Numerical Modelling in FORTRAN day 3

Paul Tackley, 2020

Today's Goals

1. Useful debugging options
2. Review points from reading homework
   - Select case, stop, cycle, etc.
3. **Input/output** to ascii (text) files & **namelist input**
4. **Modules**
5. **Interface blocks** for external subroutines
6. **Arrays**: Initialisation and array functions
7. Application to solve **1-D & 2-D diffusion equations**

Debugging your code: Useful compiler options

- Reporting the line number of an error
  - gfortran –fbacktrace program.f90
  - ifort –traceback program.f90
- Checking the bounds of arrays
  - gfortran –fcheck=bounds program.f90
  - ifort –check bounds program.f90
- Stopping when a floating point error occurs
  - gfortran -ffpe-trap=invalid,zero,overflow,denormal,inexact
  - ifort –fpe0

Debugging your code: Useful compiler options

- Detecting uninitialized variables
  - gfortran –finit-real=snan program.f90
  - ifort –check bounds –init=snan,arrays program.f90
- Allowing the use of a debugger (gdb,idb)
  - gfortran –g program.f90
  - ifort –g program.f90
- Switching off optimization (which can sometimes generate erroneous code)
  - gfortran/ifort –O0 program.f90
- Consult documentation (e.g. “man gfortran”)

Debugging your code: Useful compiler options

- Examples with many checks switched on:
  - gfortran –fbacktrace –fcheck=all –finit-real=snan -ffpe-trap=invalid,zero,overflow –O0 program.f90
  - ifort –traceback –check uninit,bounds –init=snan,arrays –fpe0 –O0 program.f90

- Note: The program will run more slowly with checking options switched on. Use only when developing/debugging your program.
- Note: instead of typing all this every time use e.g. alias gf="gfortran –fbacktrace … –O0 "

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
  - the control variable must be of type integer, logical or character
  - example on next slide

```
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
```

‘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 or equal to 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)
  - Since fortran 2008 there is also error stop
- 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

```
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
```

Inexact computer arithmetic

- Because computers store numbers & calculate in binary (base 2), and some exact decimal (base 10) numbers cannot be represented exactly in binary
- Similarly, e.g. 1/3 or 1/7 cannot be exactly represented in decimal.

Input & Ouput to ascii (text) files

- Use open() and close(), specifying a file number
  - The file number can be anything except 5 and 6, which correspond to the screen & keyboard (i.e. stdout and stdin)
- 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
```

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

```
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
```

Tape recorder

```
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

namelist input

- A handy, flexible way of reading input parameters from a text file
- The list of variables is defined in the program
- In the file they have the same names but can be in any order
- Not all variables need to be present in the file, so make sure to set defaults!
- One file can contain several namelists
- See example next slide
- Need a *carriage return* at the end of par file

```
program namelistdemo

  implicit none
  integer:: nx=1,ny=1    ! ngrid points
  real:: time=0.         ! good to define default values
  character(len=50):: outputfilename='xx'

  namelist /inputs/ nx,ny,time,outputfilename

  open(1,file='parameters',status='old')

  read(1,inputs)  ! read inputs from file 1
  close(1)

  write(*,inputs) ! echo values to stdout

  ! numerical computation goes here

end program namelistdemo
```

```
&inputs
        nx=10, ny=20
        outputfilename='opfile'
        time=4.5
/  Use a carriage return after /
```

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
```

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

This one has only numbers

```
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
```

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
```

