Numerical Modelling in FORTRAN day 5

Paul Tackley, 2014

Today's Goals

1. A few useful things
2. Learn derived variable **types**
3. Learn several other new things: binary I/O, save, forall, where, character manipulation, keyword arguments and optional arguments.
4. Practice, practice! This time solving advection-diffusion equation.

Array manipulation in f90-

- Like MATLAB, can operate on whole arrays with one statement, e.g.

```
T = T + dt*d2T
```

is the same as:

```
do j=1,ny; do i=1,nx
   T(i,j)=T(i,j)+dt*d2T(i,j)
end do; end do
```

- Built-in array functions: sum(array), product(array), matmul(matrix1,matrix2), dot_product(vec1,vec2)

Multiply all elements

矩阵的乘法

new type containing several already-defined types

array of new type

address parts using %

用%调用新的类里的元素

```
program typedemo1

  implicit none

  type person
     character(len=20):: name
     integer :: birthyear
     integer,allocatable:: childyear(:)
  end type person

  type(person):: beatle(4)

  beatle(1)%name          ="John"
  beatle(1)%birthyear     =1940
  allocate(beatle(1)%childyear(2))
  beatle(1)%childyear(1)=1963
  beatle(1)%childyear(2)=1975

  beatle(2)%name          ="Paul"
  ! etc...

  print*,beatle(1)%name, &
         beatle(1)%birthyear, &
         beatle(1)%childyear

end program typedemo1
```

A type to contain several different variables at each grid point

```
program typesdemo2
  implicit none

  ! Using this structure will likely result in
  !   slow-running code
  type gridpoint
     real :: temperature, composition, velocity(3)
  end type gridpoint

  type(gridpoint),allocatable:: grid(:,:)
  integer nx,ny

  read*,nx,ny
  allocate(grid(nx,ny))
  call random_number( grid%temperature )
  ! etc...

  print*,grid%temperature

end program typesdemo2
```

Two dimension array

putting type definition in a module allows it to be **use**d in several routines

```
module gridDefinition
  implicit none
  type grid
     integer nx,ny
     real dx,dy              ! spacing
     real,allocatable,dimension(:,:)    :: T
     real,allocatable,dimension(:,:,:)  :: V
  end type grid
end module griddefinition

program typedemo3
  use gridDefinition
  implicit none
  type(grid):: stuff

  print '(a,$)',"Input nx and ny :"
  read*,stuff%nx,stuff%ny
  call initialise_grid (stuff)
  print *,stuff%T
  print *,stuff%V

contains

  subroutine initialise_grid(a)
    implicit none
    type(grid),intent(inout):: a

    a%dx=1./a%dx; a%dy=a%dx
    allocate( a%T(a%nx,a%ny),a%V(2,a%nx,a%ny) )
    call random_number (a%T);    a%V = 0.
  end subroutine initialise_grid

end program typedemo3
```

can pass all information to one variable

internal subroutine inherits variables from containing routine

Types and namelist input

- You can put defined types in a namelist
  - type(mytype):: a
  - namelist /stuff/ a allowed
- but not individual parts of it
  - namelist /stuff/ a%b not allowed

save

- Typically, local variables in functions or subroutines are deleted on exit.
- If you **save** them, they will be permanent, and keep their value for subsequent times the procedure is called.
- Examples
  - real,save:: a,b ! in variable declaration statement
  - save ! saves all variables in procedure
  - save x,y,z ! saves named variables
- The default behaviour depends on compiler! It can also be specified, e.g.,
  - ifort **-save** (default save) or ifort **-auto** (default delete)
  - gfortran **-fno-automatic** (default save)

forall (f95): alternative to **do** loops

Can add as many as loops you want

- e.g., forall(i=1:nx,j=1:ny) T(i,j)=sin(C*(i-1))
- Order of indices shouldn’t matter => suitable for parallel computation
- Optional mask, e.g., forall(i=1:nx,j=1:ny,C(i,j)==0.)

