Numerical Modelling in FORTRAN day 4

Paul Tackley, 2014

Today’s Goals

各式各样的

1. Miscellaneous points
2. Review key points from reading homework
3. Precision, logical and complex types
4. More input/output stuff
5. Practice, practice!

Debugging your code: Useful compiler options

- gfortran

  -fbacktrace : reports line number of error

  -fbounds-check : checks bounds of arrays

  -ffpe-trap=invalid,zero,overflow,denormal : stops if fp error

  -g : allows use of debugger gdb

- ifort

  -traceback : reports line number of error

  uninitialised variables.

  -check : checks many things incl. bounds & uninit.

  -fpe0 : stops if floating point error

  -g : allows use of debugger idb (or gdb)

- Consult documentation (e.g. “man gfortran”)

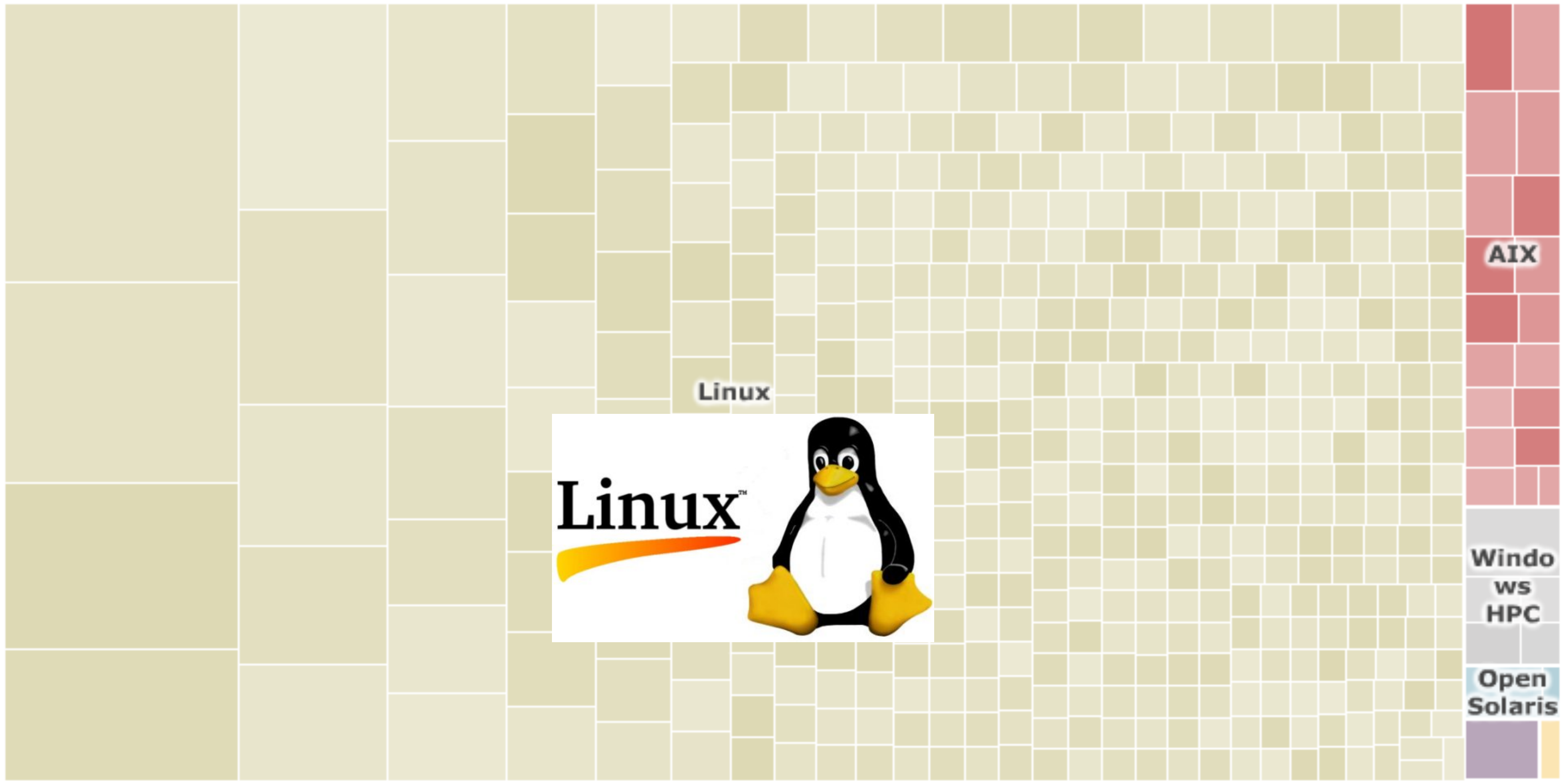
  Manu Pages of gfortran

Debugging your code: Useful compiler options

In Linux or Unix environment

- Example:
  - ➢ gfortran –ftraceback –fbounds-check -ffpe-trap=invalid,zero,overflow,denormal program.f90

Most powerful 500 computers in the world: What type of operating system?