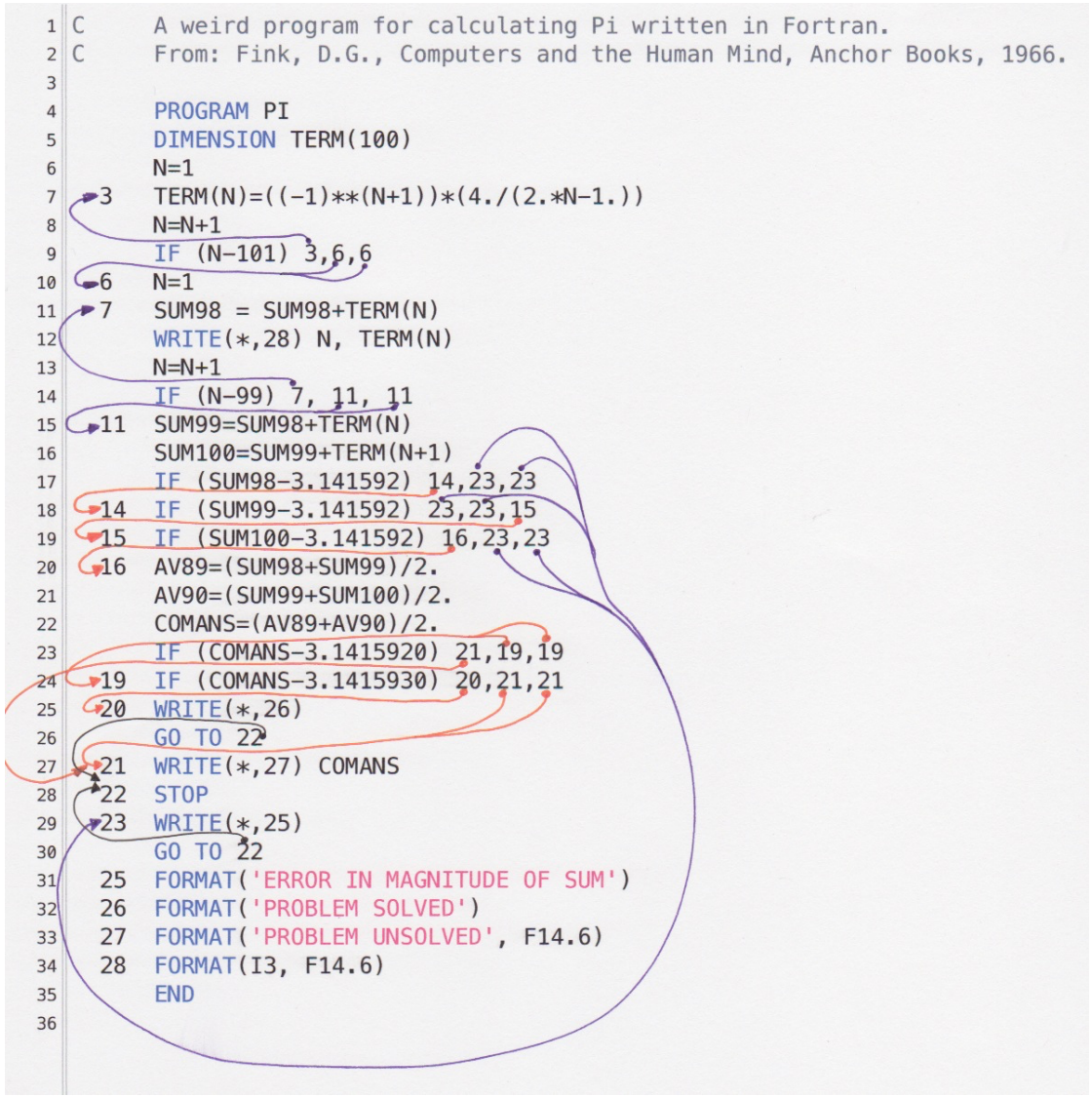
- with interface added

```
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
```

f77 things that you shouldn't use

- **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.

’Spaghetti Fortran’ example



```
C     A weird program for calculating Pi written in Fortran.
C     From: Fink, D.G., Computers and the Human Mind, Anchor Books, 1966.

      PROGRAM PI
      DIMENSION TERM(100)
      N=1
3     TERM(N)=((-1)**(N+1))*(4./(2.*N-1.))
      N=N+1
      IF (N-101) 3,6,6
6     N=1
7     SUM98 = SUM98+TERM(N)
      WRITE(*,28) N, TERM(N)
      N=N+1
      IF (N-99) 7, 11, 11
11    SUM99=SUM98+TERM(N)
      SUM100=SUM99+TERM(N+1)
      IF (SUM98-3.141592) 14,23,23
14    IF (SUM99-3.141592) 23,23,15
15    IF (SUM100-3.141592) 16,23,23
16    AV89=(SUM98+SUM99)/2.
      AV90=(SUM99+SUM100)/2.
      COMANS=(AV89+AV90)/2.
      IF (COMANS-3.1415920) 21,19,19
19    IF (COMANS-3.1415930) 20,21,21
20    WRITE(*,26)
      GO TO 22
21    WRITE(*,27) COMANS
22    STOP
23    WRITE(*,25)
      GO TO 22
25    FORMAT('ERROR IN MAGNITUDE OF SUM')
26    FORMAT('PROBLEM SOLVED')
27    FORMAT('PROBLEM UNSOLVED', F14.6)
28    FORMAT(I3, F14.6)
      END
```

Difficult to understand!

(*https://craftofcoding.wordpress.com/tag/spaghetti-code/*)

Example from https://www.polyhedron.com/spag0html

Example 2 - Original Fortran 66.

This subroutine picks off digits from an integer and branches depending on their value.

```
SUBROUTINE OBACT(TODO)
      INTEGER TODO,DONE,IP,BASE
      COMMON /EG1/N,L,DONE
      PARAMETER (BASE=10)
   13 IF(TODO.EQ.0) GO TO 12
      I=MOD(TODO,BASE)
      TODO=TODO/BASE
      GO TO(62,42,43,62,404,45,62,62,62),I
      GO TO 13
   42 CALL COPY
      GO TO 127
   43 CALL MOVE
      GO TO 144
  404 N=-N
   44 CALL DELETE
      GO TO 127
   45 CALL PRINT
      GO TO 144
   62 CALL BADACT(I)
      GO TO 12
  127 L=L+N
  144 DONE=DONE+1
      CALL RESYNC
      GO TO 13
   12 RETURN
      END
```

Example 2 - Fortran 90 Extensions.

