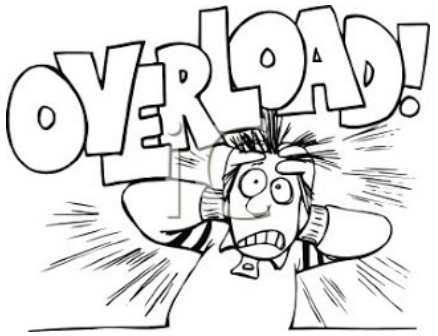
Numerical Modelling in FORTRAN day 8

Paul Tackley, 2020

Today‘s Goals

1. Learn more about procedures including pure procedures, elemental procedures, generic procedures and operators (examples of overloading)
2. Write low-Prandtl number convection program.



Projects: start thinking about

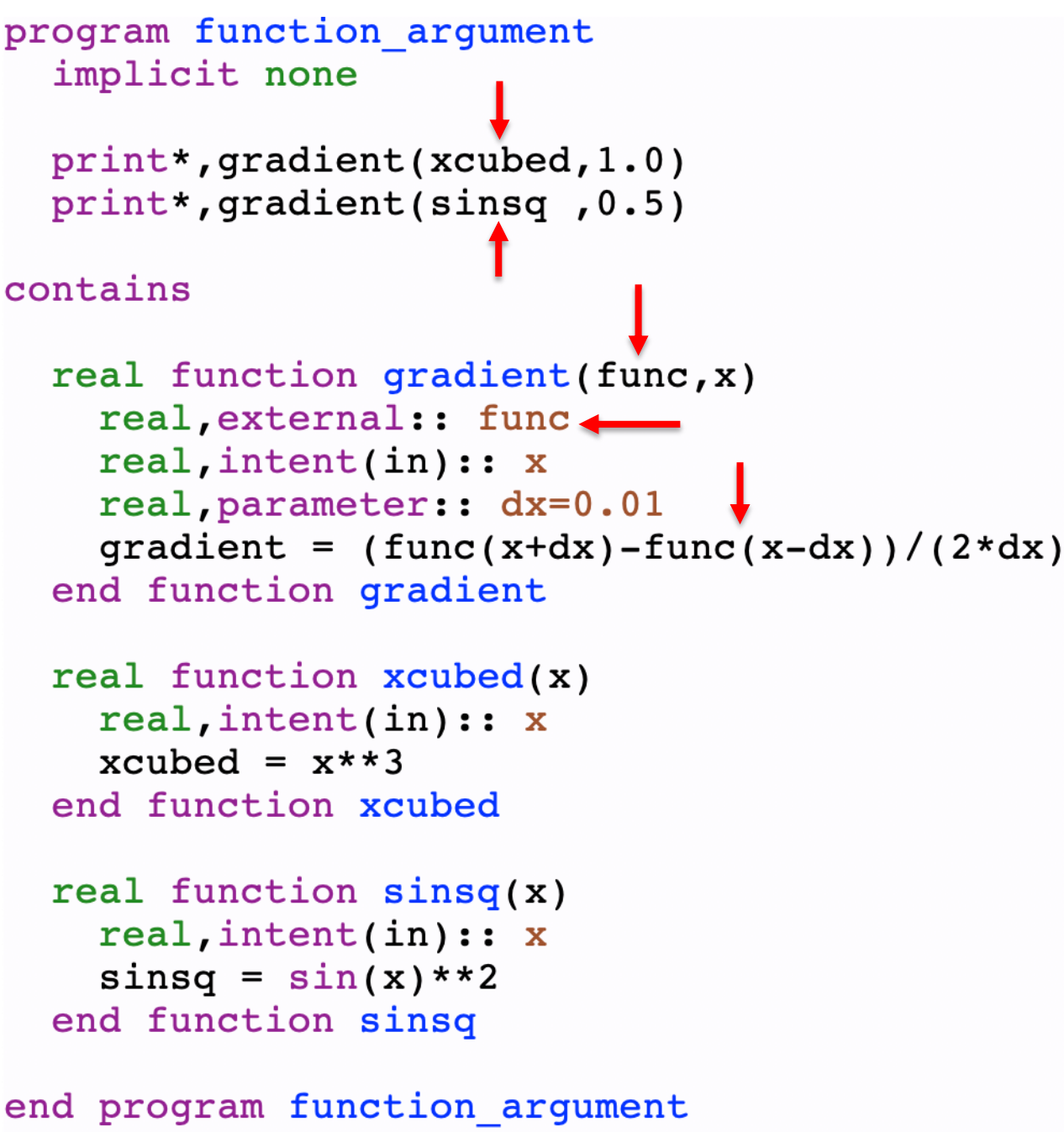
Agree topic with me
before final lecture

More about procedures (=functions/subroutines)

- Passing procedure names as dummy arguments
- Pure functions & subroutines
- Elemental procedures
- Impure elemental procedures
- Generic procedures and overloading

Passing function names as function arguments

- Also possible with subroutines (declare as external)



```
program function_argument
  implicit none

  print*,gradient(xcubed,1.0)
  print*,gradient(sinsq ,0.5)

contains

  real function gradient(func,x)
    real,external:: func
    real,intent(in):: x
    real,parameter:: dx=0.01
    gradient = (func(x+dx)-func(x-dx))/(2*dx)
  end function gradient

  real function xcubed(x)
    real,intent(in):: x
    xcubed = x**3
  end function xcubed

  real function sinsq(x)
    real,intent(in):: x
    sinsq = sin(x)**2
  end function sinsq

end program function_argument
```