Answer: Linux (unix)

Use of unix/linux enviroment

- Used on almost all ‘supercomputers’ (e.g. Rosa at CSCS, Brutus at ETH)
- MacOSX ‘terminal’ or ‘X11’ applications, Windows ‘cygwin’, Linux…
- Several online guides:

http://www.cs.brown.edu/courses/bridge/1998/res/UnixGuide.html
http://www.molvis.indiana.edu/app_guide/unix_commands.html
http://vic.gedris.org/Manual-ShellIntro/1.2/ShellIntro.pdf
http://www.ee.surrey.ac.uk/Teaching/Unix/

2. Review important points from online reading

- Module things (see examples online)
  - use *module*,only: *name1, name2*, ...
  - mainvariable1 => modvariable
  - public and private variables and functions
- Implied do loops: more variations
- Assumed size arrays
  - size() function ← Detect the array size
  - not necessary to pass array size as argument into subroutines, but lower bound info is lost

Fortran Precision

- a “bit” is 1 or 0; 8 bits = 1 byte (0 to 255)
- 32-bit (4 byte) numbers:
  - real: ~6 digits precision, range ~1e-38 to 1e38
  - integer: ±2147483647
- 64-bit (8 byte) numbers:
  - real: ~13 digits precision, range ~1d-308 to 1d308
- Fortran default precision is not defined: normally 32-bit, sometimes (e.g., on Crays) 64-bit.
- You can specify precision:

Specifying precision

- Method 1: in the code
  - see next slide for syntax
- Method 2: when compiling
  - use the appropriate flag, e.g.,
  - ifort -r8 program.f95   (makes reals 8-byte)
  - gfortran -fdefault-real-8 program.f95
- Integers and reals can be different sizes

i8 means integer 8-byte

precision syntax 句法

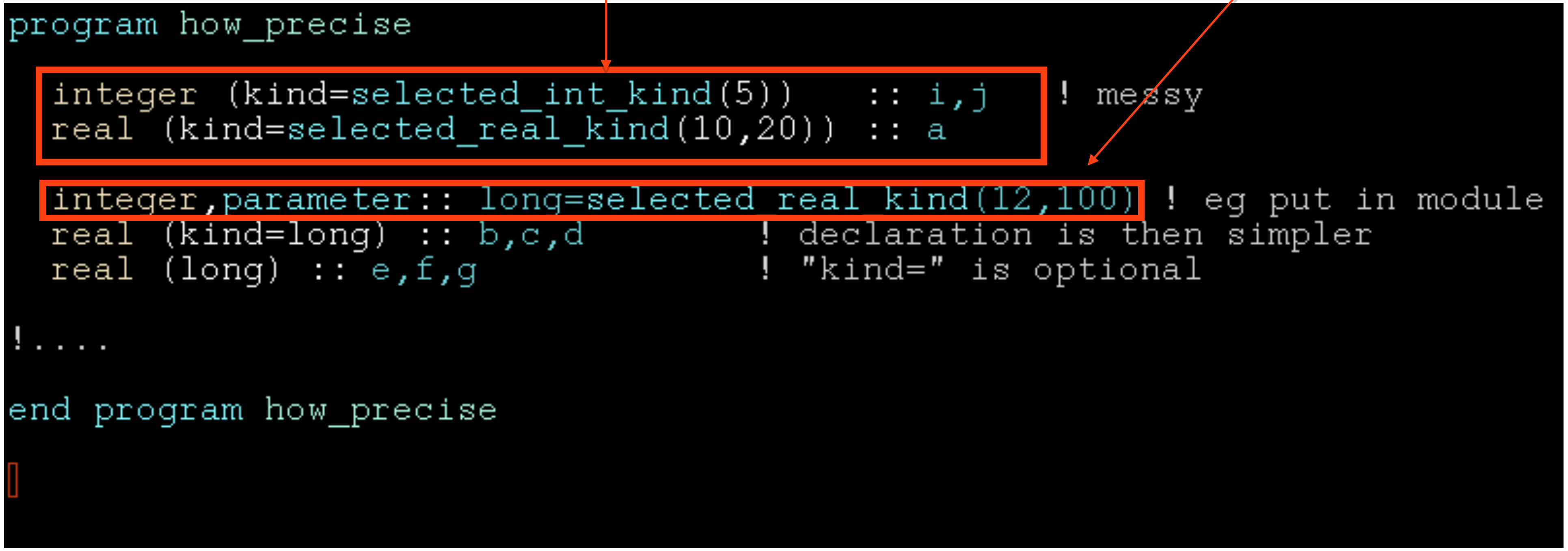
- F77:
  - real a ! normally 32-bit, might be 64
  - double precision a ! twice the above Double the default
  - real*4 a ! 4 byte (32 bit)
  - real*8 a ! 8 byte (64 bit)
- f95: use new functions ×10^n
  - selected_real_kind(#digits, max exp)
  - selected_int_kind(#digits)
  - examples see next slide

examples of these

至少12位数字，可以到10的100次方

integers at least with 5 digitals

Safe way



```
program how_precise

  integer (kind=selected_int_kind(5))    :: i,j   ! messy
  real (kind=selected_real_kind(10,20)) :: a

  integer,parameter:: long=selected_real_kind(12,100) ! eg put in module
  real (kind=long) :: b,c,d              ! declaration is then simpler
  real (long) :: e,f,g                   ! "kind=" is optional

!....

end program how_precise
```

What precision to use?

