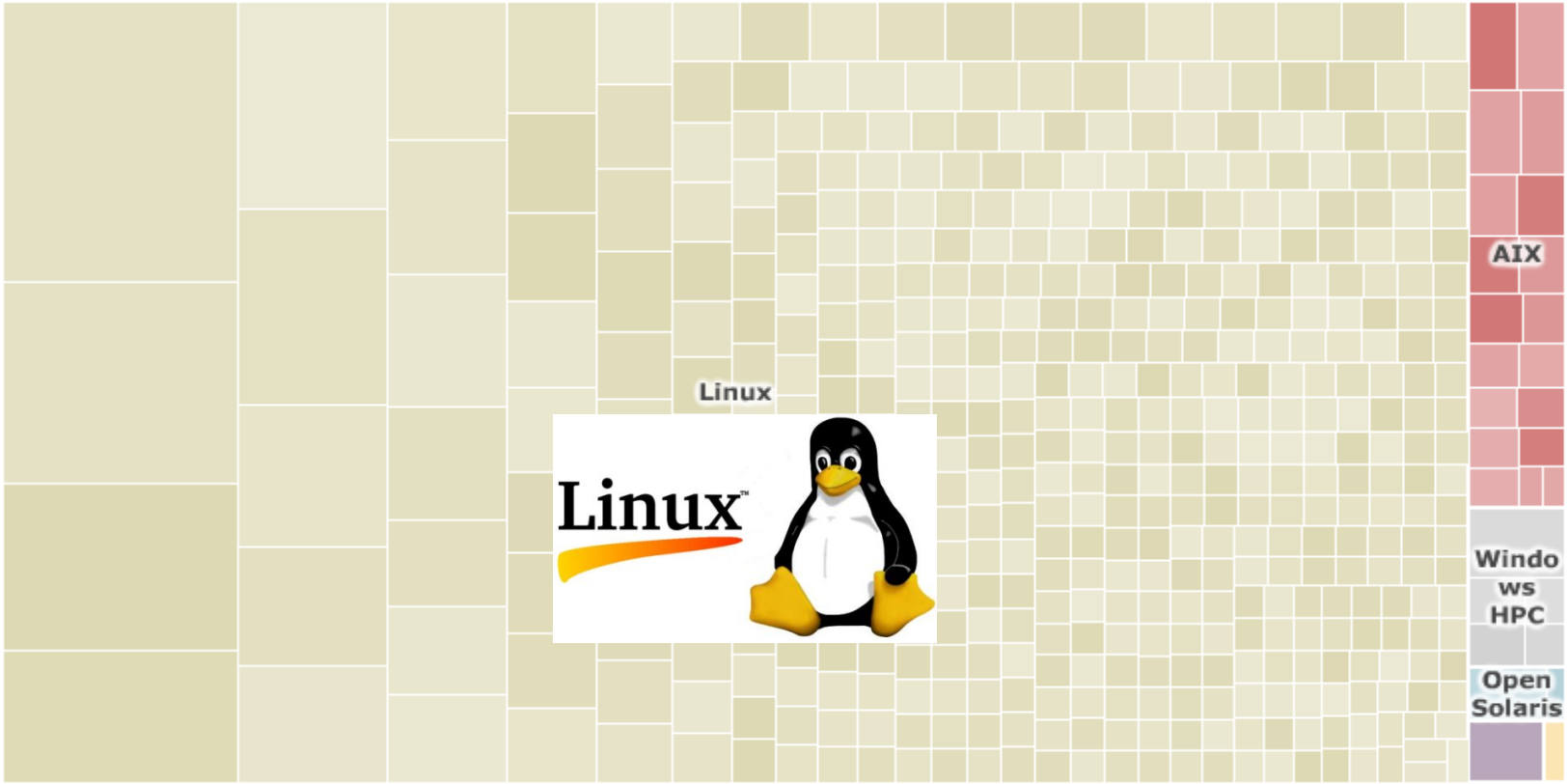
Numerical Modelling in FORTRAN day 4

Paul Tackley, 2020

Today's Goals

1. Linux environments
2. Review key points from reading homework
3. Precision
4. Formatted input/output
5. Practice: 2-D advection-diffusion equation