Pure functions

- Functions that have no side effects:
  - Do not modify any arguments: all arguments must be intent(in)
  - Do not modify module variables
  - No saved variables, external file I/O or stop
- Helpful for parallelisation of loops: function can be executed simultaneously with different loop indices

Pure subroutines

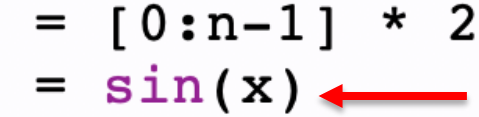
- As with pure functions, but are allowed to modify arguments declared as intent(out) or intent(inout)

```
pure real function harmmean3(a,b,c)
  real,intent(in):: a,b,c
  harmmean3 = 1./(1./a+1./b+1./c)
end function harmmean3

pure subroutine meanstddev(a,mean,stddev)
  real,intent(in):: a(:)
  real,intent(out):: mean,stddev
  mean = sum(a)/size(a)
  stddev = sqrt(sum(a*a)/size(a)-mean**2)
end subroutine meanstddev
```

Elemental functions

- Are written for scalar arguments but may also be applied to array arguments
- Intrinsic functions are elemental functions, e.g.

```
real,dimension(n):: x,a

...
x = [0:n-1] * 2*pi/(n-1)
a = sin(x)
print*,a
```



```
 0.0000000E+00  0.6427876   0.9848077   0.8660254   0.3420202
-0.3420202     -0.8660252  -0.9848078  -0.6427873   1.7484555E-07
```

Elemental functions

- Declare using the ”elemental” label

```
elemental real function Gaussian(x,mean,sigma)
  real,intent(in):: x,mean,sigma
  Gaussian = 1./(sigma*sqrt(2*pi))*exp(-0.5*((x-mean)/sigma)**2)
end function Gaussian
```

```
a = Gaussian(x,pi,0.5)
print*,a
```

```
 2.1345763E-09  5.2003152E-06  1.8032945E-03  8.9007244E-02  0.6253240
 0.6253241      8.9007527E-02  1.8032981E-03  5.2002902E-06  2.1345683E-09
```

Elemental functions

- Must normally be pure functions
- All dummy arguments must be scalars
- No pointers (either for arguments or the result)
- Dummy arguments must not be used in type declaration statements
- Elemental subroutines are also possible, with the same rules

Impure elemental procedures

- These can **modify** their calling arguments
- Must be declared with the impure keyword
- Arguments modified must be declared intent(inout).
- When called with an array, execution is **element-by-element in index order**, with the first index changing most rapidly.

```
program impure   ! See Chapman section 9.6.3
  implicit none
  real a(5),b(5),sum
  integer i

  sum = 0.
  a = [1.,2.,3.,4.,5.]
  b = cumulative_sum(a,sum)
  print*,b

contains

  impure elemental real function cumulative_sum(a,sum)
    real,intent(in):: a
    real,intent(inout):: sum
    sum = sum + a
    cumulative_sum = sum
  end function cumulative_sum

end program impure
```

```
1.000000       3.000000       6.000000       10.00000       15.00000
```

Generic procedures

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

- 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**

A 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 –

Useful for derived types

```
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
```

```
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.

```
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
```

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
```

Combine operators into 1 module

- Generic procedures
- New operators
- Overloaded operators (e.g., +, -, =)

The interfaces

```
module coords
  implicit none

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

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

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

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

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

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

contains
```

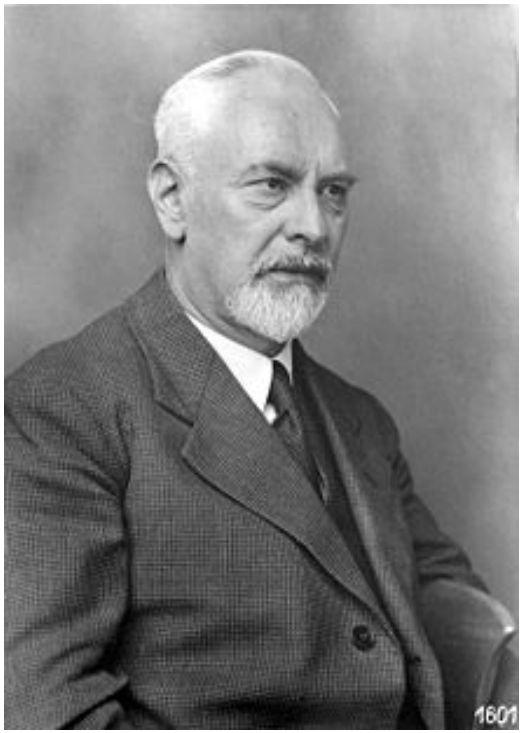
The procedures

```
  real function rapbxc(a,b,c) ! real version of fn
    real,intent(in):: a,b,c
    rapbxc = a+b*c
  end function rapbxc

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

  real function pointseparation(a,b) ! for .distance.
    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

  function pointplus(a,b)   ! for +
    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)   !  for -
    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

  subroutine absvec(a,b)  ! for = (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 coords
```

The test program

```
program test
  use coords
  real:: a=1.2,b=3.4,c=5.6,d
  integer:: i=2,j=3,k=4
  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

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

  p3 = p1 - p2      ! using overloaded -
  p3 = p1 + p2      ! using overloaded +

  d = p1            ! using overloaded =,

  d = p1.distance.p2 ! using new operator

end program test
```

Programming:
Finite Prandtl number convection
(i.e., almost any fluid)



Ludwig Prandtl (1875-1953)

Values of the Prandt number Pr

$$\mathrm{Pr} = \frac{\nu}{\kappa}$$

ν: Viscous diffusivity

κ: Thermal diffusivity

- Liquid metals: 0.004-0.03
- Air: 0.7
- Water: 1.7-12
- Rock: ~10^{24} !!! (effectively infinite)