do something to the element C(i,j)==0,改变数组里等于零的值

- Also a block version
- **forall...**
- ...some statements
- **end forall**

Example solution: day 2

Try to keep things short and simple (less room for bugs)

```
program SecDeriv
  implicit none

  integer:: n,i
  real:: h
  real,allocatable:: y(:)

  print '(a,$)','Input grid spacing    :'; read*,h
  print '(a,$)','Input number of points:'; read*,n

  allocate(y(n))
  forall(i=1:n) y(i) = ((i-1)*h)**2

  print*,'Second derivative = ',d2(y,h)
  deallocate(y)


contains

  function d2(f,h)
    real,intent(in) :: f(:),h
    real            :: d2(size(f))
    integer         :: i

    forall(i=2:size(f)-1) d2(i) = (f(i+1)+f(i-1)-2*f(i))/h**2
    d2(1)       = 0.
    d2(size(f)) = 0.
  end function d2

end program SecDeriv
```

faster than do loops

where

- Operates on all elements of an array

```
program wheredemo
  implicit none

  real a(5,5)
  call random_number(a)

  where (a>0.5) a=1. ! single-line where
  print*,a
  print*

  call random_number(a); a=a-0.5

  where (a<0.0)    ! where block
     a=0.
  elsewhere
     a=sqrt(a)
  end where

  print*,a
end program wheredemo
```

do a action when the condition is met.

Binary versus text (ascii) input/output
aka “unformatted” and “formatted”

- Binary I/O is
  - **faster** (data goes directly between memory and file, no conversion to text)
  - more **compact** (e.g., takes ~12 characters to write a 4-byte number with full precision)
  - more **accurate** (full accuracy retained, nothing lost in conversion)
- but...
  - more difficult to read files into other applications (e.g., Matlab)
  - different byte orders for different CPUs (most- or least-significant byte first)

Opening an unformatted (binary) file

- Same as text file except use **form='unformatted'**, e.g.,
- open(1,file='myfile',form='unformatted')
- Optional keywords:
  - access='direct' or 'sequential' ! see next slide
  - recl=N ! record length:see next slide
  - status='old' or 'new' or 'replace' or 'scratch' or 'unknown'

    default

    store the variable when running and delete when exit
  - position='asis' or 'rewind' or 'append'

    从上次结束处继续

    write things at the beginning

    write things at the end
  - action='read' or 'write' or 'readwrite'
  - err=label ! if error, goes to label

    line number
  - iostat=ios ! ios=0 if successful

2 types of unformatted file

- access= ‘**sequential**’
  - data is written or read sequentially: no jumping back and forth
- access= ‘**direct**’
  - data is written in fixed-length records with length defined by **recl=N**
  - any record can be directly read or rewritten in any order

write and **read** to binary file

- Basically the same as for formatted files except you don’t specify a format, e.g.,
  - write (1) a,b,i
  - read (1,err=10,end=20,iostat=ios) a,b,i
  - read (1,rec=recordnumber) stuff ! for direct access files
- err, end, and iostat are useful for handling errors (otherwise the code will stop with an error message)

Character string manipulation

- // concatonates strings
- access substrings using string(3:5) can treat string as arrays.
- **len**(string): gives length of string
- **index**(string,sub) - location of a substring in another string search a short string inside the longer string
- **char**(n) converts integer into character
- **ichar**(ch) converts character into integer
- **trim**(string) returns string without trailing spaces
- write to a string using **write**()

Character string examples

```
program strings
  implicit none
  character(len=13):: a='one two three',b='MyFile',c,d
  integer n

  print*,len(a)                 ! length of string
  print*,a(1:3)                 ! accessing substring
  print*,index(a,"two")         ! prints 5
  print*,(char(n),n=0,255)      ! prints lots of characters
  print*,ichar('a')             ! prints 97
  n=99
  write(c,'(i2)') n             ! formatted write to string
  d=trim(b) // trim(c)          ! trim & concatonate
  print*,d                      ! writes MyFile99

end program strings
```

print characters whose ascii code from =-255

string name

把后面的字符串加到前面的字符串的后面

Named (keyword) arguments and optional arguments

- Normally subroutine/function arguments are identified by the order in which they are listed
- Instead, they can be labelled,
  - e.g., call delsquared(field=a,h=gridspacing,...)
- Some arguments can also be optional
  - declare as such
  - Use ‘present’ to determine if they are present
  - In this case, keywords may be helpful to clarify things.

```fortran
program KeywordOptional
  implicit none
  real:: x=1.2,y=9.8

  print*,apbxc(x,1.2,y)        ! using order
  print*,apbxc(a=x,c=y,b=1.2)  ! using keywords

  print*,sumsome(x,y)          ! without optional arg
  print*,sumsome(x,y,5.)       ! with optional arg

contains

  real function apbxc(a,b,c)
    real,intent(in):: a,b,c
    apbxc = a+b*c
  end function apbxc

  real function sumsome(a,b,c)
    real,intent(in):: a,b
    real,intent(in),optional:: c
    if (present(c)) then
       sumsome = a+b+c
    else
       sumsome = a+b
    end if
  end function sumsome

end program KeywordOptional
```

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

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

Temperature field moves with velocity field.