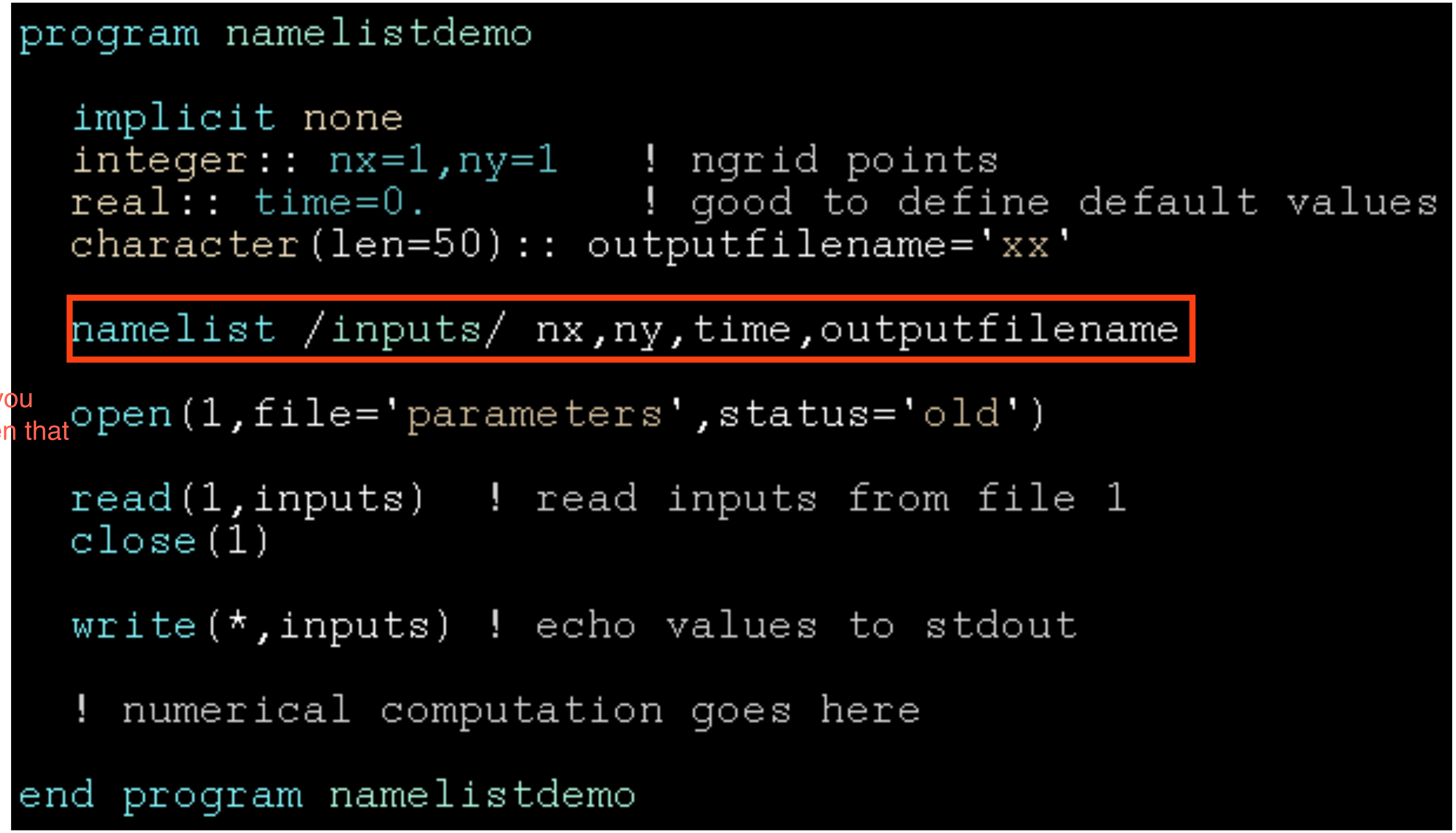
- If possible, use 32-bit
  - code runs faster
  - uses half as much memory
  - files use half as much disk space
- use 64-bit when >6-digit accuracy is needed in calculations, e.g.,
  - adding >millions of numbers
  - direct solvers (often)
- Try same run with each and see if you get the same answer!
- You can mix precision in same code

logical and complex variables

- **logical** variables typically use 32 bits even though 1 would be enough!
  - Values .true. or .false.
- **complex** variables consist of 2 numbers (real and imaginary parts) so take twice as much space as **real** variables
  - Related intrinsic functions cmplx(r,i), $\leftarrow$ to combine two real numbers to complex num
    aimag(c), real(c), conjg(c)
    get the imaginary part