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. Piz Daint @ CSCS, Euler/Leonhard @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
  - not necessary to pass array size as argument into subroutines, but lower bound info is lost

3. 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
- Sometimes 80-bit (“extended” precision) and/or 128-bit (“quad” precision) are also available
- Fortran default precision is not defined: normally 32-bit, sometimes (e.g., on Crays) 64-bit.
- Therefore best to 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
  - -ifort –r8 –i4 program.f95

precision syntax

- F77 and later:
  - real a ! normally 32-bit, might be 64
  - double precision a ! twice the above
  - real*4 a ! 4 byte (32 bit)
  - real*8 a ! 8 byte (64 bit)
- f95: use new functions
  - selected_real_kind(#digits, max exp)
  - selected_int_kind(#digits)
- with
  - Real(kind=n):: …
  - Integer(kind=n):: …

precision syntax: constants

- Constants will normally be 32-bit by default; to get more accurate 64-bit constants either
  - Change the default precision (e.g. ifort –r8)
  - Use “D” instead of ”E”, e.g. 1.234D5
  - Attach “_8” to the number, e.g. 1.234_8
- Similarly with 128-bit “Quad” precision
  - 1.234Q5
  - 1.234_16

examples of these

```
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
```

Precisions available in gfortran

```
program test_precision
  implicit none
  real*4  a4
  real*8  a8
  real*10 a10
  real*16 a16

  print*,' 4 bytes:',precision(a4),tiny(a4),huge(a4)
  print*,' 8 bytes:',precision(a8),tiny(a8),huge(a8)
  print*,'10 bytes:',precision(a10),tiny(a10),huge(a10)
  print*,'16 bytes:',precision(a16),tiny(a16),huge(a16)

end program test_precision
```

```
 4 bytes:           6   1.17549435E-38   3.40282347E+38
 8 bytes:          15   2.2250738585072014E-308   1.7976931348623157E+308
10 bytes:          18   3.362103143112093506E-4932   1.189731495357231765E+4932
16 bytes:          33   3.362103143112093506262677817321752602E-4932   1.189731495357231765085759326628007016E+4932
```

Beware

- It is *not safe to assume* that the kind is equal to the number of bytes,
  - e.g. real(4), real(8) – cannot assume that 4 & 8 are the number of bytes
- Always use selected_real_kind, e.g.
  - dp=selected_real_kind(10,300)
  - real(dp):: a,b,c

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), aimag(c), real(c), conjg(c)

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
- ‘g’ gives ‘f’ for small numbers and ‘e’ for large numbers
- 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
```

```
  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 $\delta x=0.1$: (=-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)
- δx too small => numerical precision errors! (keep $\delta x/x \geq 10^{-3}$ s.p. 10^{-6} d.p.)