SPAG has used DO WHILE, SELECT CASE, EXIT and CYCLE. No GOTOs or labels remain.

```
SUBROUTINE OBACT(Todo)
      IMPLICIT NONE
C*** Start of declarations inserted by SPAG
      INTEGER act , LENgth , NCHar
C*** End of declarations inserted by SPAG
      INTEGER Todo , DONe , BASE
      COMMON /EG1   / NCHar , LENgth , DONe
      PARAMETER (BASE=10)
      DO WHILE ( Todo.NE.0 )
         act = MOD(Todo,BASE)
         Todo = Todo/BASE
         SELECT CASE (act)
         CASE (1,4,7,8,9)
            CALL BADACT(act)
            EXIT
         CASE (2)
            CALL COPY
            LENgth = LENgth + NCHar
         CASE (3)
            CALL MOVE
         CASE (5)
            NCHar = -NCHar
            CALL DELETE
            LENgth = LENgth + NCHar
         CASE (6)
            CALL PRINT
         CASE DEFAULT
            CYCLE
         END SELECT
         DONe = DONe + 1
         CALL RESYNC
      ENDDO
      RETURN
      END
```

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
```

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
```

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
```

Version with 2-dimensional arrays & allocation

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

  allocate(x(n,m),y(n,m))
  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
    print*,shape(a) ! just for information

  end function arrayAdd

end program fntest
```

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)/)

data statement examples

- 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 following 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 exercises 2 & 3:
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
4. Modify your 1-D diffusion program to solve the 2-D diffusion equation

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

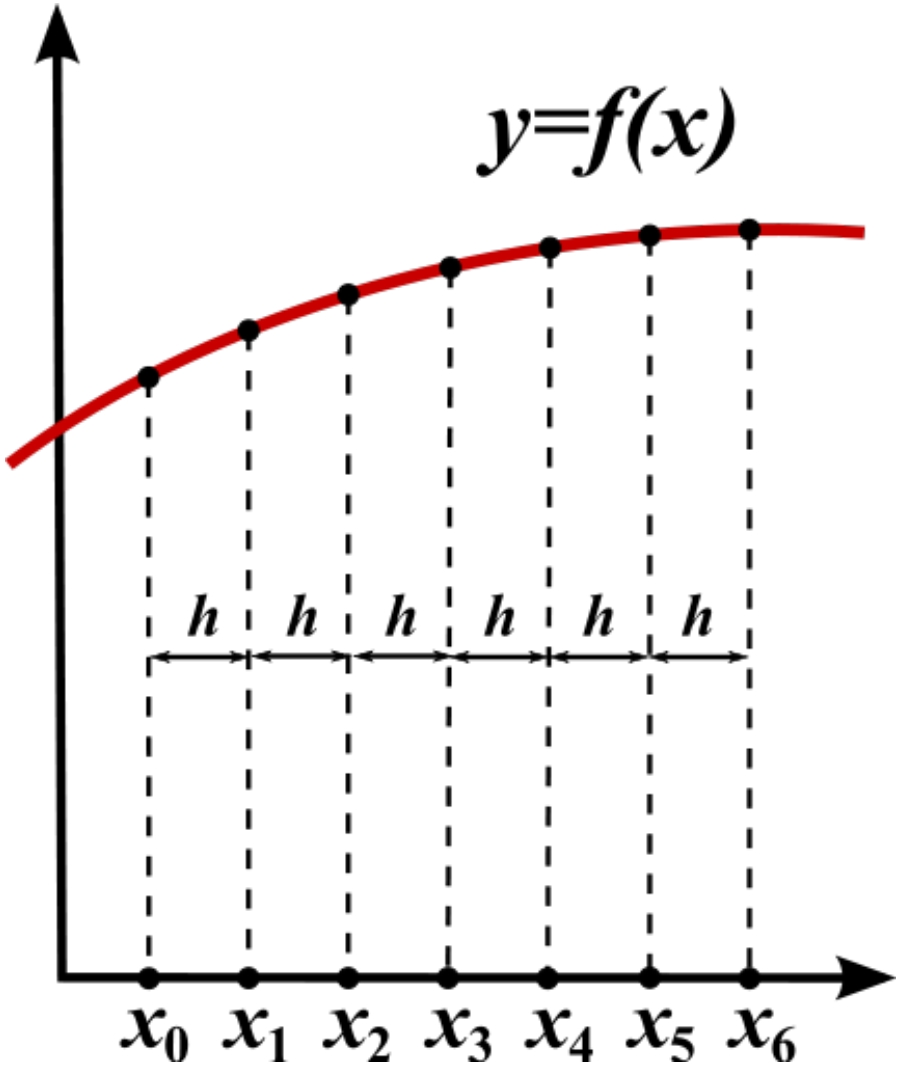
Calculate T at the next timestep using the explicit finite-difference time derivative