Finite-Prandtl number convection

- Existing code assumes infinite Prandtl number
  - also known as Stokes flow
  - appropriate for highly-viscous fluids like rock, honey etc.
- Fluids like water, air, liquid metal have a lower Prandtl number so equations must be modified

Applications for finite Pr

- Outer core (geodynamo)
- Atmosphere
- Ocean
- Anything that's not solid like the mantle

Equations

- Conservation of mass (= ‘continuity’)
- Conservation of momentum (‘Navier-Stokes’ equation: F=ma for a fluid)
- Conservation of energy

Claude Navier
(1785-1836)

Sir George Stokes
(1819-1903)

Finite Pr Equations

Navier-Stokes equation: F=ma for a fluid

Coriolis force

$$\rho\left(\frac{\partial \vec{v}}{\partial t}+\vec{v}\cdot\nabla\vec{v}\right)=-\nabla P+\rho\nu\nabla^2\vec{v}+2\rho\vec{\Omega}\times\vec{v}+g\rho\alpha T\hat{y}$$

"ma"

Valid for constant viscosity only

continuity and energy equations same as before

$$\frac{\partial T}{\partial t}+\vec{v}\cdot\nabla T=\kappa\nabla^2 T+Q \qquad \nabla\cdot\vec{v}=0$$

ρ=density, ν=kinematic viscosity, g=gravity,
α=thermal expansivity

Non-dimensionalise the equations

- Reduces the number of parameters
- Makes it easier to identify the dynamical regime
- Facilitates comparison of systems with different scales but similar dynamics (e.g., analogue laboratory experiments compared to core or mantle)

Non-dimensionalise to thermal diffusion scales

- Lengthscale D (depth of domain)
- Temperature scale (T drop over domain)
- Time to D^2/κ
- Velocity to κ/D
- Stress to $\rho\nu\kappa/D^2$

Nondimensional equations

$$\nabla \cdot \vec{v} = 0 \qquad \frac{\partial T}{\partial t} + \vec{v} \cdot \nabla T = \nabla^2 T$$

$$\frac{1}{\mathrm{Pr}}\left(\frac{\partial \vec{v}}{\partial t} + \vec{v} \cdot \nabla \vec{v}\right) = -\nabla P + \nabla^2 \vec{v} + \frac{1}{Ek}\vec{\Omega} \times \vec{v} + Ra.T\hat{y}$$

$$\mathrm{Pr} = \frac{\nu}{\kappa} \qquad Ek = \frac{\nu}{2\Omega D^2} \qquad Ra = \frac{g\alpha \nabla T D^3}{\nu\kappa}$$

Prandtl number Ekman number Rayleigh number

As before, use streamfunction

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

Also simplify by assuming 1/Ek=0

Eliminating pressure

- Take curl of 2D momentum equation: curl of grad=0, so pressure disappears
- Replace velocity by vorticity: $\vec{\omega} = \nabla \times \vec{v}$
- in 2D only one component of vorticity is needed (the one perpendicular to the 2D plane), $\nabla^2 \psi = \omega_z$

$$\frac{1}{\mathrm{Pr}}\left(\frac{\partial \omega}{\partial t} + v_x \frac{\partial \omega}{\partial x} + v_y \frac{\partial \omega}{\partial y} \right) = \nabla^2 \omega - Ra \frac{\partial T}{\partial x}$$

=> the streamfunction-vorticity formulation

$$\frac{1}{\mathrm{Pr}}\left(\frac{\partial \omega}{\partial t}+v_x\frac{\partial \omega}{\partial x}+v_y\frac{\partial \omega}{\partial y}\right)=\nabla^2\omega-Ra\frac{\partial T}{\partial x}$$

$$\nabla^2\psi=-\omega \qquad \left(v_x,v_y\right)=\left(\frac{\partial \psi}{\partial y},-\frac{\partial \psi}{\partial x}\right)$$

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

Note: Effect of high Pr

$$\frac{1}{\mathrm{Pr}}\left(\frac{\partial \omega}{\partial t}+v_x\frac{\partial \omega}{\partial x}+v_y\frac{\partial \omega}{\partial y}\right)=\nabla^2\omega-Ra\frac{\partial T}{\partial x}$$

If Pr->infinity, left-hand-side=>0 so equation becomes Poisson like before:

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

Taking a timestep

(i) Calculate ψ from ω using: $\nabla^2\psi = \omega$

(ii) Calculate v from ψ $\left(v_x, v_y\right) = \left(\frac{\partial\psi}{\partial y}, -\frac{\partial\psi}{\partial x}\right)$

(iii) Time-step ω and T using explicit finite differences:

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

$$\frac{\partial \omega}{\partial t} = -v_x\frac{\partial \omega}{\partial x} - v_y\frac{\partial \omega}{\partial y} + \mathrm{Pr}\nabla^2\omega - Ra\,\mathrm{Pr}\frac{\partial T}{\partial x}$$

