Numerical Modelling in FORTRAN day 5

Paul Tackley, 2020

Today's Goals

1. A few useful things
2. Learn derived variable **type**s
3. Learn several other new things: save, forall, do concurrent, where, character manipulation, keyword arguments and optional arguments.
4. Practice, practice!

Today‘s Practice

1. Learn about **iterative** solvers for boundary value problems, including the **multigrid** method
2. Practice, practice! This time programming an iterative solver for **Poisson’s** equation.
   - Useful for e.g., gravitational potential, electromagnetism, convection (streamfunction-vorticity formulation)

Feedback on homeworks

- Random_number() can fill whole arrays with different random numbers, e.g.

```
allocate(a(50,50))
call random_number(a)
```

- Loops are not necessary!

~~do i=1,50; do j=1,50~~
~~ call random_number(a(i,j))~~
~~end do; end do~~

Array manipulation in f90-

- Like MATLAB, you 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
```

To avoid the boundary points:

```
T(2:nx-1,2:ny-1) = T (2:nx-1,2:ny-1) + &
                        dt*d2T (2:nx-1,2:ny-1)
```

Array manipulation in f90-

- Built-in array functions:
- sum(array): the sum of all elements of array (=> a scalar)
- product(array): the product of all elements of array (=>a scalar)
- matmul(matrix1,matrix2): matrix multiplication of the two matrices (=> a matrix)
- dot_product(vec1,vec2): the dot product of two vectors (=> a scalar)

Derived data types

- You can define new data types that consist of combinations of existing data types.
- They can also contain procedures (i.e. subroutines/functions) that operate on the new data type.

new type containing several already-defined types

array of new type

Access parts using %

```
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
```

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

internal subroutine inherits variables from containing routine

```
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%nx; 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
```

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
- If you want to input only some parts, declare normal variables to input them into, then copy to the derived type.

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)

Do concurrent (f2008): alternative to **do** loops

- e.g.,

```
do concurrent (i=1:nx,j=1:ny)
    T(i,j)=sin(C*(i-1))
end do
```

- Order of indices must not matter => suitable for parallel computation
- Optional mask, e.g.,

```
do concurrent(i=1:nx,j=1:ny,C(i,j)==0.)
```

- Advantages (i) can replace nested loops with a single loop (ii) parallelisation

forall (f95): similar but now officially obsolescent

- forall() has basically the same syntax as do concurrent(), with the addition of a single-line version, e.g.,

```
forall (i=1:nx,j=1:ny) T(i,j)=sin(C*(i-1))
```

but some limitations that make it more difficult to parallelise

=> use do concurrent for new programs.

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))
  do concurrent(i=1:n)
     y(i) = ((i-1)*h)**2
  end do

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


contains

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

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

end program SecDeriv
```

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
```

where

- Program output:

```
 1.00000000       1.00000000       0.417891920       1.00000000       1.00000000       2.17741132E-02   1.00000000
 8.94768834E-02   1.00000000       0.398063660       1.00000000       1.00000000      0.190855205       0.309326708
 1.00000000       1.00000000        1.00000000       1.00000000       5.62288761E-02  0.408534586       0.207549155
 1.00000000       1.00000000        1.00000000      0.178424537

0.573729157       0.00000000       8.01521540E-02   0.167172164       0.00000000      0.345419466       0.273811072
 0.00000000       0.00000000       0.00000000       0.381393850      0.676242709       0.00000000        0.00000000
0.661335111      0.473338306       0.00000000       0.342708558       0.00000000      0.345400393       0.165343955
 0.00000000       0.00000000       0.00000000       0.479803115
```

Top numbers: any above 0.5 have been rounded to 1.0

Bottom numbers: negative ones rounded to 0.0, positive ones square rooted.

Character string manipulation

- // concatonates strings
- access substrings using string(3:5)
- **len**(string): gives length of string
- **index**(string,sub) - location of a substring in another 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
```

Character string examples

- Program output:

```
          13
 one
           5


!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~????????????????????????????
????????????????????????????????????????????????????????
??????????????????????????
          97
 MyFile99
```

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.

```
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
```