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!

```
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. Read ‘formatted input and output’ on http://www.cs.mtu.edu/%7eshene/COURSES/cs201/NOTES/format.html
2. Program 2-dimensional advection-diffusion, as detailed on the following slides

Today solve advection-diffusion equation for a fixed velocity field v

$$\frac{\partial T}{\partial t} + \vec{v} \cdot \nabla T = \kappa \nabla^2 T$$

For incompressible flow

$$\nabla \cdot \vec{v} = 0$$

A note on numerical advection

'pure' advection: $$\frac{\partial T}{\partial t} = -\vec{v} \cdot \nabla T$$

Is very difficult to treat accurately, as will be demonstrated in class for 1-dimensional advection with a constant velocity.

- Simple-minded schemes either go unstable or smear out temperature anomalies (numerical diffusion).
- More sophisticated schemes can cause artificial ripples (numerical dispersion) and other types of distortion.
- Many papers have been written on numerical advection!

Next, demonstration of why we need upwind advection

- Simple 1D Matlab script shows how centered or downwind schemes go unstable very quickly!
- Upwind method can be thought of linear interpolation to where the material is coming from
- Higher-order versions are used for research codes

Timestepping notes

- UPWIND finite differences for the advection (v.gradT, or v*dT/dx) terms take the forward or backward derivative **in the direction that material is coming from**
  - If vx>0, vx_dTdx = vx*(T(i)-T(i-1))/dx
  - If vx<0, vx_dTdx = vx*(T(i+1)-T(i))/dx
- As with diffusion **the advection timestep dt is limited by the time it takes for material to move 1 grid spacing**. You need to calculate both the advective and diffusive timesteps and take the minimum.