For incompressible flow

$$\nabla \cdot \vec{v} = 0$$ Not destroy mass

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

对流

- Simple 1D Matlab script shows how centered or downwind schemes go unstable very quickly!

right side is the direction where the material comes from.

- 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

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

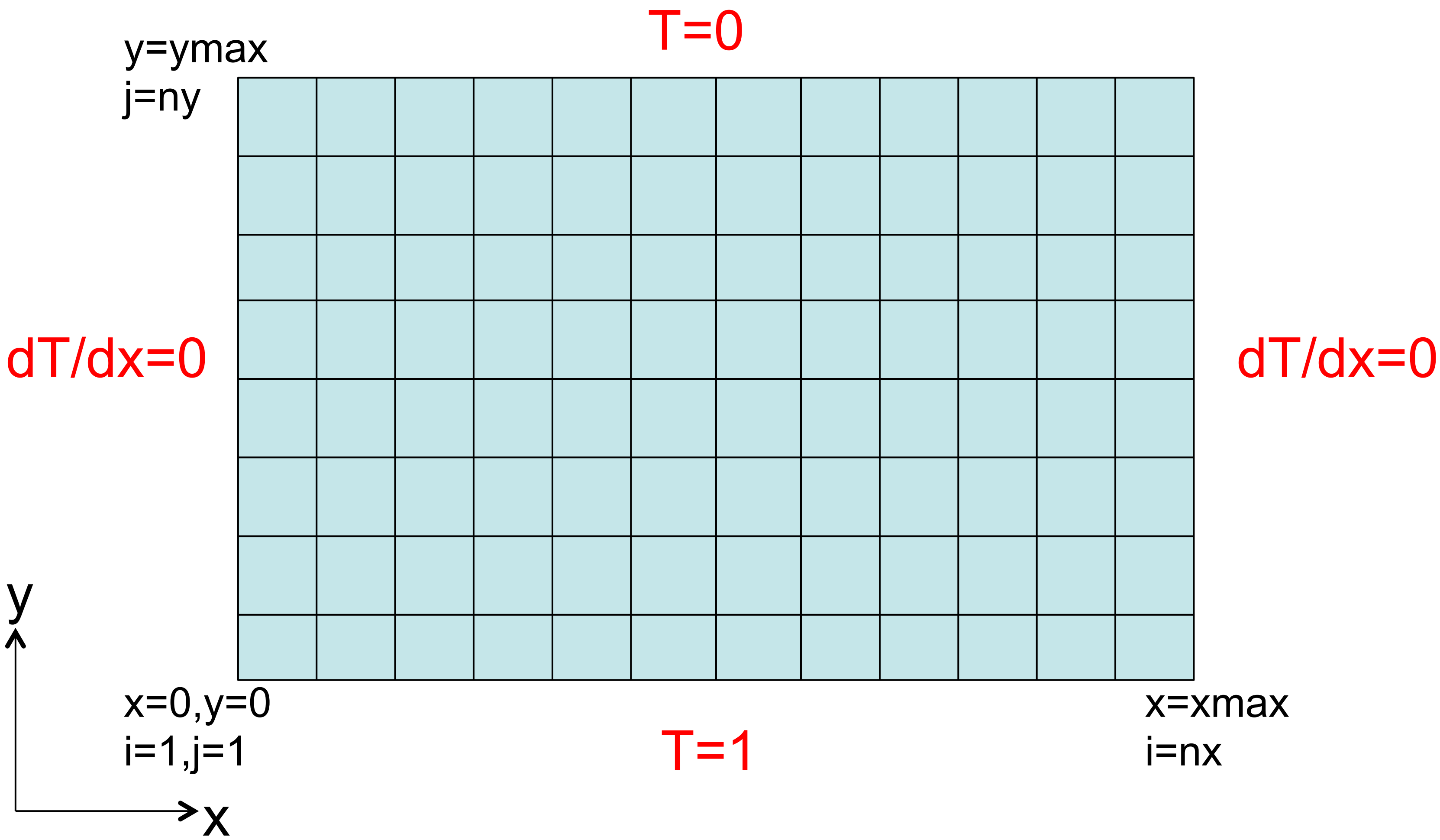
$$\left(\nabla^2 T\right)_{i,j}^{(t_1)} = \left(\frac{T_{i-1,j}^{(t_1)} - 2T_{i,j}^{(t_1)} + T_{i+1,j}^{(t_1)}}{(\Delta x)^2} + \frac{T_{i,j-1}^{(t_1)} - 2T_{i,j}^{(t_1)} + T_{i,j+1}^{(t_1)}}{(\Delta y)^2}\right)$$