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)
```

- Better to use allocatable arrays

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

Review last week: Advection-diffusion for fixed flow field

(i) Calculate velocity at each point using centered derivatives

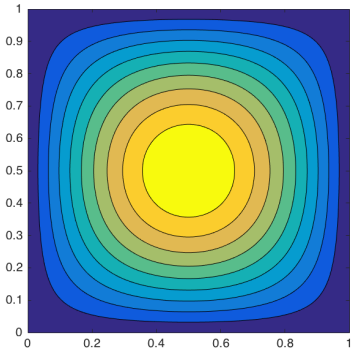
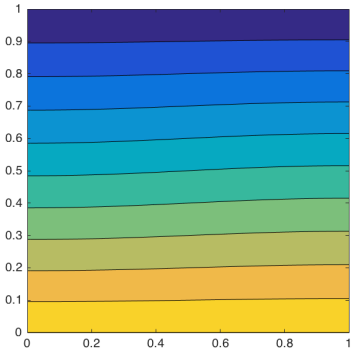
$$\left(v_x, v_y\right) = \left(\frac{\partial \psi}{\partial y}, -\frac{\partial \psi}{\partial x}\right)$$

(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

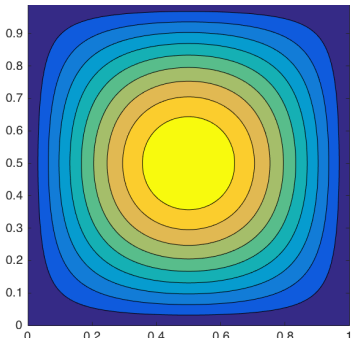
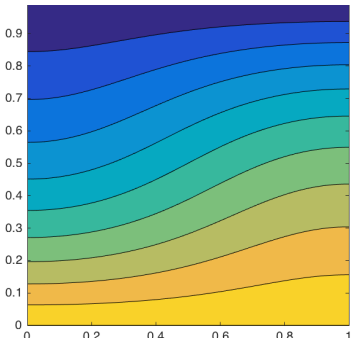
$$\frac{\partial T}{\partial t} = -v_x \frac{\partial T}{\partial x} - v_y \frac{\partial T}{\partial y} + \nabla^2 T$$

This is what it should look like

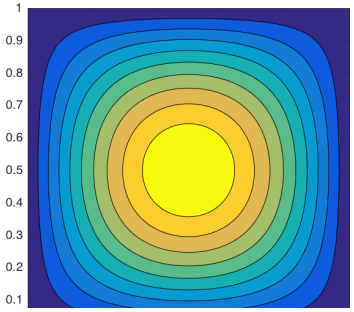
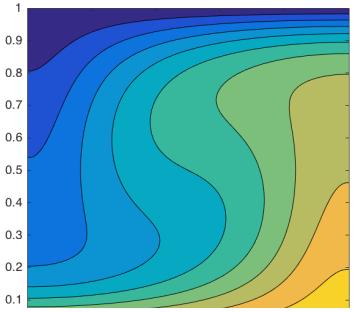
B=0.1

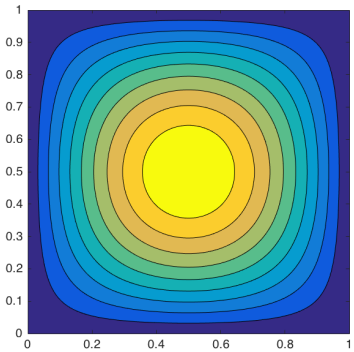
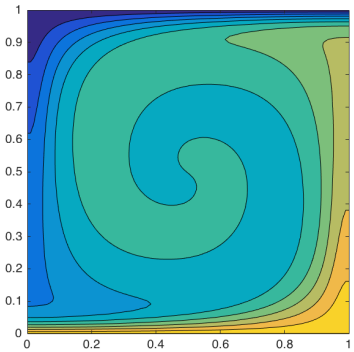
B=1

B=10

B=100

The next step: 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):

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

Substituting the streamfunction for velocity, we get:

$$\nabla^4 \psi = -Ra \frac{\partial T}{\partial x}$$

writing as 2 Poisson equations:

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

the **streamfunction-vorticity** formulation

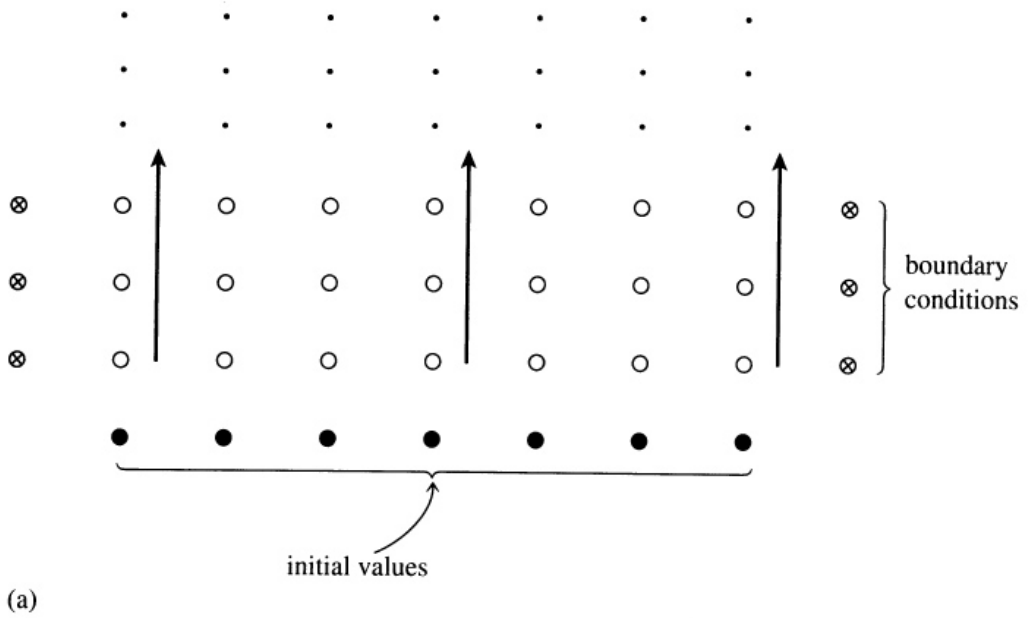
we need a Poisson solver

- An example of a **boundary value problem** (uniquely determined by interior equations and values at boundaries), as compared to