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

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

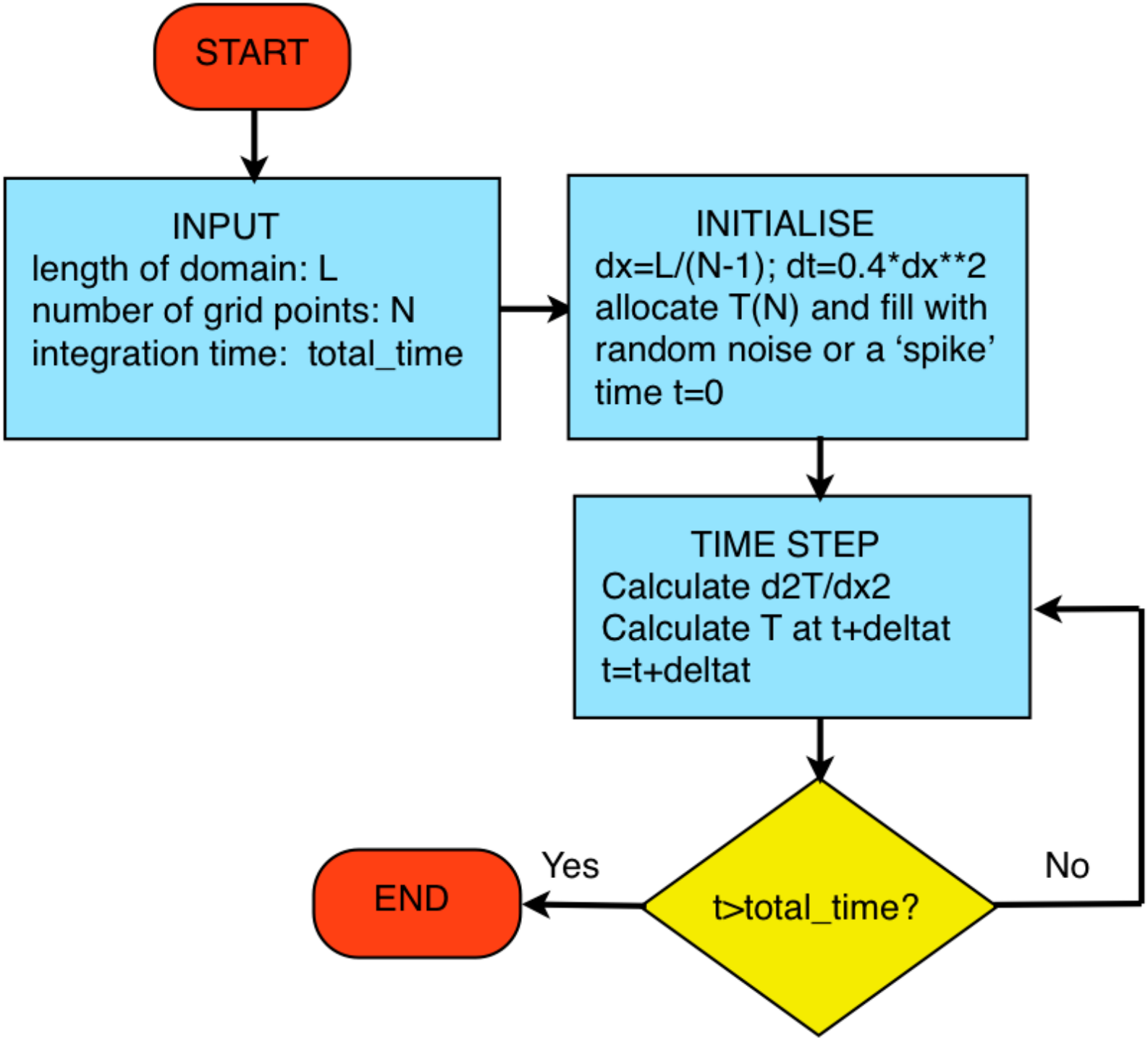
Where the 2nd x derivative is calculated in your module, using the equation from last week

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)