T time step is the same as before

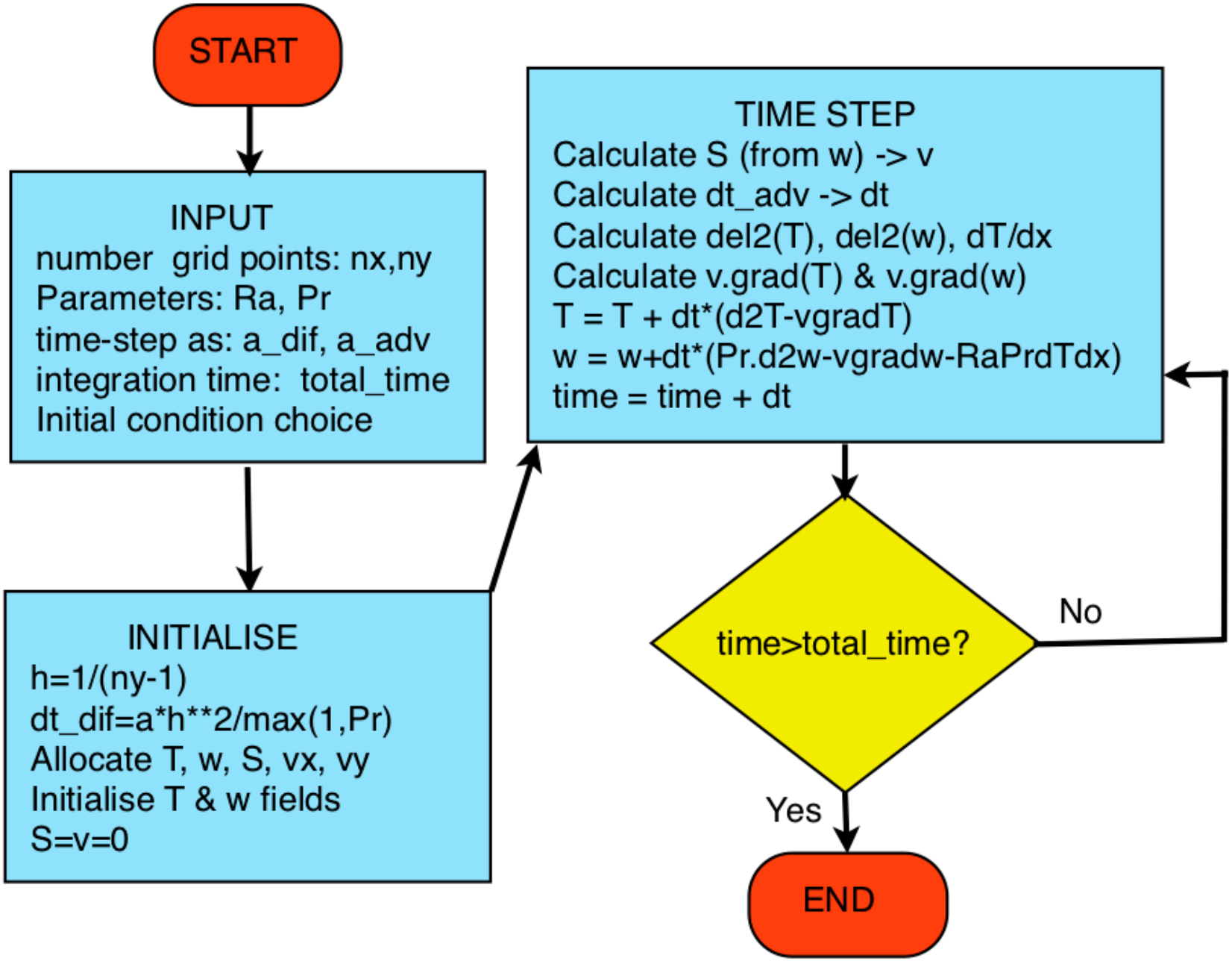
$$\frac{T_{new} - T_{old}}{\Delta t} = -v_x \frac{\partial T_{old}}{\partial x} - v_y \frac{\partial T_{old}}{\partial y} + \nabla^2 T_{old}$$

$$T_{new} = T_{old} + \Delta t \left(\nabla^2 T_{old} - v_x \frac{\partial T_{old}}{\partial x} - v_y \frac{\partial T_{old}}{\partial y} \right)$$

w must now be time stepped in a similar way

$$\frac{\omega_{new} - \omega_{old}}{\Delta t} = -v_x \frac{\partial \omega_{old}}{\partial x} - v_y \frac{\partial \omega_{old}}{\partial y} + \mathrm{Pr} \nabla^2 \omega_{old} - Ra \, \mathrm{Pr} \frac{\partial T_{old}}{\partial x}$$

$$\omega_{new} = \omega_{old} + \Delta t \left(\mathrm{Pr} \nabla^2 \omega_{old} - v_x \frac{\partial \omega_{old}}{\partial x} - v_y \frac{\partial \omega_{old}}{\partial y} - Ra \, \mathrm{Pr} \frac{\partial T_{old}}{\partial x} \right)$$


START
INPUT
number grid points: nx,ny
Parameters: Ra, Pr
time-step as: a_dif, a_adv
integration time: total_time
Initial condition choice
INITIALISE
h=1/(ny-1)
dt_dif=a*h**2/max(1,Pr)
Allocate T, w, S, vx, vy
Initialise T & w fields
S=v=0
TIME STEP
Calculate S (from w) -> v
Calculate dt_adv -> dt
Calculate del2(T), del2(w), dT/dx
Calculate v.grad(T) & v.grad(w)
T = T + dt*(d2T-vgradT)
w = w+dt*(Pr.d2w-vgradw-RaPrdTdx)
time = time + dt
time>total_time?
No
Yes
END

Stability condition

Diffusion: $$dt_{diff} = a_{diff} \frac{h^2}{\max(\mathrm{Pr},1)}$$

Advection: $$dt_{adv} = a_{adv} \min\left(\frac{h}{\max val(abs(vx))}, \frac{h}{\max val(abs(vy))} \right)$$

Combined: $$dt = \min(dt_{diff}, dt_{adv})$$

Modification of previous convection program

- Replace Poisson calculation of w with time-step, done at the same time as T time-step
- Get a compiling code!
- Make sure it is stable and convergent for values of Pr between 0.001 and 10
- Hand in your code, and your solutions to the test cases in the following slides
- Due date: 30 November (2 weeks)

Test cases

- All have nx=257, ny=65, Ra=1e6, total_time=0.1, Tinit='cosine', initial W=0, unless otherwise stated
- In addition to the fields, plot and hand in a graph of maximum velocity vs. time for each case.