- **initial value problems** (depend on initial conditions as well as boundary conditions, like the diffusion equation)

Siméon Denis Poisson (1781-1840)

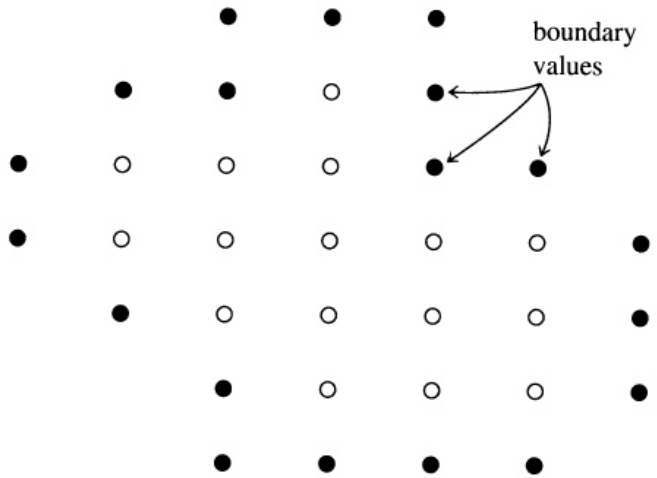
Initial value vs boundary value problems ...often, the problem is both

- (a) initial value problem



(a)

- (b) Boundary value problem



(b)

Example: 1D Poisson

Poisson: $\nabla^2 u = f$ (Solve for u; f fixed)

In 1-D: $\dfrac{\partial^2 u}{\partial x^2} = f$

Finite-difference form: $\dfrac{1}{h^2}\left(u_{i-1} - 2u_i + u_{i+1}\right) = f_i$

Example with 5 grid points:

$$u_0 = 0$$

$$u_0 - 2u_1 + u_2 = h^2 f_1$$

$$u_1 - 2u_2 + u_3 = h^2 f_2$$

$$u_2 - 2u_3 + u_4 = h^2 f_3$$

$$u_4 = 0$$

Problem: simultaneous solution needed

Ways to solve Poisson's equation

- **Problem**: A large number of finite-difference equations must be solved simultaneously
- **Method 1. Direct**
  - Put finite-difference equations into a matrix and call a subroutine to find the solution
  - Pro: get the answer in one step
  - Cons: for large problems
    - matrix very large (nx*ny)^2
    - solution very slow: time~(nx*ny)^3
- **Method 2. Iterative**
  - Start with initial guess and keep improving it until it is "good enough"
  - Pros: for large problems
    - Minimal memory needed.
    - Fast if use multigrid method: time~(nx*ny)
  - Cons: Slow if don't use multigrid method

Direct (matrix) method

- Example with 3 equations:

$$a_1x + b_1y + c_1z = d_1$$
$$a_2x + b_2y + c_2z = d_2$$
$$a_3x + b_3y + c_3z = d_3$$

- Can be written as:

$$\begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix}$$

- Our 5 equations:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} 0 \\ h^2 f_1 \\ h^2 f_2 \\ h^2 f_3 \\ 0 \end{bmatrix}$$

- There are standard libraries to solve such systems of equations, for example UMFPACK, as used by MATLAB (u=M\f)

Ways to solve Poisson's equation

- **Problem**: A large number of finite-difference equations must be solved simultaneously
- **Method 1. Direct**
  - Put finite-difference equations into a matrix and call a subroutine to find the solution
  - Pro: get the answer in one step
  - Cons: for large problems
    - matrix very large (nx*ny)^2
    - solution very slow: time~(nx*ny)^3
- **Method 2. Iterative**
  - Start with initial guess and keep improving it until it is "good enough"
  - Pros: for large problems
    - Minimal memory needed.
    - Fast if use multigrid method: time~(nx*ny)
  - Cons: Slow if don't use multigrid method

Iterative (Relaxation) Methods

- An alternative to using a direct matrix solver for sets of coupled PDEs
- Start with ‘guess’, then iteratively improve it
- Approximate solution ‘relaxes’ to the correct numerical solution
- Stop iterating when the error (‘residue’) is small enough

Why?

- Storage:
  - Matrix method has large storage requirements: (#points)^2. For large problems, e.g., 1e6 grid points, this is impossible!
