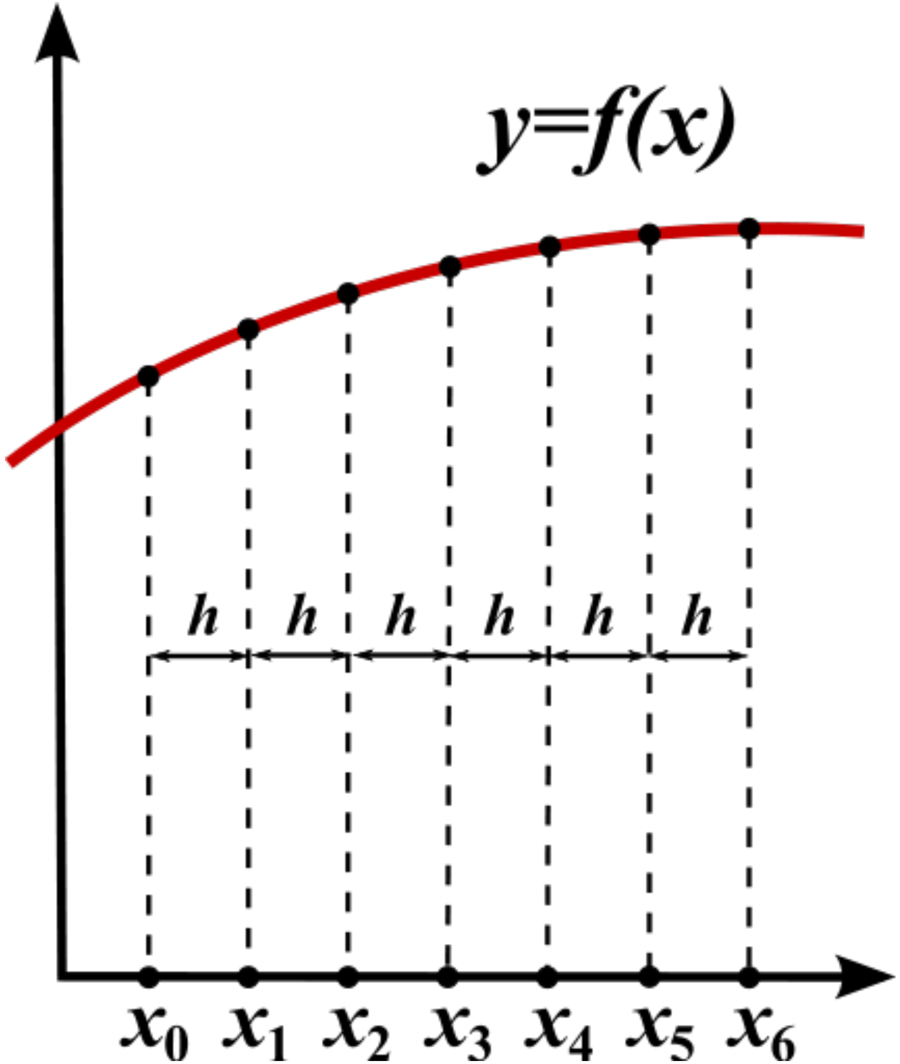
Numerical Modelling in FORTRAN day 3

Paul Tackley, 2014

Today's Goals

1. Review subroutines/functions and finite-difference approximation from last week
2. Review points from reading homework
   – Select case, stop, cycle, etc.
3. Input/output to ascii files
4. Interface blocks for external subroutines
5. **Modules**
6. Arrays: Initialisation and array functions
7. Application to solve 1-D diffusion equation

Finite Difference grid in 1-D



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

$$\left(\frac{dy}{dx}\right)_i \approx \frac{\Delta y}{\Delta x} = \frac{y(i+1) - y(i)}{h}$$