Standard parameter file

On web site, lowPrandt_parameters.txt

```
&inputs
 Pr=0.01   ! <---- CHANGE ONLY THIS
 nx=257    ny=65
 total_time=0.1
 Ra=1.e6
 err=1.e-3
 a_dif=0.15   a_adv=0.4
 Tinit='cosine'
 /
```

Empty parameter file

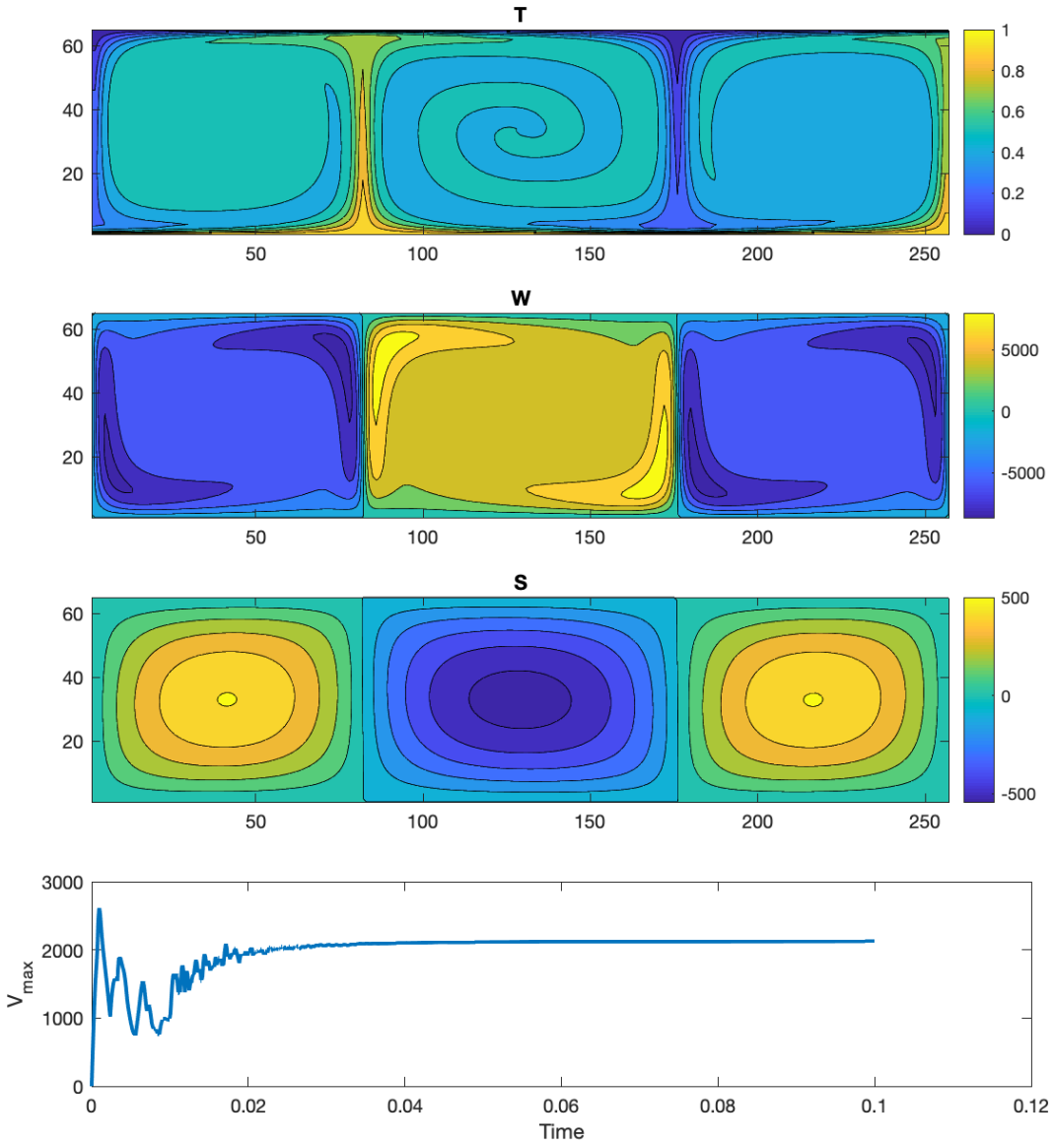
For testing that you can set default values for Input parameters.
On web site, lowPrandt_parameters_EMPTY.txt

```
&inputs
 /
```