  - Iterative method just uses #points
- Time:
  - Matrix method takes a long time for large #points: scaling as N^3 operations
  - The iterative **multigrid** method has #operations scaling as N

Example: 1D Poisson

Poisson: $\nabla^2 u = f$ In 1-D: $\frac{\partial^2 u}{\partial x^2} = f$

Finite-difference form: $\frac{1}{h^2}\left(u_{i-1} - 2u_i + u_{i+1}\right) = f_i$

Assume we have an approximate solution $\tilde{u}_i$

The error or residue: $R_i = \frac{1}{h^2}\left(\tilde{u}_{i-1} - 2\tilde{u}_i + \tilde{u}_{i+1}\right) - f_i$

Now calculate correction to $\tilde{u}_i$ to reduce residue

Correcting $\tilde{u}_i$

From the residue equation note that: $\frac{\partial R_i}{\partial \tilde{u}_i} = \frac{-2}{h^2}$

So adding a correction $+\frac{1}{2}h^2 R_i$ to $\tilde{u}_i$ should zero R

i.e., $\tilde{u}_i^{n+1} = \tilde{u}_i^n + \alpha \frac{1}{2} h^2 R_i$

Unfortunately it doesn't zero R because the surrounding points also change, but it does reduce R

α is a 'relaxation parameter' of around 1:
$\alpha > 1$ => 'overrelaxation'
$\alpha < 1$ => 'underrelaxation'

2 types of iterations: Jacobi & Gauss-Seidel

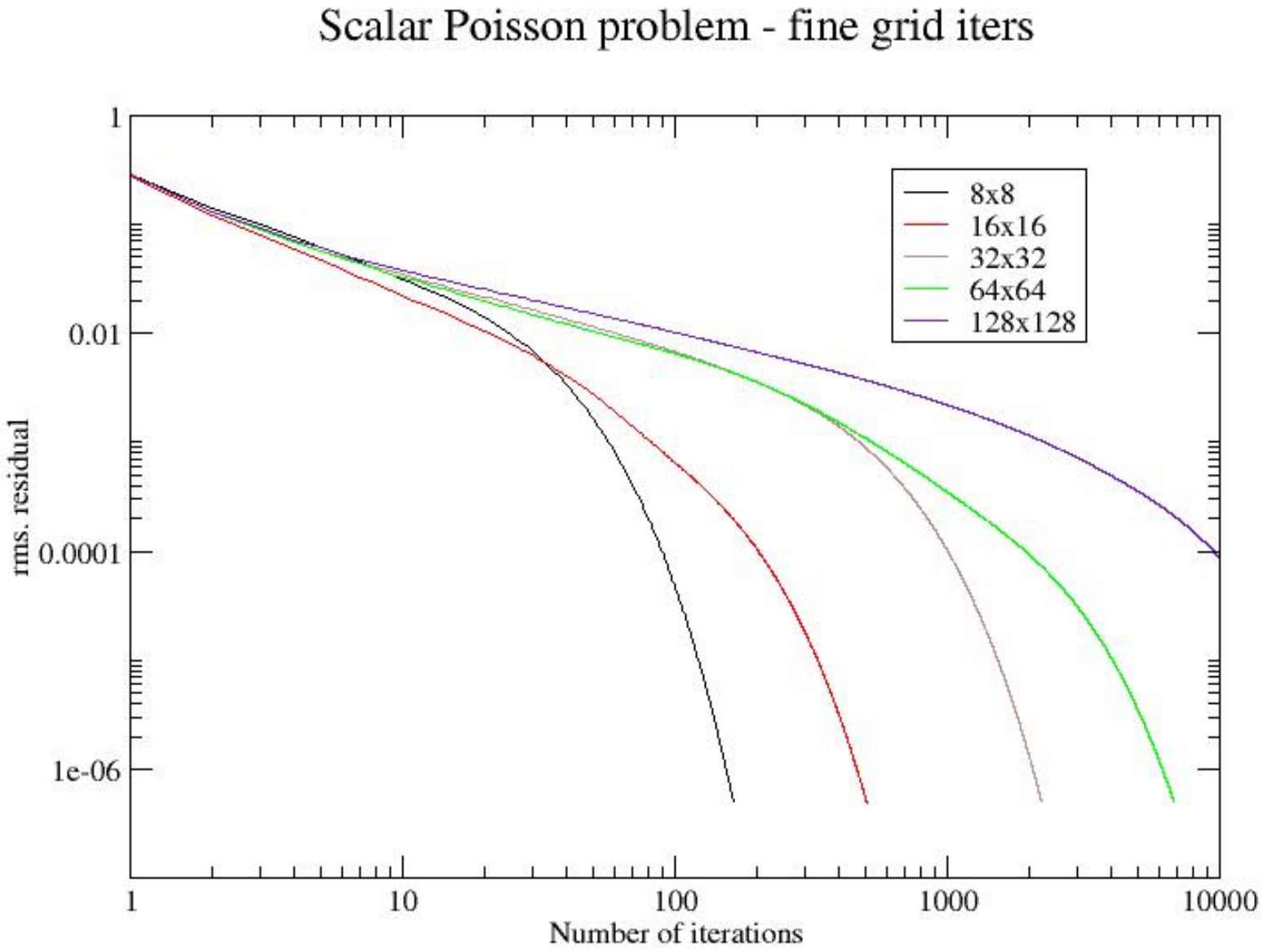
- Gauss-Seidel: update point at same time as residue calculation
  - do (all points)
  - residue=...
  - u(i,j) = u(i,j) + f(residue)
  - end do
- Jacobi: calculate all residues first then update all points
  - do (all points)
  - residue(i,j)=...
  - end do
  - do (all points)