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):

$$f'(x) = \frac{f(x_2) - f(x_1)}{x_2 - x_1}$$

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

2. Review important points from online reading

- **select case** statement
  - does same thing as **if...elseif...else**..
  - good for taking different actions based on different outcomes of a single test
  - example on next slide

```fortran
program casedemo

  implicit none
  integer :: i
  integer,parameter :: low=3, high=5

  ! This program does nothing useful

  do i = 1,10       ! repeats loop with i=1,2,3...10

     select case (i)
     case (high+1:)      ! means >high
        print*,i," is greater than",high
     case (:low)         ! means <=low
        print*,i," is less or equal to",low
     case default
        print*,i," is nothing special"
     end select

  end do

end program casedemo
```

i=high+i; infinite

i<low

'if' version from class 1

```
program loopdemo

  implicit none
  integer :: i
  integer,parameter :: low=3, high=5

  ! This program does nothing useful

  do i = 1,10        ! repeats loop with i=1,2,3...10

     if (i>high) then
        print*,i," is greater than 5"
     else if (i<=low) then
        print*,i," is less than 3"
     else
        print*,i," is nothing special"
     end if

  end do

end program loopdemo
```

more things

- single line **if** (example next slide)
- **stop** to finish execution (example next slide)
- nested **do** loops (example next slide)
- nested **if** blocks (example in reading)
- **cycle** inside a counted **do** loop goes to next value before reaching **end do**.
  - don’t use this, it makes the code confusing

如果if后只有一个statement，不用end if

stop后的东西都不会执行；
exit 退出循环

```fortran
program variousthings

  implicit none
  integer n,i,j

  do
     read*,n

     if (n==2) print*,'i equals 2' ! SINGLE LINE IF

     if (n==0) then
        print*,'You entered 0 so I am stopping'
        stop     ! STOP command
     end if

     do i=1,n    ! nested DO loops
        do j=1,n
           print*,'i,j=',i,j
        end do
     end do
  end do

end program variousthings
```

‘do’ loop counters (do a=a1,a2,a3)

- Up to f90: a* can be real or integer
- F95 onwards: must be integer
  - gfortran gives an error if real
  - ifort accepts real
- Conclusion: stick to integer so your code works on any computer/compiler

```
program testDO
  real:: a,b
  integer:: i

  do a=0.,5.,0.1   ! real
     print*,a
  end do

  do i=0,50     ! integer
     a=i/10.0; print*,a
  end do

end program testDO
```

Note inexact numbers with first version:

```
0.0000000E+00
0.1000000
0.2000000
0.3000000
0.4000000
0.5000000
0.6000000
0.7000000
0.8000001
0.9000001
…
4.699998
4.799998
4.899998
4.999998
0.0000000E+00
0.1000000
0.2000000
0.3000000
0.4000000
0.5000000
0.6000000
0.7000000
0.8000000
0.9000000
…
4.700000
4.800000
4.900000
5.000000
```

Input & Ouput to ascii files

- Use open() and close(), specifying a file number

integer

- The file number can be anything except 5 and 6, which correspond to the screen & keyboard
- Use the read() and write() statements replacing the first * with the file number
- An output example next slide:

outputs array to text file

```
program fileIO

  implicit none
  integer n,i
  real,allocatable:: a(:)

  write(*,'(a,$)') 'How many random numbers?'
  read*,n
  allocate (a(n))
  call random_number(a)

  open(2,file='stuff.dat')
  do i=1,n
     write(2,*) a(i)
  end do
  close(2)

end program fileIO
```

$ means ~~it~~ will not go to next line

* means default.

input & output (2)

- The file stuff.dat can be read into MATLAB using “load stuff.dat”, then plot
- We will need to do this for visualising results!
- Reading into f95 is easy if you know how many numbers there are, but otherwise requires care! Examples follow.
- Make sure there is a carriage return after the last line of the file!

reading from ascii file

如果没有old, 系统会自动创建。读取文件是加上old

```
program fileread ! file starts with #of points
   implicit none
   integer n,i
   real,allocatable:: a(:)

   open(1,file='data.dat',status='old')

   read(1,*) n
   allocate (a(n))
   do i = 1,n
      read(1,*) a(i)
   end do

   print*,a

end program fileread
```

return a value giving you information whether successful or not
i=0 means it reaches the end

```
program fileread ! unknown #of points
  implicit none
  integer n,i
  real,allocatable:: a(:)
  real b

  open(1,file='stuff.dat',status='old')

  n = 0
  do     ! loop to check how many values
     read(1,*,iostat=i) b
     if (i<0) exit
     if (i/=0) stop 'error reading data'
     n = n + 1
  end do

  print*,'found',n,'values'
  allocate (a(n))

  rewind(1)   ! moves file pointer back to start
  do i = 1,n    ! now read them into a
     read(1,*) a(i)
  end do

  print*,a

end program fileread
```