  - In 2-D, material should advect no more than ~0.6 of a grid spacing

2D Velocity can be represented by a scalar streamfunction

$$v_x = \frac{\partial \psi}{\partial y} \qquad v_y = -\frac{\partial \psi}{\partial x}$$

- automatically satisfies incompressibility

Advantages of using the streamfunction

- Two vector velocity components are reduced to one scalar
- Continuity is automatically satisfied
- If also solving the Navier-Stokes equation, pressure can be algebraically eliminated from the momentum equation, reducing the number of variables further

Today's exercise: fixed flow field

(i) Calculate velocity at each point using centered finite differences

$$\left(v_x, v_y\right) = \left(\frac{\partial \psi}{\partial y}, -\frac{\partial \psi}{\partial x}\right)$$

(ii) Take timesteps to integrate the advection-diffusion equation for the specified length of time using UPWIND finite-differences for dT/dx and dT/dy (and centered for del2(T))

$$\frac{\partial T}{\partial t} = -v_x \frac{\partial T}{\partial x} - v_y \frac{\partial T}{\partial y} + \nabla^2 T$$

Finite difference version

$$T_{i,j}^{(t_1+\Delta t)} = T_{i,j}^{(t_1)} + \Delta t\left[\kappa\left(\nabla^2 T\right)_{i,j}^{(t_1)} - \left(\vec{v}\bullet\nabla T\right)_{i,j}^{(t_1)}\right]$$

where

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

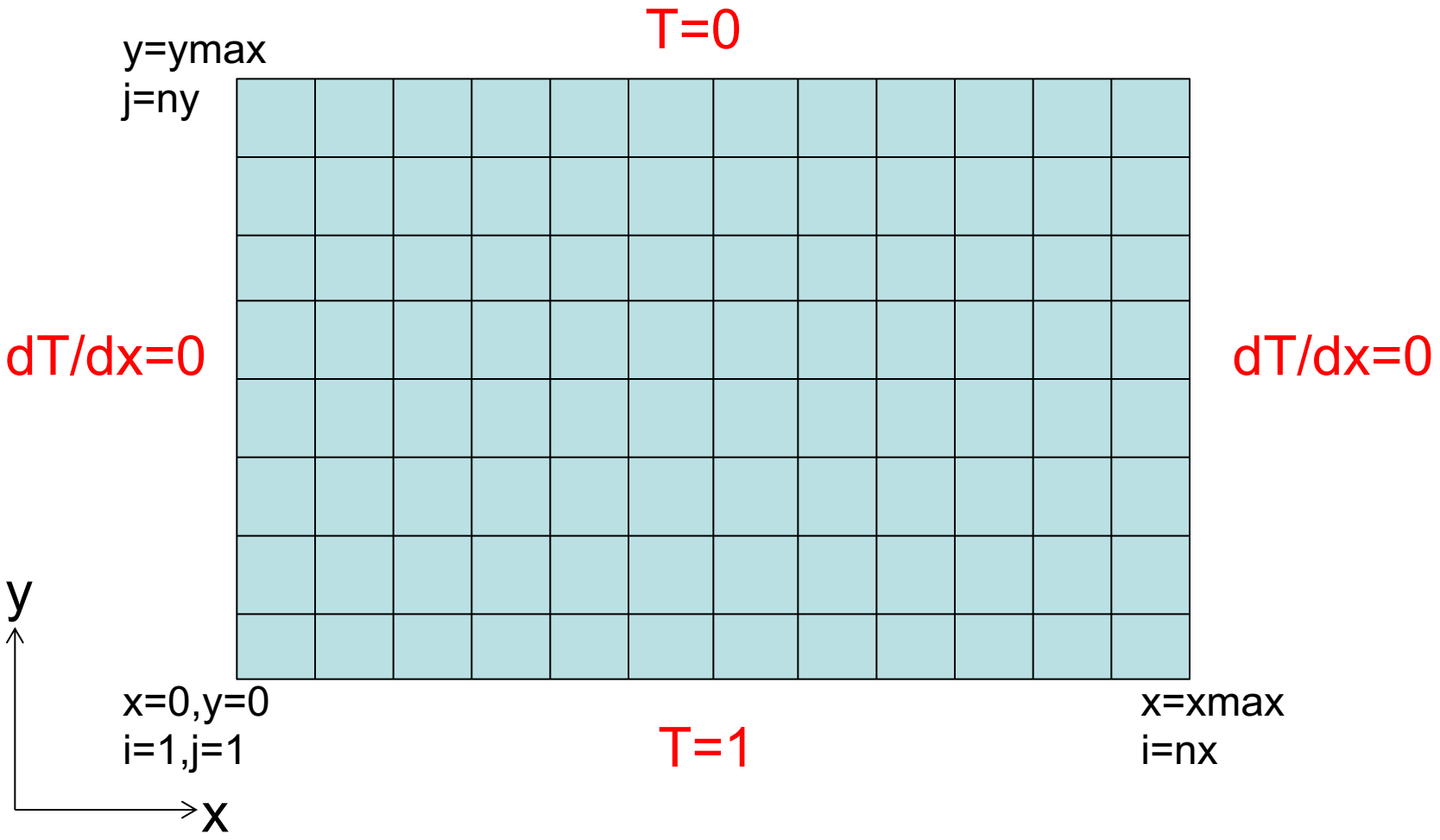
$$\left(\vec{v}\bullet\nabla T\right)_{i,j}^{(t_1)} = vx_{i,j}\left(\frac{\partial T}{\partial x}\right)_{i,j} + vy_{i,j}\left(\frac{\partial T}{\partial y}\right)_{i,j}$$

Upwind derivatives

$$\Delta t = \min\left[a_{diff}\frac{\left(\min(\Delta x,\Delta y)\right)^2}{\kappa}, a_{adv}\min\left(\frac{\Delta x}{vx_{\max}}, \frac{\Delta y}{vy_{\max}}\right)\right]$$

