Numerical Modelling in FORTRAN day 7

Paul Tackley, 2014

Today's Goals

1. Learn about **pointers**, **generic procedures** and **operators**
2. Learn new things about **iterative** solvers
3. Combine advection-diffusion and Poisson solvers to make a **convection simulation program**

Pointers

- Point to either
  - (i) data stored by a variable declared as a **target**, or
  - (ii) another pointer, or
  - (iii) an area of allocated memory
- (i) and (ii) are mainly useful for creating interesting data structures (like linked lists); for us (iii) will be the most useful.
- "**=>**" is used to point a pointer at something
- See examples next slide
- Functions can also return pointers ("**pointer functions**")

Pointers to scalar variables

```
program pointless
  implicit none
  real,pointer:: p
  real,target:: a=3.1,b=4.7

  p=>a                 ! point p to a
  print*,p             ! prints 3.1 (value of a)
  p=4.0                ! changes a to 4.0
  print*,a             ! prints 4.0
  p=>b                 ! now points to b
  print*,b             ! prints 4.7
  print*,associated (p) ! test status: prints true
  nullify(p)           ! points p at nothing
  print*,associated (p) ! prints false

end program pointless
```

看P的
状态

Pointers to array or array section

```
program pointarray
  implicit none
  integer,parameter:: n=10
  real,target:: T(n,n)
  real,pointer:: bot_boundary(:), &
       top_boundary(:), center4(:,:)

  top_boundary => T(:,1)
  bot_boundary => T(:,n)
  center4 => T(n/2:n/2+1,n/2:n/2+1)

  call random_number(T)
  top_boundary = 0.0  ! easy way to do
  bot_boundary = 1.0  ! boundary conditions
  center4 = 1.0

  print*,T
end program pointarray
```

Pointers in defined types

similar to allocatable array (i.e.,not pointing to a variable)

在95或
者03中

```
program pointertype
  implicit none

  type array2D
     real,pointer:: a(:,:)    ! 2D array
  end type array2D

  type(array2D),allocatable:: T(:) ! 1D array of 2D arrays
  integer i,n,ng

  print '(a,$)',"Number of multigrid levels:"
  read*,ng
  allocate (T(ng))   ! allocate number of grids
  do i = 1,ng
     n = 2**(i+1)          ! #grid points in this grid
     print*,'Allocating grid',n,n
     allocate (T(i)%a(n,n))  ! each grid
  end do

  !.... rest of program

end program pointertype
```

Hint: arrays in subprograms

- Arrays that are declared using subroutine arguments have a limited size, e.g.

```
Subroutine do_something (n,m)
Integer,intent(in):: n,m
Real:: local_array(n,m)
```

Define a local array

- Better to use allocatable arrays

```
Subroutine do_something (n,m)
Integer,intent(in):: n,m
Real,allocatable:: local_array(:,:)
allocate(local_array(n,m))
```

Generic procedures

(i.e., using a generic name to access different procedures)

固有的

- e.g., the intrinsic (built-in) **sqrt** function. There are several versions: **sqrt**(real), **dsqrt**(double precision), **csqrt**(complex). Use the generic name and the correct one will automatically be used.
- You can define the same thing. Define similar sets of procedures then define a generic **interface** to them (see example next slide).
- Easiest way: in a **module**

generic interface **apbxc**

to functions **rapbxc** and **iapbxc**

Program using this generic function

```
module goodstuff
  implicit none

  interface apbxc   ! define generic interface
     module procedure rapbxc,iapbxc
  end interface

contains

  real function rapbxc(a,b,c) ! actual function
    real,intent(in):: a,b,c
    rapbxc = a+b*c
  end function rapbxc

  integer function iapbxc(a,b,c)
    integer,intent(in):: a,b,c
    iapbxc = a+b*c
  end function iapbxc

end module goodstuff
!--------------------------------
program generic
  use goodstuff
  implicit none
  real:: a=1.2,b=3.4,c=5.6
  integer:: i=2,j=3,k=4

  print*,apbxc(a,b,c)   ! real arguments
  print*,apbxc(i,j,k)   ! integer argumnts

end program generic
```

Overloading

- **Overloading** means that one operator or procedure name is used to refer to several procedures: which one is used depends on the variable types.
- Examples
  - **apbxc** is overloaded with **rapbxc** and **iapbxc**
  - *,+,-,/ are overloaded with integer,real,complex versions in different precisions
- You can overload existing operators, or define new overloaded operators or procedures

Overloading existing + and –

```
module coordstuff
  implicit none

  type point                    ! defined type
     real:: x,y,z
  end type point

  interface operator (+)   ! new version of +
     module procedure pointplus
  end interface
  interface operator (-)   ! new version of -
     module procedure pointminus
  end interface

contains
  function pointplus(a,b)
    type(point):: pointplus
    type(point),intent(in):: a,b
    pointplus%x = a%x + b%x
    pointplus%y = a%y + b%y
    pointplus%z = a%z + b%z
  end function pointplus

  function pointminus(a,b)
    type(point):: pointminus
    type(point),intent(in):: a,b
    pointminus%x = a%x - b%x
    pointminus%y = a%y - b%y
    pointminus%z = a%z - b%z
  end function pointminus
end module coordstuff
```