Discussion

- **iostat** as 3rd argument
  - 0 means successful read
  - <0 means end of file
  - >0 means some other error
- **rewind** to move back to start of file
- **status='old'** means the file must already exist, otherwise program will stop with an error
  - If this is not specified and the file does not exist, then a new file will be created

Interface blocks for **External functions**

(f90-)

- Defines all arguments in addition to function type
- All functions can be listed in one interface block
- Advantages
  - minimises bugs: compiler checks arguments
  - allows implicit size arrays (# of elements is passed in)
  - allows optional arguments and n=4 type syntax
- Disadvantage
  - makes code longer and messier
  - If you change the function arguments then they must also be changed in all the interface blocks
- Recommendation: Use **modules** instead of external functions

recall this from class 2

```
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
```

- with interface added

```
program interfacedemo
   implicit none

   interface     ! INTERFACE BLOCK
      integer function factorial(n)
        implicit none
        integer,intent(in) :: n
      end function factorial
   end interface

   integer :: n=0

   do while (n<1)
      print*,'Input a positive integer:'
      read*,n
   end do
   print*,n,'! =',factorial(n)
end program interfacedemo

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
```

MODULES (f90- only)

- **Modules** are collections of variables and/or functions/subroutines that are
  - defined outside main program
  - can be **used** in a main program or other subroutine, function or module
- The best way of sharing variables between different routines
  - replaces f77 **common** blocks
- The best way of defining functions and subroutines that are used in several places

MODULES general form

- **module** *name*
- *variable definitions*
- **contains**
- *functions & subroutines*
- **end module** *name*

```
module fact
   ! no variables in this module
contains
   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 module fact

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

program moddemo     ! MAIN PROGRAM
   use fact
   implicit none

   integer :: n=0

   do while (n<1)
      print*,'Input a positive integer:'
      read*,n
   end do
   print*,n,'! =',factorial(n)
end program moddemo
```

"use" could only use the compiled modules.

- main program is typically in a different file- to compile specify all source files after gfortran

this one has only numbers

```fortran
module useful_stuff
  implicit none
  real,parameter:: pi=3.1415926, &
       days_in_year=365.25,      &
       earth_radius=6.37e6
end module useful_stuff

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

program mod_demo
  use useful_stuff
  implicit none
  real distance

  distance = 2*pi*earth_radius* &
       days_in_year

  print*,'We travel',distance,'meters/year'
end program mod_demo
```

f77 things that you shouldn't use in f95

- **common** blocks: contain a list of variables to be shared with other routines having same common block. Use modules instead
- **include** statement: includes a text file (e.g., containing a common block definition). Use modules instead
- **goto**: use proper control structures like if...endif, do while, do...exit, case… etc.

Returning an array from a function

- Normally, a function returns a single number, but you can return an array if you define it carefully, either as:
  - External function with interface block
  - Internal function
  - Module function

Array function as external function: use interface block

```
program fntest
  implicit none

  interface
     function arrayAdd(a,b,n)
       implicit none
       real,dimension(n):: arrayAdd
       integer,intent(in):: n
       real,dimension(n),intent(in):: a,b
     end function arrayAdd
  end interface

  integer,parameter:: n=10
  real,dimension(n):: x,y

  call random_number(x); call random_number(y)
  print*,arrayAdd(x,y,n)

end program fntest


function arrayAdd(a,b,n)
  implicit none
  real,dimension(n):: arrayAdd
  integer,intent(in):: n
  real,dimension(n),intent(in):: a,b

  arrayAdd = a+b

end function arrayAdd
```

dimension of arrayadd

Fortran allows to add array together.