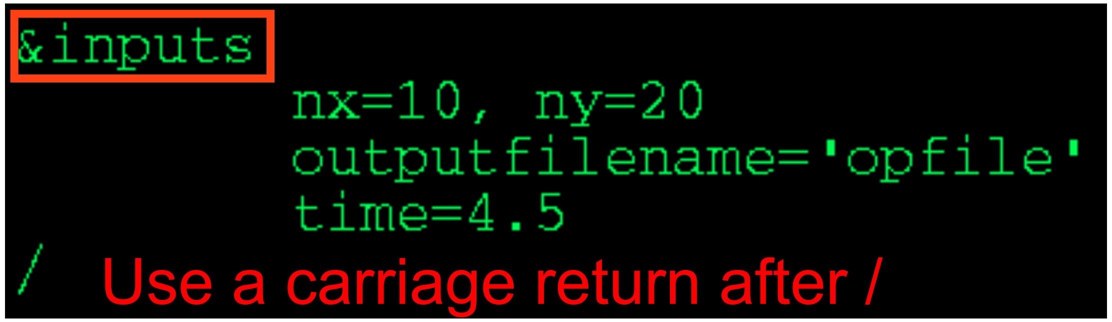
3. I/O: namelist input

- 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 $\boxed{\text{set defaults!}}$
- 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
```

to read, you must open that



```
&inputs
        nx=10, ny=20
        outputfilename='opfile'
        time=4.5
/
```

Use a carriage return after /

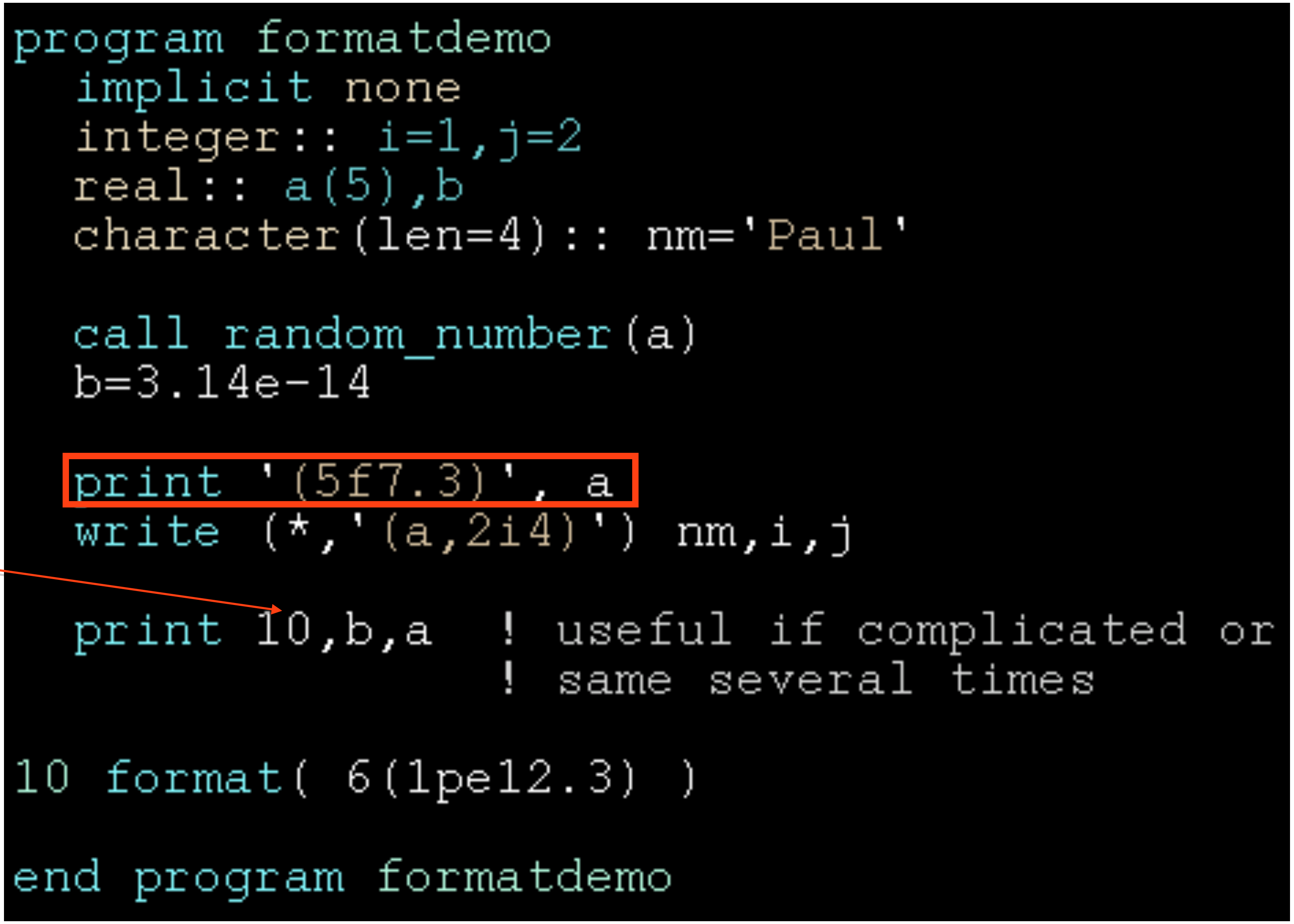
write(file name, type),
usually we use * as default

Specifying output format

- String: a10 means 10 characters
- Integer: i5 => integer 5 digits
- Floating: f10.4 => 10 characters, 4 after decimal point
- Exponential (e.g., 0.234e-12):
  - e14.4 means 14 characters, 4 after decimal point
  - Avoid the wasted leading 0 either by putting ‘1p’ first => 1pe14.4 or (f90-) using ‘es‘, e.g. es14.4.

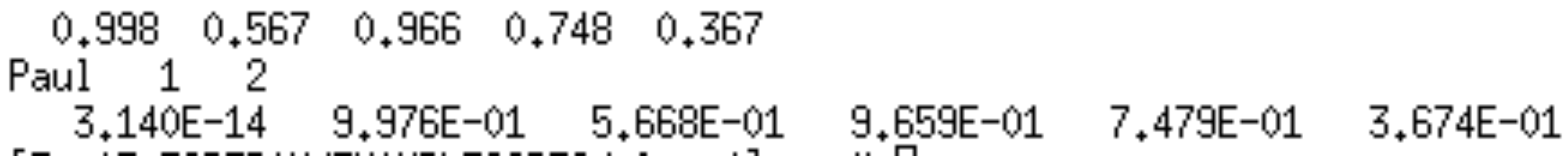
3i5 means 3 integers and 5

- e.g., (3i5,a10,f10.4,3es14.4))
- This can go directly in a print or write statement, or in a format line with a number (see next)