(a_{adv}~0.6)

Grid and boundary conditions



Physical problem

- Temperature boundary conditions
  - T=1 at bottom (y=0)
  - T=0 at top (y=1)
  - dT/dx=0 at sides (i.e., zero flux)
- Stream function: try a simple cellular flow, with different values of the constant B

$$\psi = B \sin(\pi x / x_{\max}) \sin(\pi y / y_{\max})$$

Note that this has a fixed value (0) at the boundaries, making them impermeable

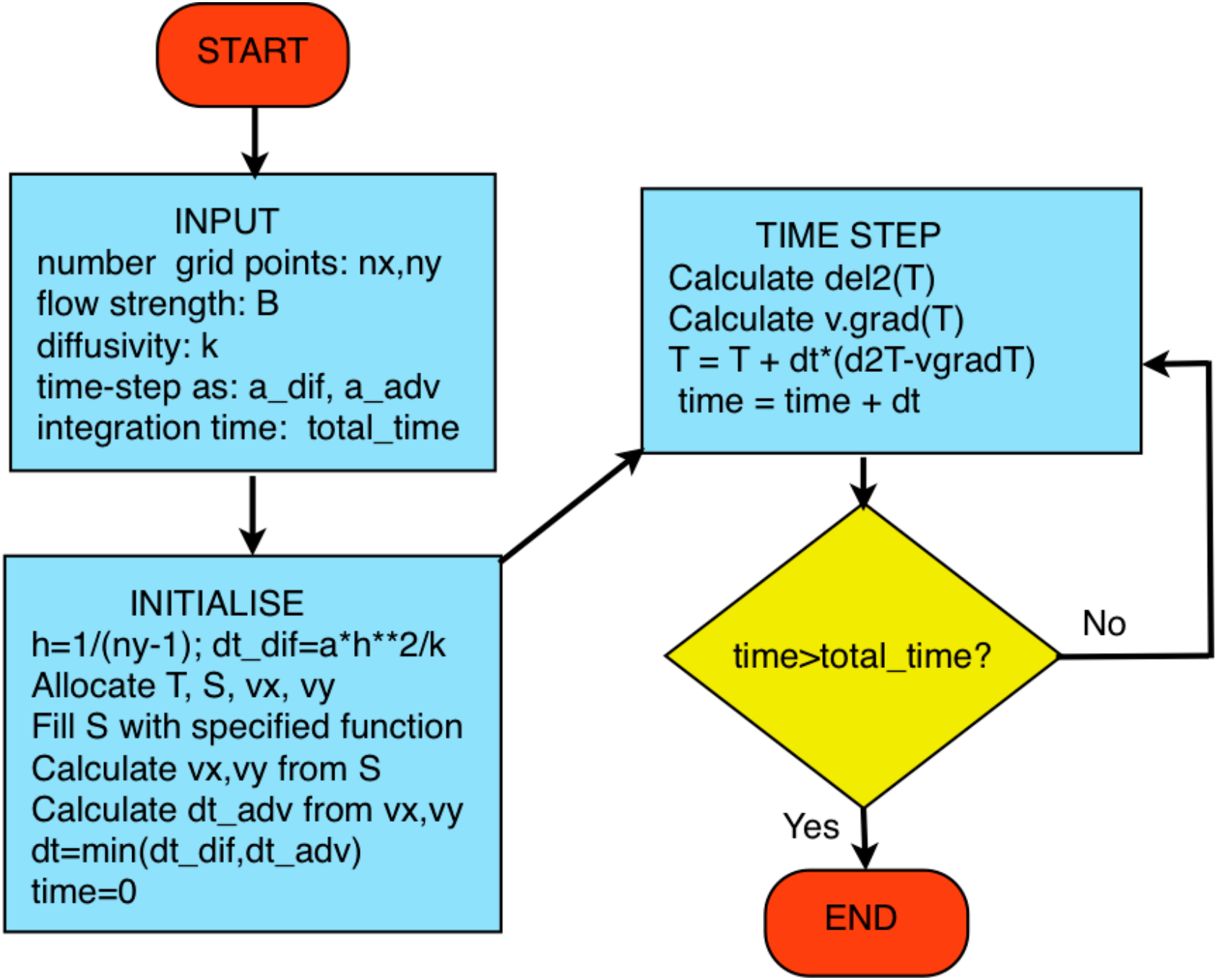
Treatment of boundary conditions

Sides: $\frac{dT}{dx} = 0$

Easiest : Solve advection & diffusion on points 2…nx-1.
After each step, set T(1,:)=T(2,:) and T(nx)=T(nx-1)

Top: $T = 0$ Bottom: $T = 1$

Set T(:,1)=1. and T(:,ny)=0.


START
INPUT
number grid points: nx,ny
flow strength: B
diffusivity: k
time-step as: a_dif, a_adv
integration time: total_time
INITIALISE
h=1/(ny-1); dt_dif=a*h**2/k
Allocate T, S, vx, vy
Fill S with specified function
Calculate vx,vy from S
Calculate dt_adv from vx,vy
dt=min(dt_dif,dt_adv)
time=0
TIME STEP
Calculate del2(T)
Calculate v.grad(T)
T = T + dt*(d2T-vgradT)
time = time + dt
time>total_time?
No
Yes
END

Program structure

- Write new subroutines to
  - calculate vel. (vx,vy) from streamfunction S
  - calculate v.grad(T)
- Put the new subroutines with the old del2 subroutine into a module
- (for bonus credit). Write the various routines (del2, velocity calculation, gradT) as array functions, as in Class 3 slides 33-38.

Program structure (2)

- Read in parameters using namelist. The parameters are as last week with the addition of B, the strength of the flow and a_adv.
- Initialise S and calculate V using your subroutine or function.
  - Use V to calculate the maximum advection timestep.

Standard parameter file