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


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

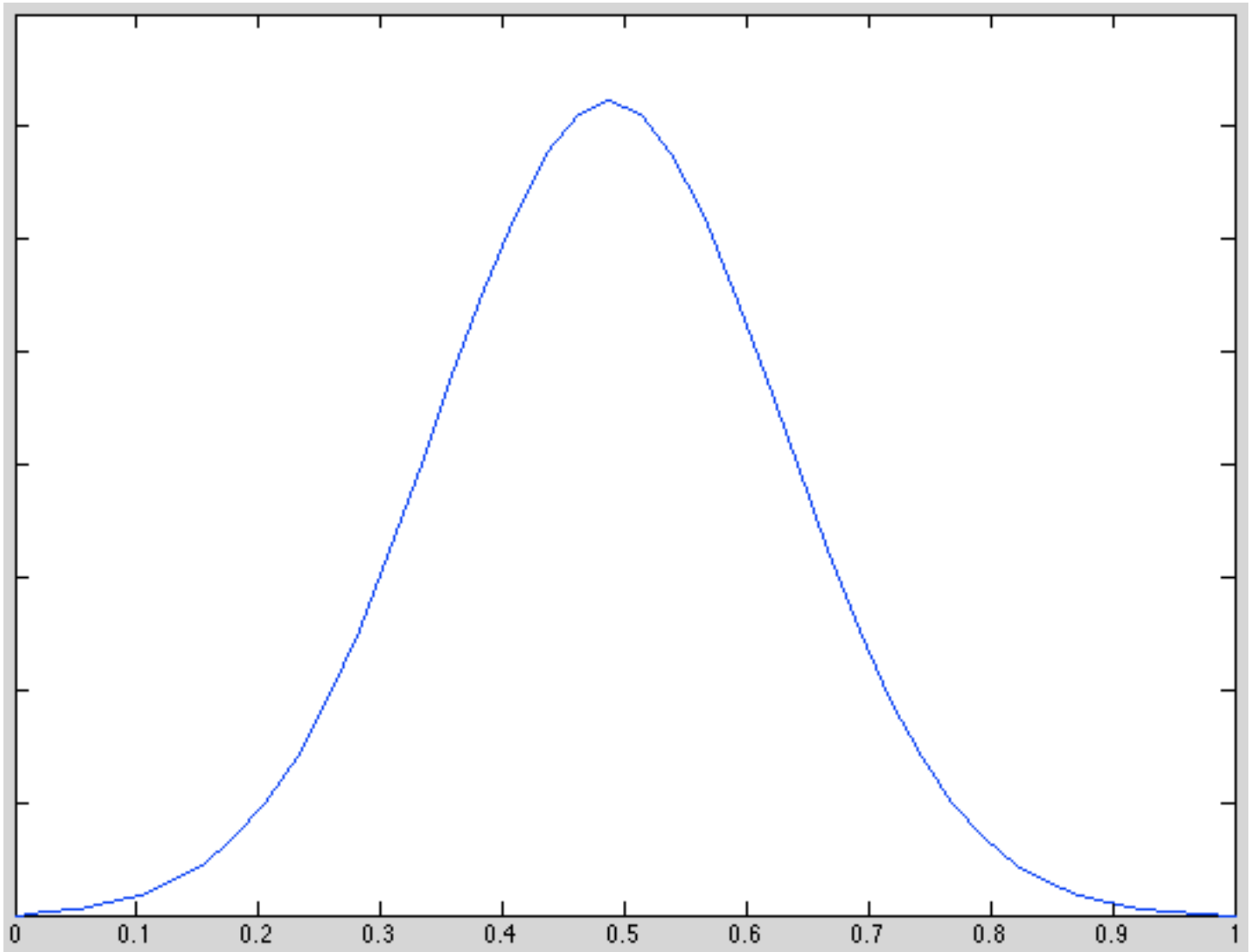
- Ask the user *L, N, total_time* (see flow chart)
- The user chooses to initialise the field with either random noise or a delta function=spike (i.e. 0 at all points except the central point, where it is 1.0). Write to an ascii file
- Take several time steps. For each timestep:
  - Calculate the second 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 (delta function) in centre



Thermal diffusion in 2D with constant diffusivity

$$\frac{\partial T}{\partial t} = \kappa\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) = \kappa\nabla^2 T$$

Where κ is the thermal diffusivity (m^2/s): $k = \rho C \kappa$

- Now discretise this using finite differences
  - nx points in *x*-direction, ny in y-direction, grid spacing=h
  - *e.g.*, $x_i = x_{min} + (i-1)h$
  - $T_{i,j}$ where i=1…nx, j=1…ny

Finite-difference ‘stencil’