As internal function

```
program fntest
  implicit none
  integer,parameter:: n=10
  real,dimension(n):: x,y

  call random_number(x); call random_number(y)
  print*,arrayAdd(x,y,n)


contains

  function arrayAdd(a,b,n)
    implicit none
    real,dimension(n):: arrayAdd
    integer,intent(in):: n
    real,dimension(n),intent(in):: a,b

    arrayAdd = a+b

  end function arrayAdd

end program fntest
```

as a module

```
module addfn
contains

  function arrayAdd(a,b,n)
    implicit none
    real,dimension(n):: arrayAdd
    integer,intent(in):: n
    real,dimension(n),intent(in):: a,b
    arrayAdd = a+b
  end function arrayAdd

end module addfn

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

program fntest
  use addfn
  implicit none
  integer,parameter:: n=10
  real,dimension(n):: x,y

  call random_number(x); call random_number(y)
  print*,arrayAdd(x,y,n)

end program fntest
```

we can skip this

arrayAdd with no length argument!

```
program fntest
  implicit none
  integer,parameter:: n=10
  real,dimension(n):: x,y

  call random_number(x); call random_number(y)
  print*,arrayAdd(x,y)

contains

  function arrayAdd(a,b)
    implicit none
    real,dimension(:),intent(in):: a,b
    real,dimension(size(a)):: arrayAdd

    arrayAdd = a+b

  end function arrayAdd

end program fntest
```

get the size of array.

Version with 2-dimensional arrays

```
program fntest
  implicit none
  integer,parameter:: n=10,m=5
  real,dimension(n,m):: x,y

  call random_number(x); call random_number(y)
  print*,arrayAdd(x,y)

contains

  function arrayAdd(a,b)
    implicit none
    real,dimension(:,:),intent(in):: a,b
    real,dimension(size(a,1),size(a,2)):: arrayAdd

    arrayAdd = a+b

  end function arrayAdd

end program fntest
```

automatically

number of dimension

Array initialisation, **data, reshape**

Array initialisation examples:

- real:: a(5)=(/1.2, 3.4, 5.6, 7.8, 9.0/)
- integer:: d(10)=(/i=1,10/)
- real:: x(3)=(/tan(x),sin(x),cos(x)/)

can use built in functions

data statement examples

Load value

- data a /1.2, 3.4, 5.6, 7.8, 9.0/
- data b /4*1.2/   ! same as /1.2,1.2,1.2,1.2/

reshape example (converts 1D list to multiD array)

- real:: a(2,2)
- a=reshape((/1., 2., 3., 4./) , (/2,2/))

Homework

- Finish the exercises and read from
- http://www.cs.mtu.edu/%7eshene/COURSES/cs201/NOTES/fortran.html
  - Functions and modules (particularly modules and interface blocks)
  - Subroutines
  - One dimensional arrays

Exercises

1. Write new **module** versions of last weeks subprograms (i.e., 1. mean&std.dev; 2. second derivative). Then **use** these in the next two exercises:
2. Write a main program that
   - reads numbers from an ascii file (one number per line, the program should sense how many as in the example program given),
   - Uses your module to calculate the mean & standard deviation, and
   - writes the answers to the screen
3. Write a main program that solves (i.e., steps forward in time) the 1-D diffusion equation, as detailed on the next slide

The diffusion equation

Diffusion of T $$\frac{\partial T}{\partial t} = \kappa \frac{\partial^2 T}{\partial x^2}$$