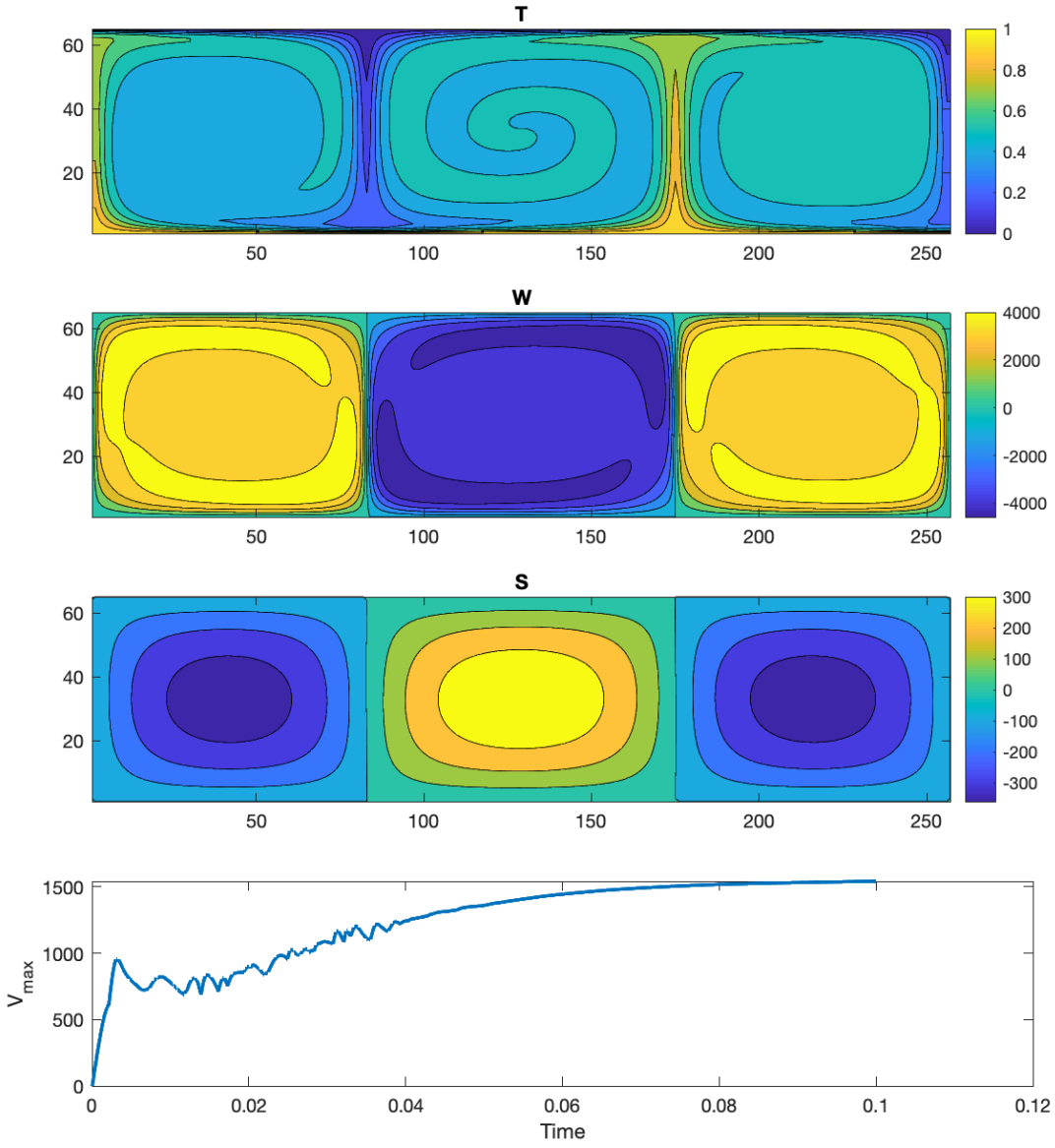
Check for uninitialized variables

- Code should assume sensible values of input parameters, because they are optional in namelist blocks
  - Test using empty namelist block
- All arrays should be initialized (typically to 0)
- Test using recommended debugging options