```
program formatdemo
   implicit none
   integer:: i=1,j=2
   real:: a(5),b
   character(len=4):: nm='Paul'

   call random_number(a)
   b=3.14e-14

   print '(5f7.3)', a
   write (*,'(a,2i4)') nm,i,j

   print 10,b,a    ! useful if complicated or
                   ! same several times

10 format( 6(1pe12.3) )

end program formatdemo
```

读取第十行的声明



```
  0.998  0.567  0.966  0.748  0.367
Paul   1   2
   3.140E-14   9.976E-01   5.668E-01   9.659E-01   7.479E-01   3.674E-01
```

Finite difference approximation: Forward, Backward, Centered

- Forward $$f'(x) = \frac{f(x+\delta x) - f(x)}{\delta x}$$
- Backward $$f'(x) = \frac{f(x) - f(x-\delta x)}{\delta x}$$
- Centered (BEST) $$f'(x) = \frac{f(x+\delta x) - f(x-\delta x)}{2\delta x}$$

FBC accuracy

- e.g., $f(x)=3.6+1.9x-2.7x^4$
- Compute $f'(2.5)$ with dx=0.1: (=-166.85)
  - Forward f.d. approx.=-177.24 (error 10.39)
  - Backward f.d. approx=-157.00 (error -9.85)
  - Centered f.d. approx.=167.10 (error 0.25)
- dx too small => numerical precision errors!
  (keep dx/x≥10^{-3} s.p. 10^{-6} d.p.)

出现精确性错误，所以dx不要选的太小

FBC accuracy (2)

- Finite-difference derivative program on next slide demonstrates
  - Centered differencing more accurate
  - Accuracy improves with decreasing grid spacing until truncation error becomes problem
  - Increase to 64-bit precision to reduce truncation error!