Simplify by assuming kappa=1

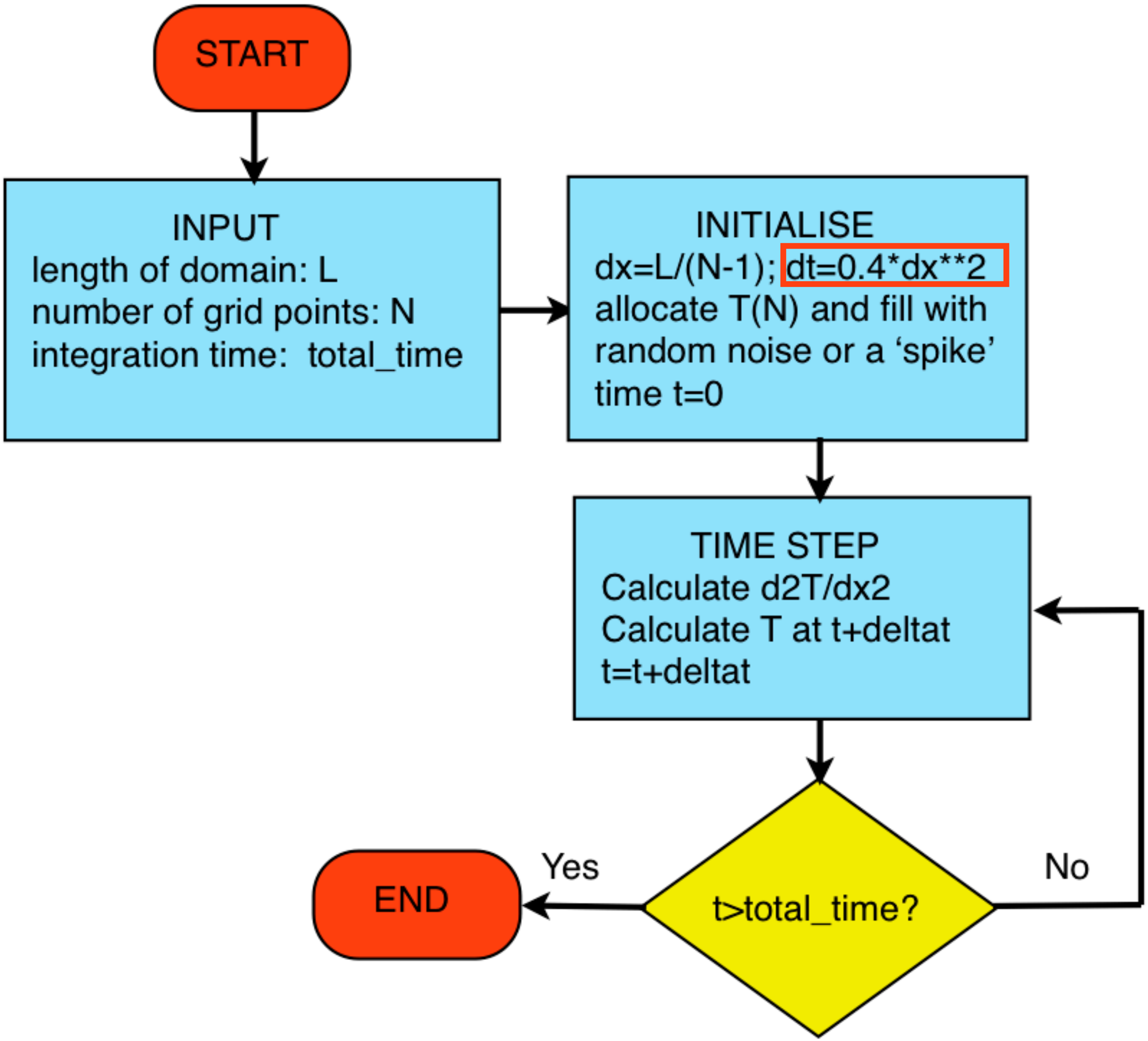
Represent T on a series of evenly-spaced grid points in space x

Calculate T at the next timestep using the explicit finite-difference time derivative

$$\frac{T_i^{t+\Delta t} - T_i^t}{\Delta t} = \left(\frac{\partial^2 T}{\partial x^2}\right)_i^t$$

hence $$T_i^{t+\Delta t} = T_i^t + \Delta t \left(\frac{T_{i-1}^t + T_{i+1}^t - 2T_i^t}{\Delta x^2}\right)$$

Where the 2nd x derivative is calculated in your module, using the equation from last week


START
INPUT
length of domain: L
number of grid points: N
integration time: total_time
INITIALISE
dx=L/(N-1); dt=0.4*dx**2
allocate T(N) and fill with
random noise or a 'spike'
time t=0
TIME STEP
Calculate d2T/dx2
Calculate T at t+deltat
t=t+deltat
t>total_time?
Yes
No
END