Useful for defined types

```
program test
   use coordstuff
   type(point):: p1,p2,p3
   p1%x=1.2; p1%y=0. ; p1%z=3.1
   p2%x=0. ; p2%y=1.2; p2%z=1.7

   p3 = p1 - p2       ! using overloaded -
   print*,p3
   p3 = p1 + p2       ! using overloaded +
   print*,p3
end program test
```

Defining new operator .distance.

always begin with .

```
module coordsagain
  implicit none

  type point                    ! defined type
     real:: x,y,z
  end type point

  interface operator (.distance.) ! new operator
     module procedure pointseparation
  end interface

contains

  real function pointseparation(a,b)
    type(point),intent(in):: a,b
    pointseparation = sqrt(   &
         (a%x-b%x)**2+(a%y-b%y)**2+(a%z-b%z)**2)
  end function pointseparation

end module coordsagain
!-----------------------------------
program test
  use coordsagain
  type(point):: p1,p2
  real:: d
  p1%x=1.2; p1%y=0. ; p1%z=3.1
  p2%x=0. ; p2%y=1.2; p2%z=1.7

  d = p1.distance.p2      ! using new operator
  print*,d
end program test
```

distance between the two points

Overloading "=". Useful for conversion between different types: in this case point to real.

Must use subroutine with 1st argument intent(out) and 2nd argument intent(in).

```
module coords3
  implicit none

  type point                       ! defined type
     real:: x,y,z
  end type point

  interface assignment (=)      ! new "="
     module procedure absvec ! converts
  end interface                   ! point to real

contains

  subroutine absvec(a,b)    ! calculates distance
    real,intent(out):: a    !   from origin
    type(point),intent(in):: b
    a = sqrt(b%x**2+b%y**2+b%z**2)
  end subroutine absvec

end module coords3
!-----------------------------------
program test
  use coords3
  type(point):: p1
  real:: d
  p1%x=1.2; p1%y=0. ; p1%z=3.1

  d = p1        ! using new =,
  print*,d
end program test
```

Now back to iterative solvers…

Goal for today's exercise: Calculate velocity field from temperature field =>**convection**

e.g., for highly viscous flow (e.g., Earth's mantle) with constant viscosity (P=pressure, Ra=Rayleigh number):

$$-\nabla P + \nabla^2 \vec{v} = -Ra.T\hat{y}$$

(y points up)

Substituting the streamfunction for velocity, we get:

$$\left(v_x, v_y\right) = \left(\frac{\partial \psi}{\partial y}, -\frac{\partial \psi}{\partial x}\right) \qquad \nabla^4 \psi = Ra\frac{\partial T}{\partial x}$$

writing as 2 Poisson equations:

$$\nabla^2 \psi = \omega \qquad \nabla^2 \omega = Ra\frac{\partial T}{\partial x}$$

the **streamfunction-vorticity** formulation

Mathematical aside

$$-\nabla P+\nabla^2\vec{v}=-Ra.T\hat{y}$$ Is a vector equation with x and y parts:

$$-\frac{\partial P}{\partial x}+\nabla^2 v_x=0 \quad (1) \qquad -\frac{\partial P}{\partial y}+\nabla^2 v_y=-Ra.T \quad (2)$$

Take $\frac{\partial(1)}{\partial y}-\frac{\partial(2)}{\partial x}\Rightarrow$ $$\nabla^2\left(\frac{\partial v_x}{\partial y}-\frac{\partial v_y}{\partial x}\right)=Ra\frac{\partial T}{\partial x}$$

Use streamfunction: $$\nabla^2\left(\frac{\partial^2\psi}{\partial y^2}+\frac{\partial^2\psi}{\partial x^2}\right)=\nabla^2\nabla^2\psi=Ra\frac{\partial T}{\partial x}$$

This is the z-component of the curl of the momentum equation:

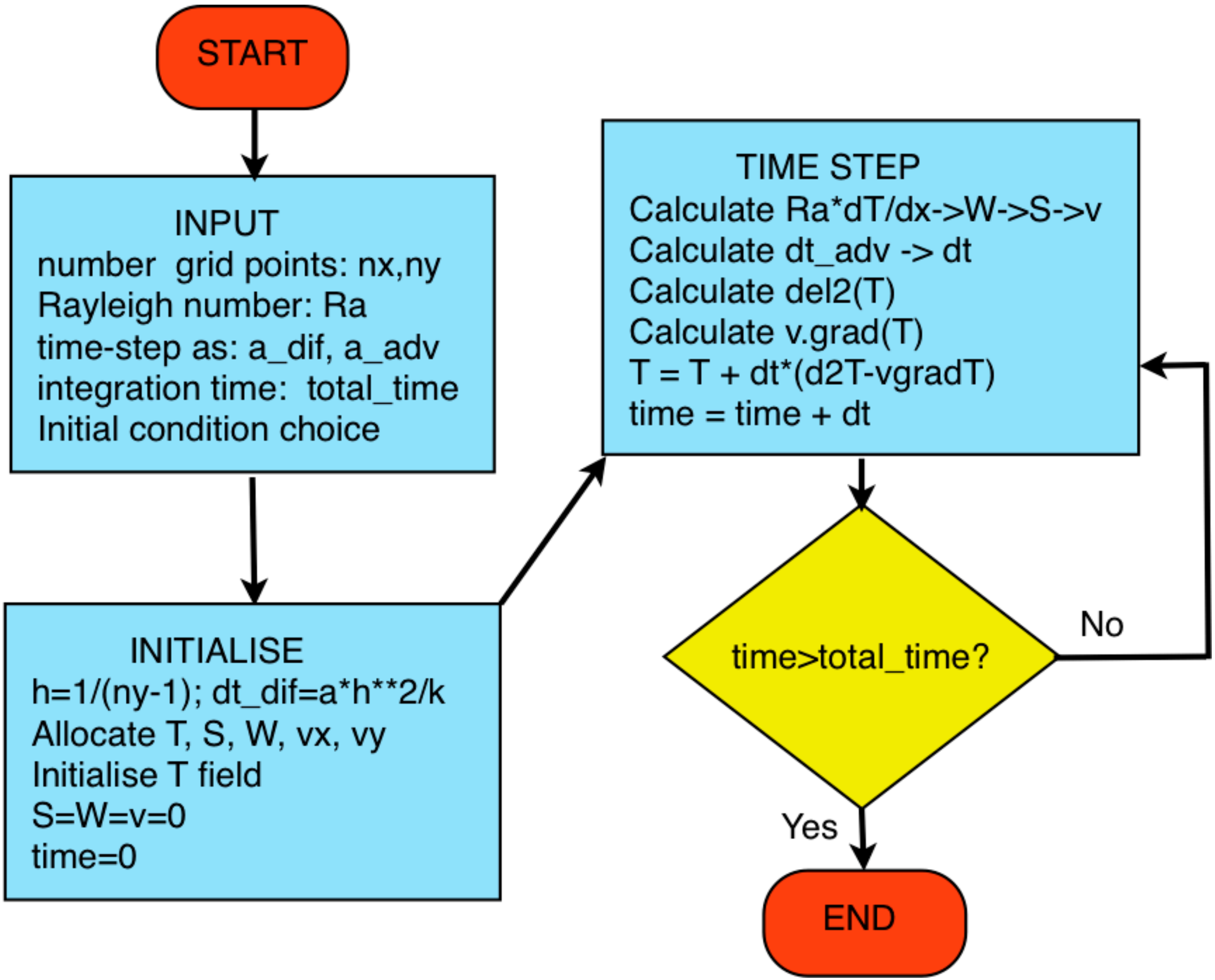
$$\nabla\times\mathbf{F}=\left(\frac{\partial F_z}{\partial y}-\frac{\partial F_y}{\partial z}\right)\hat{\mathbf{x}}+\left(\frac{\partial F_x}{\partial z}-\frac{\partial F_z}{\partial x}\right)\hat{\mathbf{y}}+\left(\frac{\partial F_y}{\partial x}-\frac{\partial F_x}{\partial y}\right)\hat{\mathbf{z}}$$

Combine advection-diffusion and Poisson solver routines

- Initialise
  - T=random or spike or ...
  - S, W=0
  - grid spacing h=1/ny and diffusive timestep
- Timestep until desired time is reached
  - calculate right-hand side=-Ra*dT/dx
  - Poisson solve to get W from rhs
  - Poisson solve to get S from W
  - calculate V from S
  - calculate advective timestep and min(adv,diff)
  - take advection and diffusion time step
  - t=t+dt: have we finished? If not, loop again.

Use multigrid solver

Use existing adv-diff routines


START
INPUT
number grid points: nx,ny
Rayleigh number: Ra
time-step as: a_dif, a_adv
integration time: total_time
Initial condition choice
INITIALISE
h=1/(ny-1); dt_dif=a*h**2/k
Allocate T, S, W, vx, vy
Initialise T field
S=W=v=0
time=0
TIME STEP
Calculate Ra*dT/dx->W->S->v
Calculate dt_adv -> dt
Calculate del2(T)
Calculate v.grad(T)
T = T + dt*(d2T-vgradT)
time = time + dt
time>total_time?
No
Yes
END

Program definition

- Input parameters:
  - nx,ny : number of grid points (y=vertical, grid spacing is the same, i.e., dx=dy=h)
  - Ra: Rayleigh number (typical range 1e3-1e7)
  - total_time: large enough to reach stable state
- Boundary conditions
  - T: as before, T=1 at bottom, 0 at top, dT/dx=0 at sides
  - S and W: 0 at all boundaries
- Example structure
  - Module with Poisson solver routines
  - Module with 2_deriv, v_gradT, S_to_V routines
  - main program
- Try to write frames at different times and make an animation!

Exercise

- Get everything together and run a test case with
  - nx=257, ny=65 (i.e., aspect ratio 4)
  - Ra=1e5
  - total_time=0.1
  - err=1.e-3
  - Random initial T field
- Email code, plots and parameter file for this case.

Should look something like this