```fortran
program Deriv1
  implicit none
  integer          :: n,i
  real,allocatable:: y(:),dydxL(:),dydxC(:),dydxR(:)
  real             :: x,dx

  write(*,'(a,$)') 'Input number of grid points:'; read*,n
  allocate (y(n),dydxL(n),dydxC(n),dydxR(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 derivativeLCR (y,dx,dydxL,dydxC,dydxR)  ! calculate dydx

  do i = 2,n-1  ! write result, -sin(x) and error
     x = (i-1)*dx
     print*,-sin(x),-sin(x)-dydxL(i),-sin(x)-dydxC(i),-sin(x)-dydxR(i)
  end do

  deallocate(y,dydxL,dydxC,dydxR)    ! finish

contains

  subroutine derivativeLCR (a,h,apB,apC,apF)
    real    ,intent(in) :: a(:),h
    real    ,intent(out),dimension(size(a)):: apB,apC,apF
    integer             :: np,i              ! local variable

    np = size(a)
    apB=0.;apC=0.;apF=0.
    do i = 2,np-1
       apB(i) = (a(i  )-a(i-1))/h     ! Backward
       apC(i) = (a(i+1)-a(i-1))/(2*h) ! Centered
       apF(i) = (a(i+1)-a(i  ))/h     ! Forward
    end do