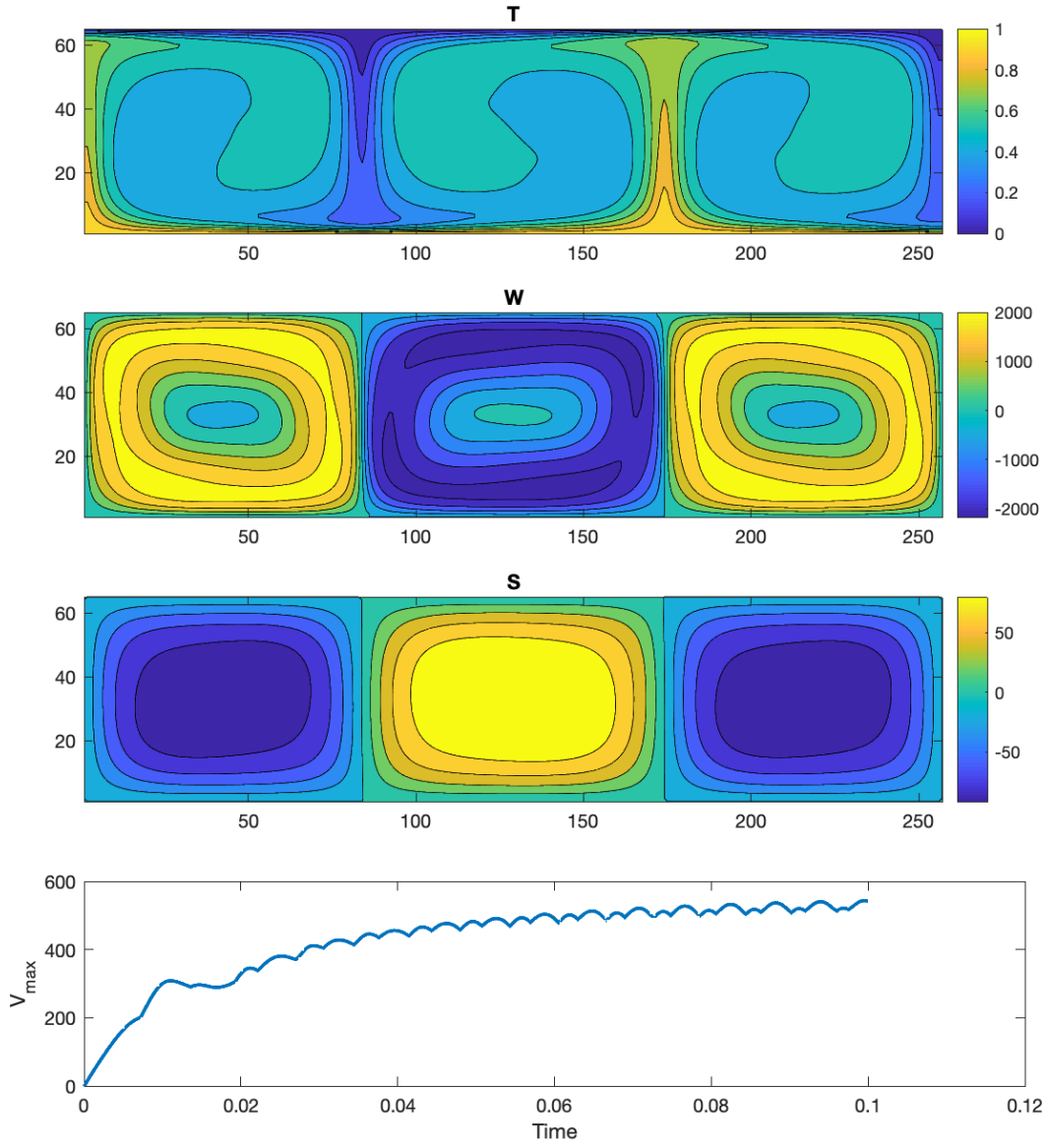
Pr=10


T
60
40
20
50
100
150
200
250
1
0.8
0.6
0.4
0.2
0
W
60
40
20
50
100
150
200
250
5000
0
-5000
S
60
40
20
50
100
150
200
250
500
0
-500
3000
2000
1000
0
V_{max}
0
0.02
0.04
0.06
0.08
0.1
0.12
Time

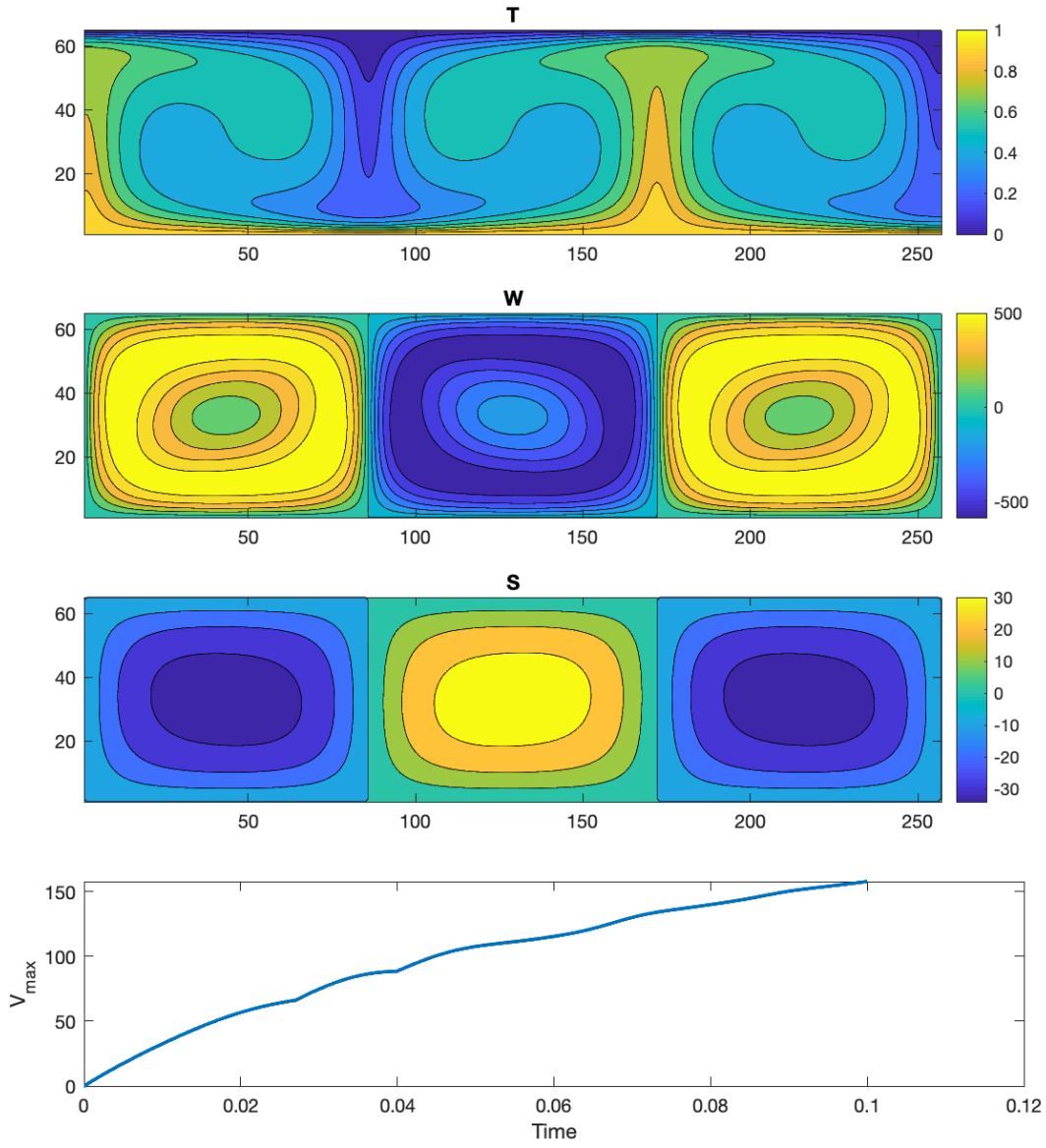
Pr=1


T
60
40
20
50
100
150
200
250
1
0.8
0.6
0.4
0.2
0
W
60
40
20
50
100
150
200
250
4000
2000
0
-2000
-4000
S
60
40
20
50
100
150
200
250
300
200
100
0
-100
-200
-300
1500
1000
500
0
V_{max}
0
0.02
0.04
0.06
0.08
0.1
0.12
Time