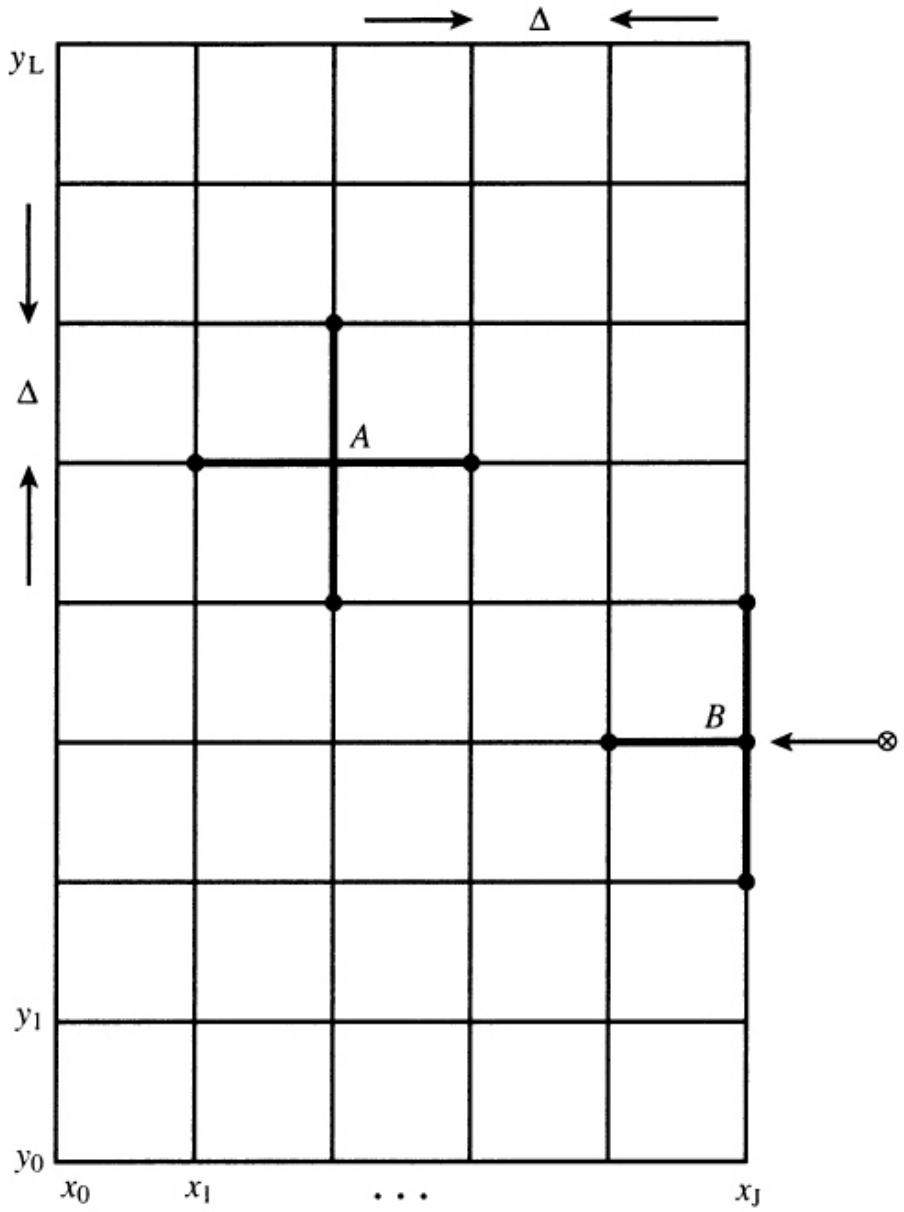


Figure 19.0.2. Finite-difference representation of a second-order elliptic equation on a two-dimensio
grid. The second derivatives at the point A are evaluated using the points to which A is shown connect
The second derivatives at point B are evaluated using the connected points and also using “right-ha
side” boundary information, shown schematically as ⊗.

Apply time-integration method

- Discretise time derivative
- "Explicit": like forward FD approximation

$$\frac{T_{i,j}^{(t_1+\Delta t)} - T_{i,j}^{(t_1)}}{\Delta t} = \kappa \left(\frac{T_{i-1,j}^{(t_1)} - 2T_{i,j}^{(t_1)} + T_{i+1,j}^{(t_1)}}{(\Delta x)^2} + \frac{T_{i,j-1}^{(t_1)} - 2T_{i,j}^{(t_1)} + T_{i,j+1}^{(t_1)}}{(\Delta y)^2} \right)$$

$$T_{i,j}^{(t_1+\Delta t)} = T_{i,j}^{(t_1)} + \Delta t \kappa \left(\frac{T_{i-1,j}^{(t_1)} - 2T_{i,j}^{(t_1)} + T_{i+1,j}^{(t_1)}}{(\Delta x)^2} + \frac{T_{i,j-1}^{(t_1)} - 2T_{i,j}^{(t_1)} + T_{i,j+1}^{(t_1)}}{(\Delta y)^2} \right)$$

- $T(t_2)$ appears only on left-hand side, so simple to program!