  end subroutine derivativeLCR

end program Deriv1
```

Exercises

1. Make your 1D second derivative function into a function that (i) returns an array and (ii) does not need the number of points as an argument, and test using the same main program (with one less argument)
2. Write a module to calculate del-squared of a 2D field, assuming equal grid spacing in each direction (but possibly unequal number of points)
3. Use this in a 2D diffusion program. The program should read parameters from a file using namelist, and write the intial and final states to ascii files (specifying an appropriate format) that can be read with a plotting program such as Matlab. Do various tests. Details on next slides.
4. Read ‘formatted input and output’ on http://www.cs.mtu.edu/%7eshene/COURSES/cs201/NOTES/format.html
5. Hand in (i) your fortran files (ii) plots of initial and final T fields for a *stable* case, (iii) the critical “a” value

Thermal diffusion in 2D with constant coefficient

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

Where kappa 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=~~D~~ Should be H
- *e.g.*, $x_i = x_{min} + (i-1)h$
- $T_{i,j}$ where i=1…nx, j=1…ny

Finite-difference ‘stencil’

Assuming Temperature is 0 at the boundaries

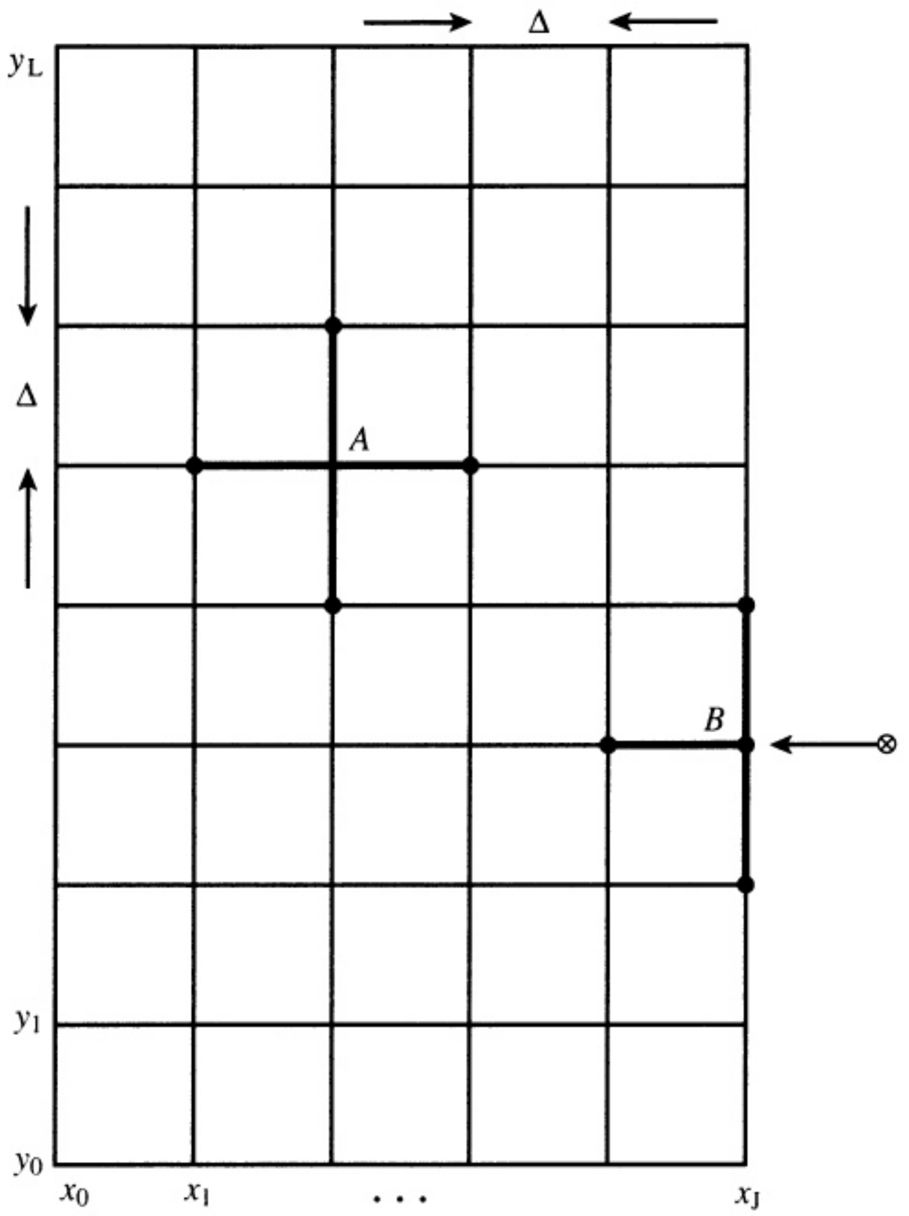