Solving 1D diffusion equation

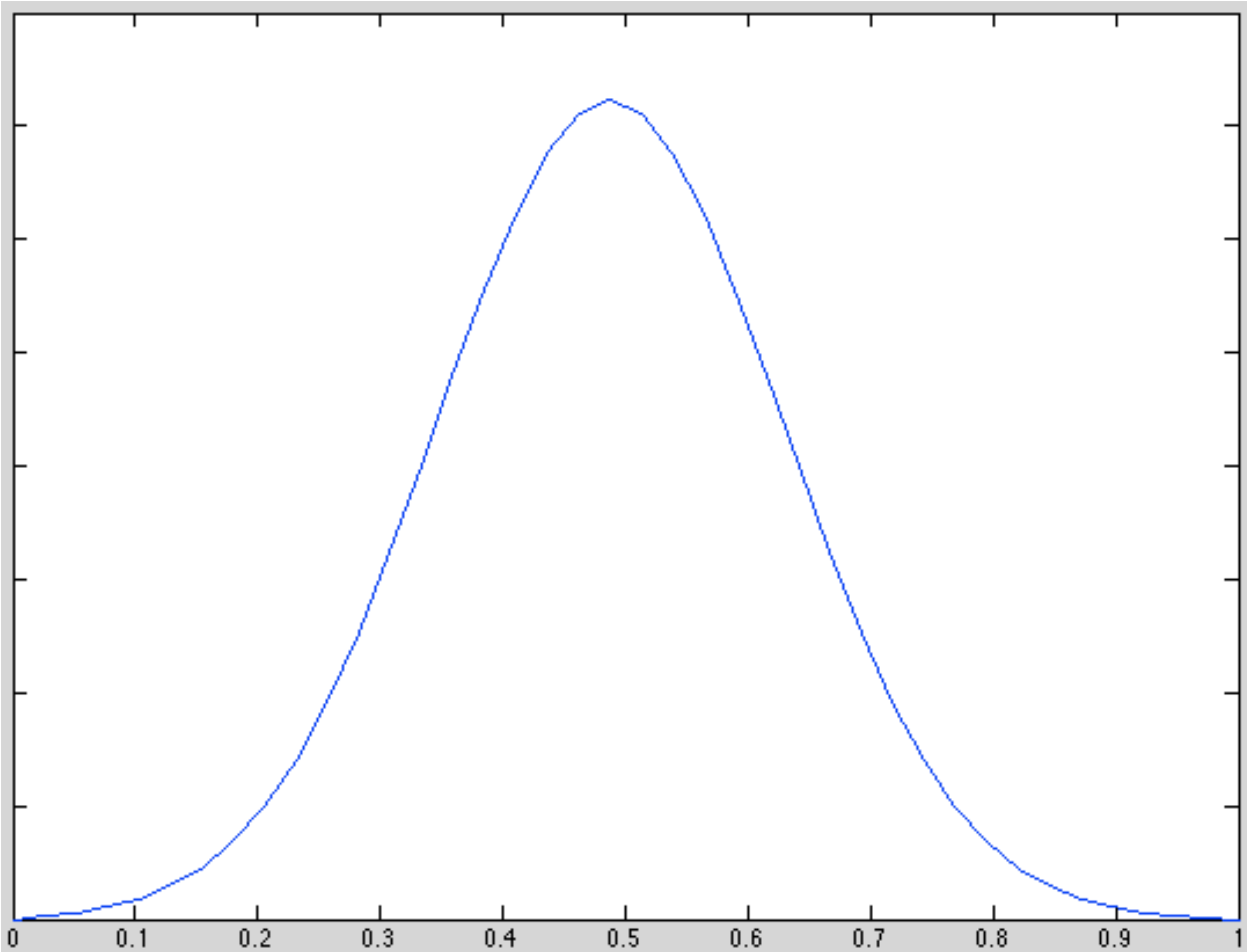
- Ask the user *L, N, total_time* (see flow chart)
- Initialise the field with random noise or a delta function (spike) and write to an ascii file
- Take several time steps. For each timestep:
  - Calculate the second derivative of the field
  - Use explicit time integration to calculate the field a time deltat later
  - Boundary conditions T=0
- Write the final field to an ascii file
- Plot the initial and final field using e.g., MATLAB or Excel, and hence check the code is working correctly! If the time step is too large it should go unstable!

Boundary conditions

- Assume T=0 at the boundaries
- Make sure your initial T field has T=0 at the boundary points
- Make sure the T field has T=0 at the boundary points after each time step
- You can ignore the boundaries when calculating del-squared (set to 0)

TEST CASE

- L=1; Total_time=0.01; initial spike in centre



Hand in

- .f90 or .f95 files for module and 2 programs that use it
- A graph showing the result of the diffusion test case on the previous slide (plotted using excel, matlab or another program of your choice)