```
&inputs
        nx=100,  ny=100
        B=100.
        k=1.0
        a_adv=0.4,  a_dif=0.2
        total_time=10.0
 /
```

Your code should work with this file, named AdvDif_parameters.dat, as the input file.

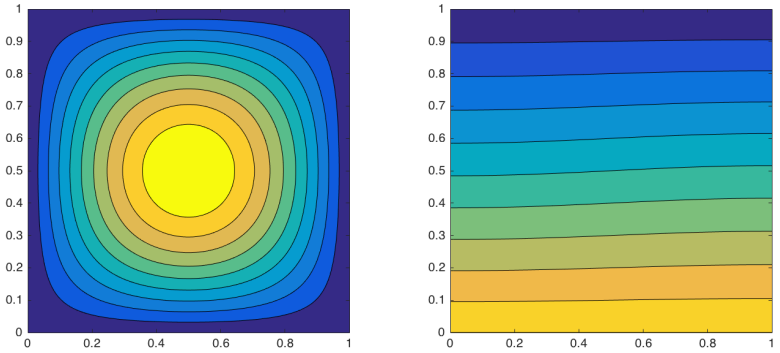
Obtaining results

- Take timesteps and write and plot the results
- After a sufficient integration time the system should reach a steady-state, i.e. it doesn't change any more with time
- The initial condition will not matter if you run it to steady state
- Email your Fortran files and plots for 3 different B values: 1,10 and 100

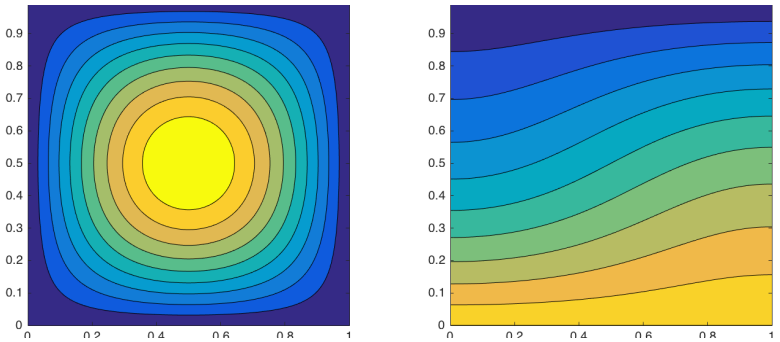
Solutions

- k=1
- depth=width=1
- steady-state (long integration time)
- Stream function left, Temperature right

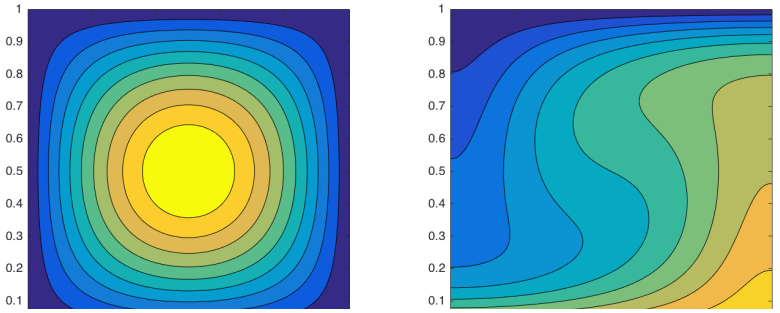
B=0.1



B=1



B=10



B=100