Figure 19.0.2. Finite-difference representation of a second-order elliptic equation on a two-dimensio grid. The second derivatives at the point A are evaluated using the points to which A is shown connect The second derivatives at point B are evaluated using the connected points and also using “right-ha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

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=Δz=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)

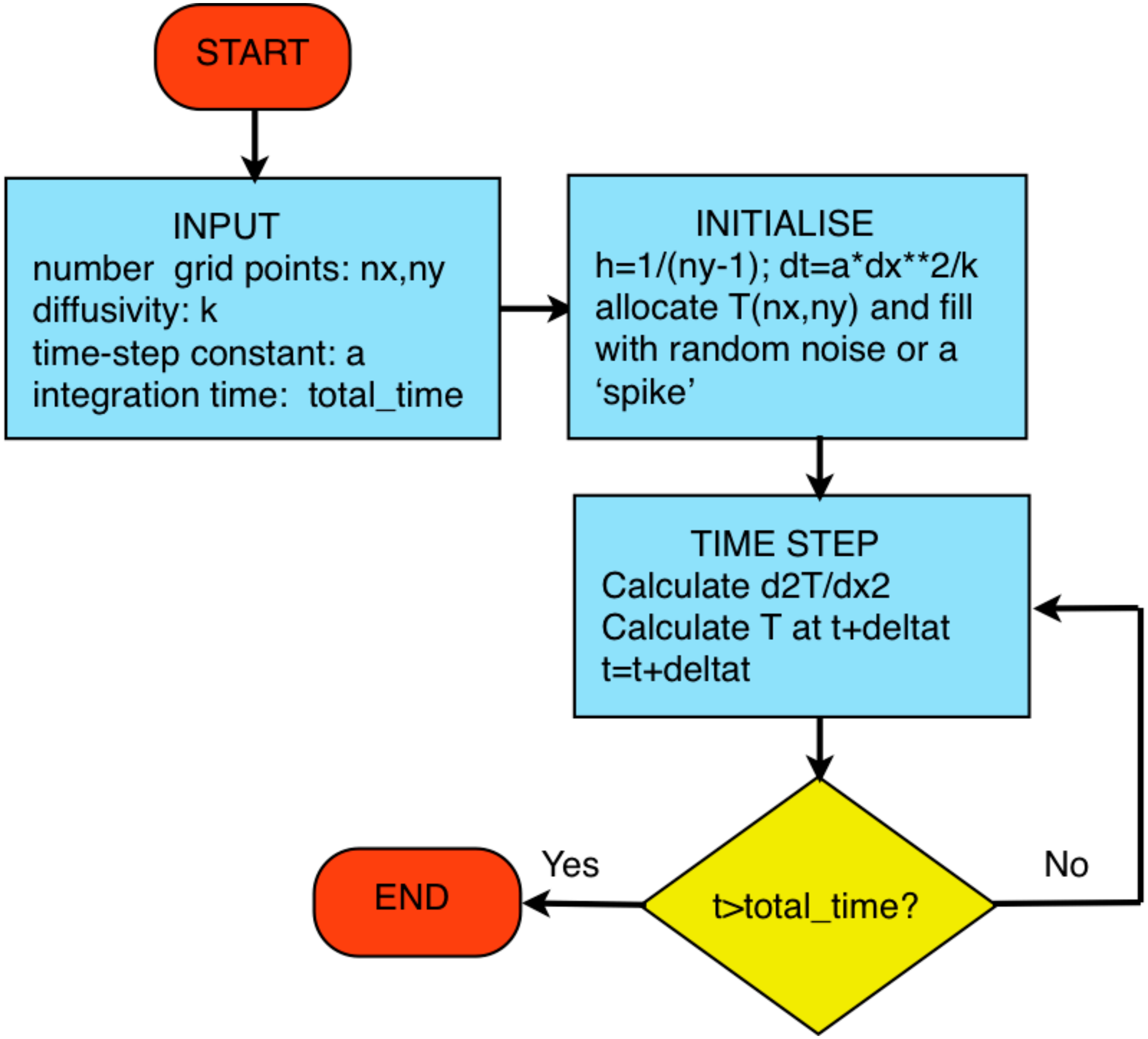
A note about time stepping

- 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:

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

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

namelist


START
INPUT
number grid points: nx,ny
diffusivity: k
time-step constant: a
integration time: total_time
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
- Set up variables and numerical details
  - T field: random or spike
  - grid spacing=1/(ny-1) (assumes 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 d2 using the module function
  - update T field: T=T+dt*kappa*d2

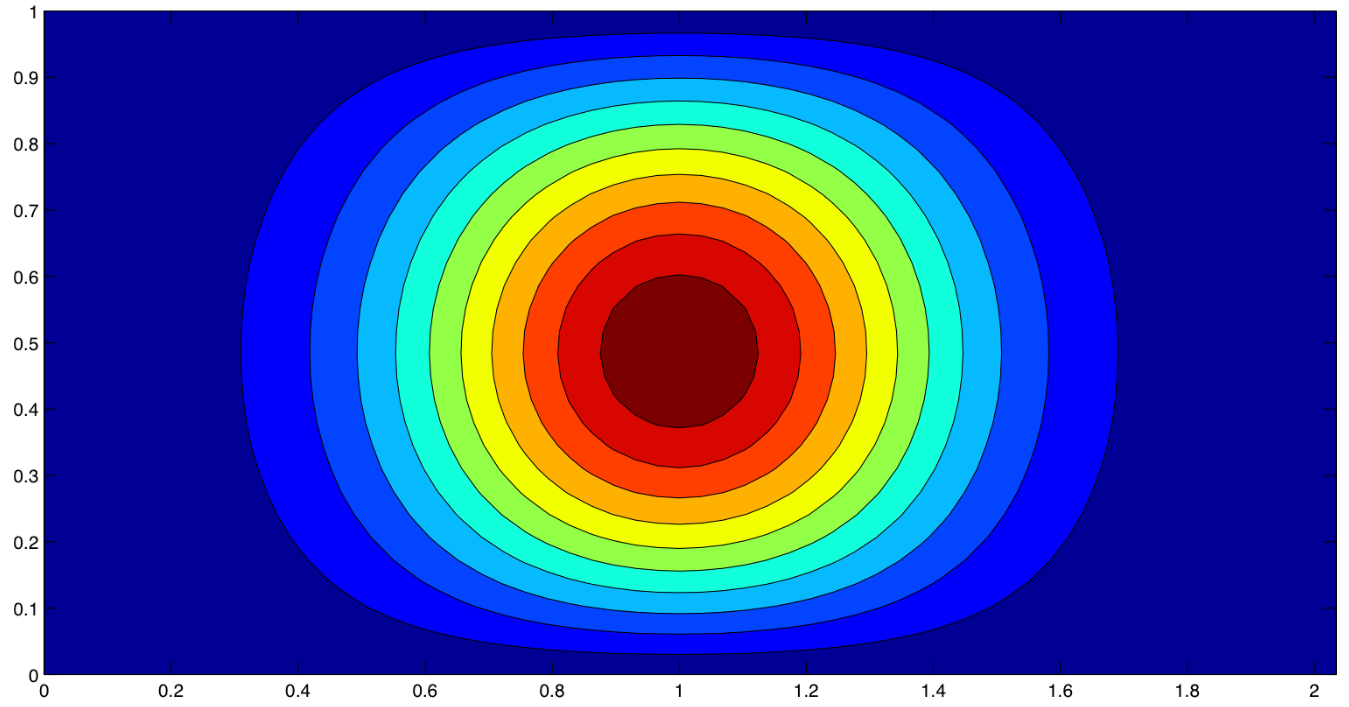
Boundary conditions

- To begin with, 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)

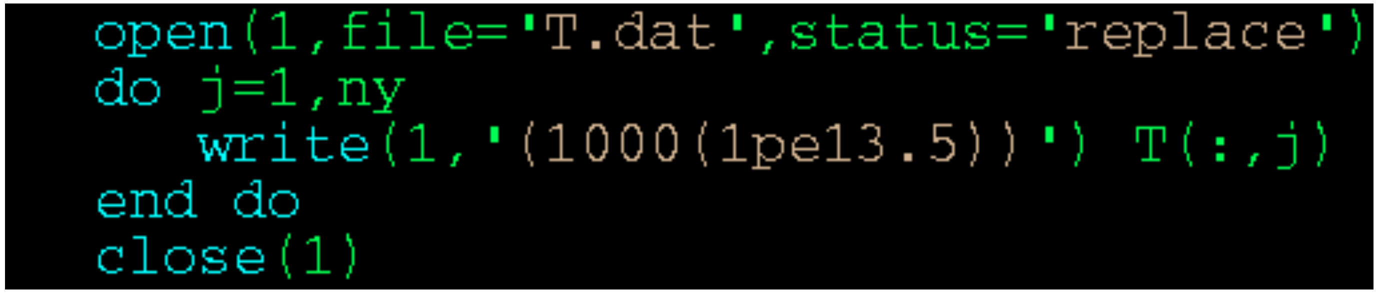
Tests

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 it the same?.

Spike,
nx=2*ny
total_time=0.05



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



```fortran
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)
```