If Δx=Δy=h, simplifies to:

$$T_{i,j}^{(t_1+\Delta t)} = T_{i,j}^{(t_1)} + \Delta t \kappa \left(\frac{T_{i-1,j}^{(t_1)} + T_{i+1,j}^{(t_1)} + T_{i,j-1}^{(t_1)} + T_{i,j+1}^{(t_1)} - 4T_{i,j}^{(t_1)}}{h^2} \right)$$

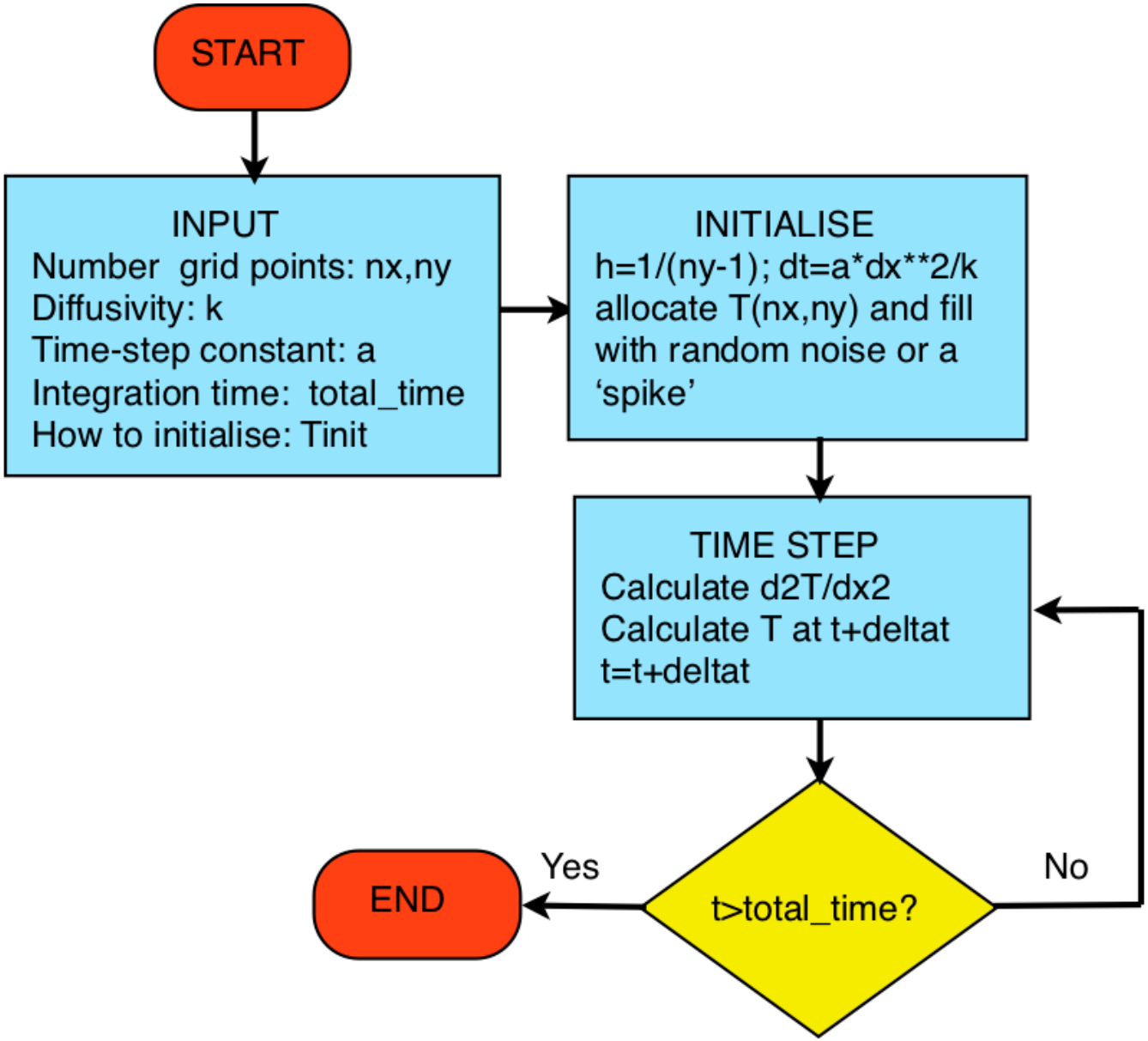
Store $T_{i,j}$ in an array T(i,j)

A note about time stepping

- The explicit method is **unstable** if the time step is too large. This means, you get oscillations and the amplitude grows exponentially with time.
- This depends on the grid spacing:

$$\Delta t_{critical} = a(\Delta x)^2 / \kappa$$

- where **a** is a constant (0.5 in 1D: you need to determine this for 2D)


START
INPUT
Number grid points: nx,ny
Diffusivity: k
Time-step constant: a
Integration time: total_time
How to initialise: Tinit
INITIALISE
h=1/(ny-1); dt=a*dx**2/k
allocate T(nx,ny) and fill
with random noise or a
'spike'
TIME STEP
Calculate d2T/dx2
Calculate T at t+deltat
t=t+deltat
t>total_time?
Yes
No
END