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

Upwind derivatives

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

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

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

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

Treatment of boundary conditions

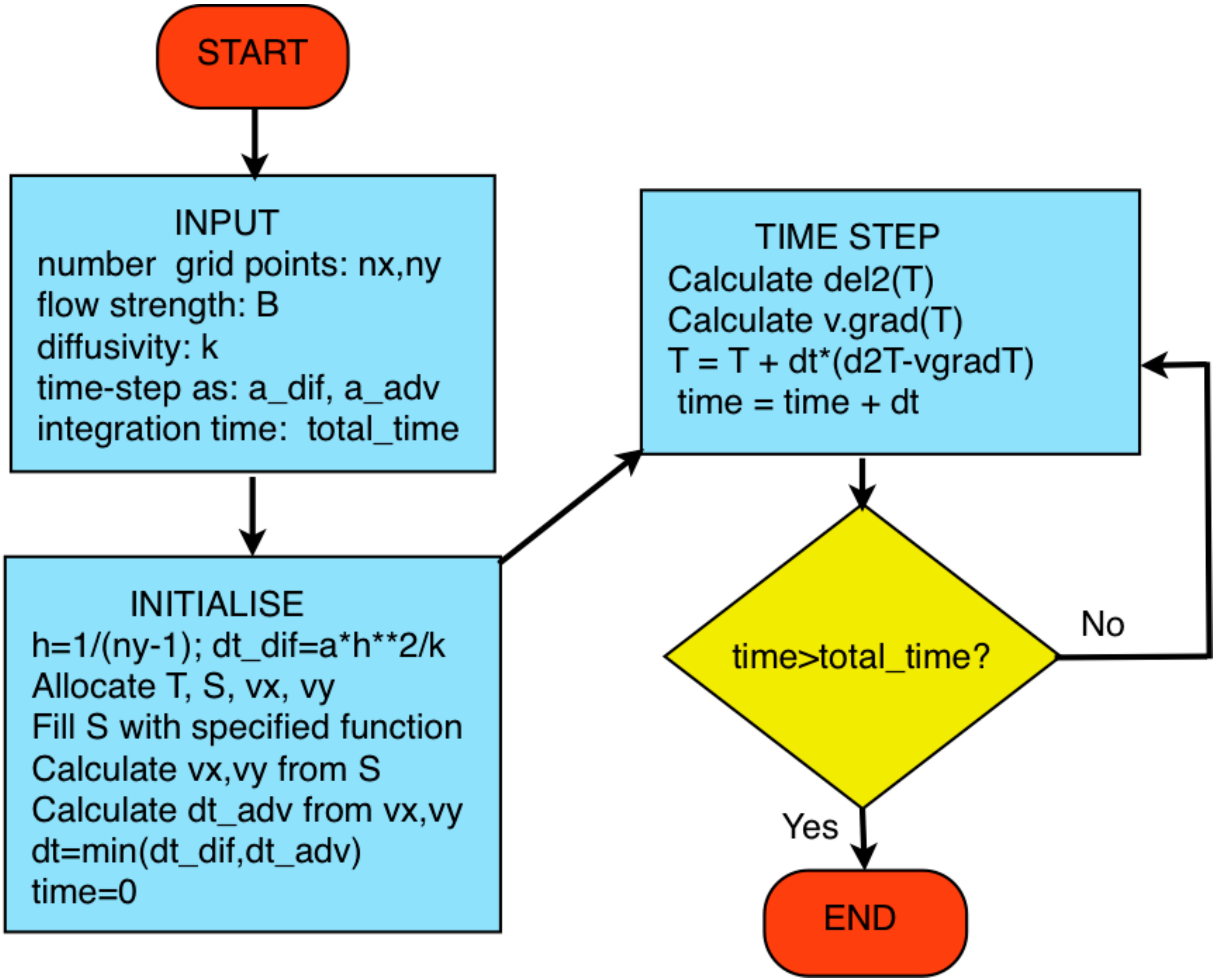
Sides: $$\frac{dT}{dx} = 0$$

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
  - $\boxed{\text{calculate vel. (vx,vy) from streamfunction S}}$
  - $\boxed{\text{calculate v.grad(T)}}$
- Put the new subroutines with the old del2 subroutine into a module
- (for maximum credit) Define a new variable type that holds T, phi, V, nx, ny, grid spacing and pass this between routines

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.

Obtaining results

- Take timesteps and write and plot the results
- After a sufficient integration time the system should reach a $\boxed{\text{steady-state}}$, 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)

B=0.1

B=1

B=10

B=100