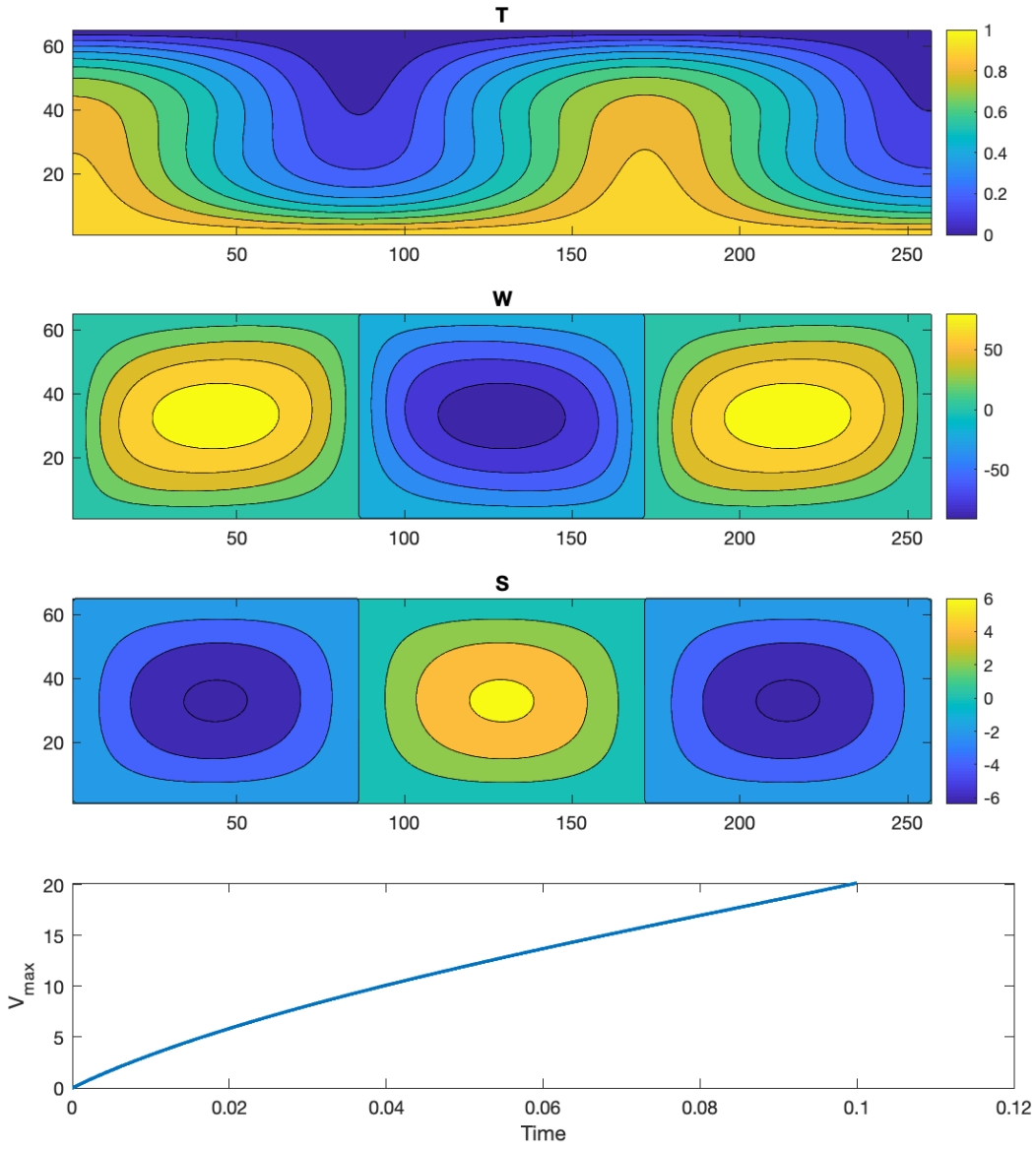
Pr=0.1


T
60
40
20
50
100
150
200
250
1
0.8
0.6
0.4
0.2
0
W
60
40
20
50
100
150
200
250
2000
1000
0
-1000
-2000
S
60
40
20
50
100
150
200
250
50
0
-50
600
400
200
0
V_{max}
0
0.02
0.04
0.06
0.08
0.1
0.12
Time

Pr=0.01


T
60
40
20
50
100
150
200
250
1
0.8
0.6
0.4
0.2
0
W
500
0
-500
S
30
20
10
0
-10
-20
-30
150
100
50
0
0.02
0.04
0.06
0.08
0.1
0.12
Time
V max

Pr=0.001


T
60
40
20
50
100
150
200
250
1
0.8
0.6
0.4
0.2
0
W
60
40
20
50
100
150
200
250
50
0
-50
S
60
40
20
50
100
150
200
250
6
4
2
0
-2
-4
-6
20
15
10
5
0
V_{max}
0
0.02
0.04
0.06
0.08
0.1
0.12
Time

Hand in

- Source files
- Images and graphs for the 5 test cases
- Make sure it compiles with
  - “gfortran –fcheck=all –finit-real=snan –ffpe-trap=invalid,zero,overflow –O0”
- Make sure it runs with both the downloaded full parameter file and the empty parameter file as command-line arguments (as with previous program)