Structure of the diffusion program

- Read in control parameters using namelist:
  - number of grid points nx,ny
  - kappa. Start with kappa=1
  - total integration time total_time. Start with=0.1
  - Time step constant 'a'
- Set up variables and numerical details
  - T field: random or spike
  - dx=dy=1/(ny-1) (assume domain size =1 in y)
  - time step dt, as $\Delta t = a(\Delta x)^2 / \kappa$ where a is a constant - try between 0.1 and 1.0
  - number of time steps nsteps= total_time/dt
- Write a loop to perform nsteps steps, which
  - find 2nd derivative using module function or subroutine
  - update T field: T=T+dt*kappa*d2

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)

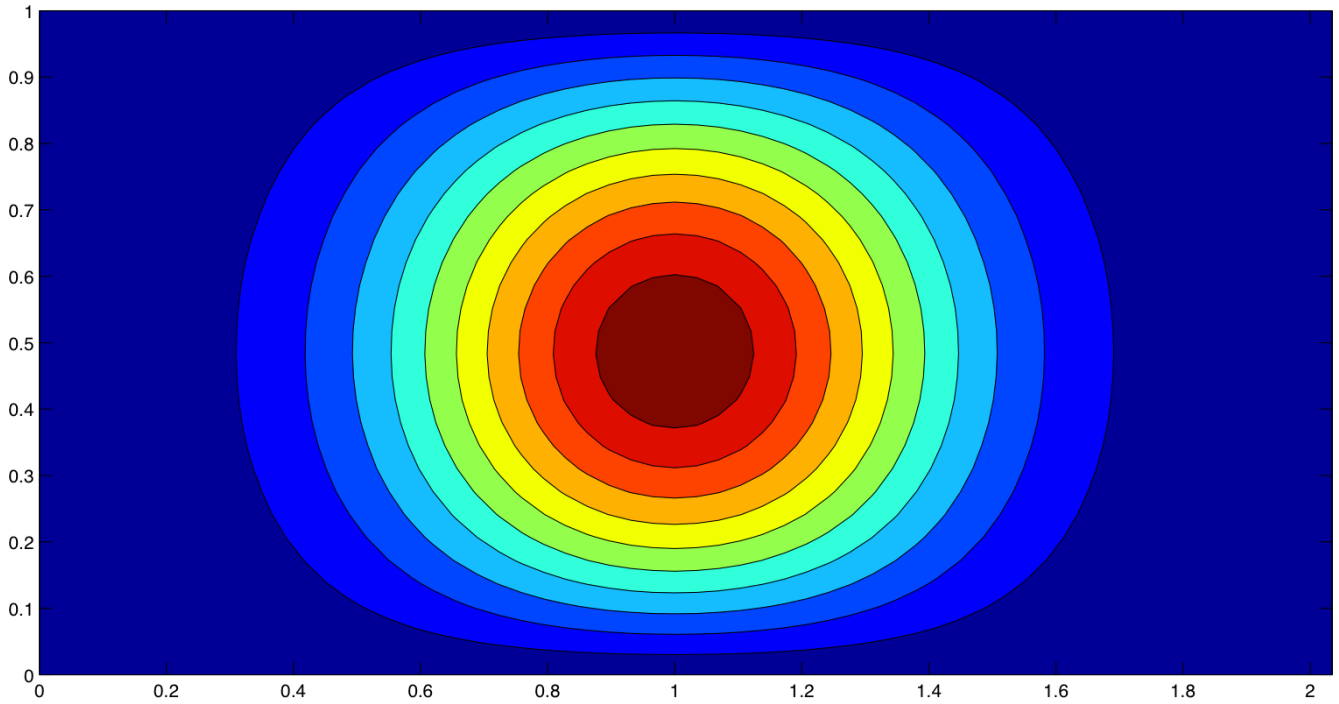
Test input file

```
&inputs
        nx=32
        ny=16
        k=1.0
        a=0.1
        total_time=0.05
        Tinit='spike'
/
```

Tests

1. Test your programs for two initial T distributions
   - (i) random (the solution should become smooth with time)
   - (ii) a spike, i.e., 0 everywhere except 1 in the centre cell (the solution should become a Gaussian).
2. Determine the critical value of ‘a’ above which the solution goes unstable, displaying oscillations that grow exponentially. Try different numbers of grid points: is critical ‘a’ the same?

Spike,
nx=2*ny
total_time=0.05



Writing a 2D array to a file, to be read by Matlab

```
open(1,file='T.dat',status='replace')
do j=1,ny
   write(1,'(1000(1pe13.5))') T(:,j)
end do
close(1)
```

To read into Matlab & plot:

load T.dat
contourf(T)

Hand in

- All .f90 files
- Figures showing the results of the 1-D and 2-D diffusion test cases (plotted using Matlab, Excel or another program of your choice)
- The critical ‘a’ value for 2-D diffusion