  - u(i,j) = u(i,j) + f(residue(i,j))
  - end do

Use Gauss-Seidel or Jacobi ?

- Gauss-Seidel converges faster and does not require storage of all points' residue
- alpha>1 (over-relaxation) can be used for GS, but <1 required for J to be stable.
- For our multigrid program, optimal αlpha is about 1 for GS, 0.7 for Jacobi
- Conclusion: use Gauss-Seidel iterations

Demonstration of 1-D relaxation using Matlab: Things to note

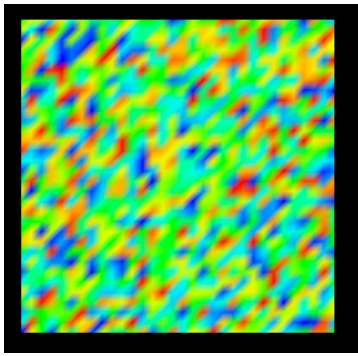
- The residue becomes smooth as well as smaller (=>short-wavelength solution converges fastest)
- #iterations increases with #grid points. How small must R be for the solution to be ‘good enough’ (visually)?
- Effect of α :
  - smaller => slower convergence, smooth residue
  - larger => faster convergence
  - too large => unstable



- Higher N => slower convergence

Now 2D Poisson's eqn.

$$\nabla^2 u = f$$



Finite-difference approximation:

$$\frac{1}{h^2}\left(u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j} - 4u_{i,j}\right) = f_{ij}$$

Use iterative approach=>start with u=0, sweep through grid updating u values according to:

$$\tilde{u}_{ij}^{n+1} = \tilde{u}_{ij}^{n} + \alpha R_{ij}\frac{h^2}{4}$$

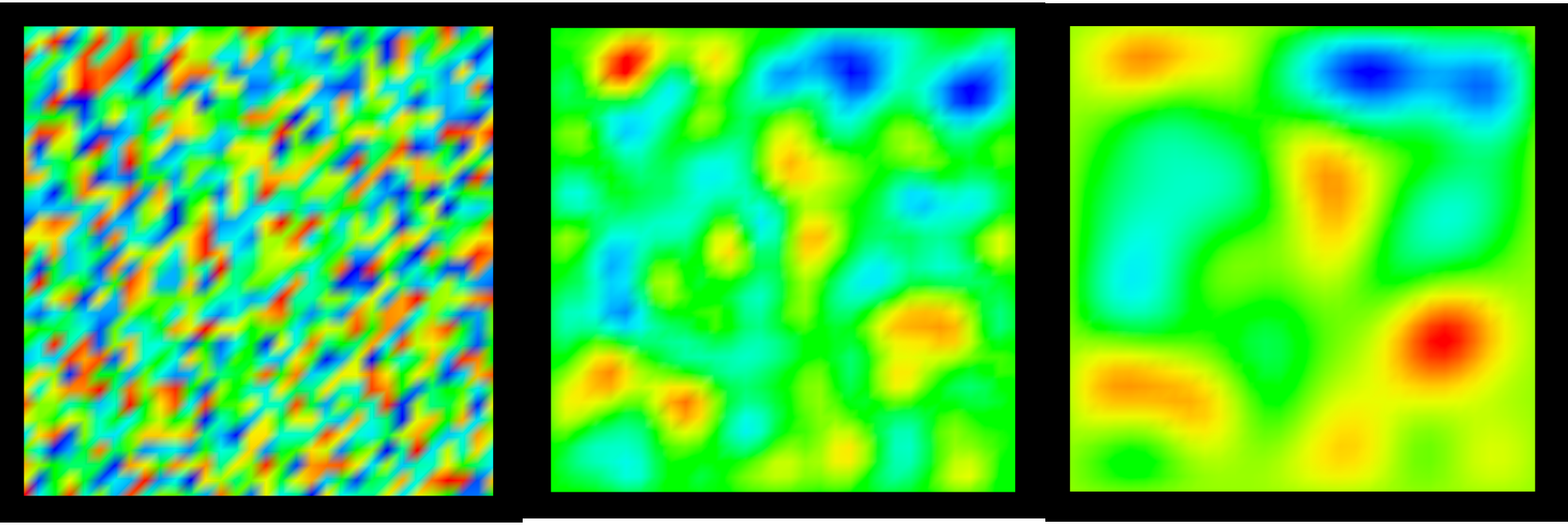
Where Rij is the **residue** ("error"): $R = \nabla^2 \tilde{u} - f$

Residue after repeated iterations

Start
rms residue=0.5

5 iterations
Rms residue=0.06

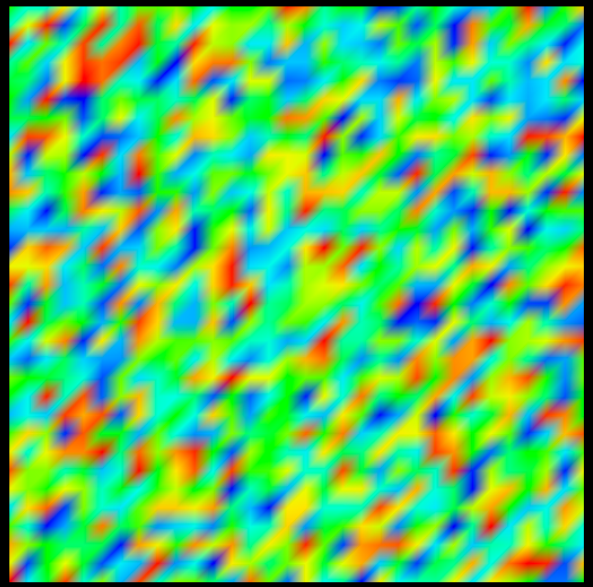
20 iterations
Rms residue=0.025



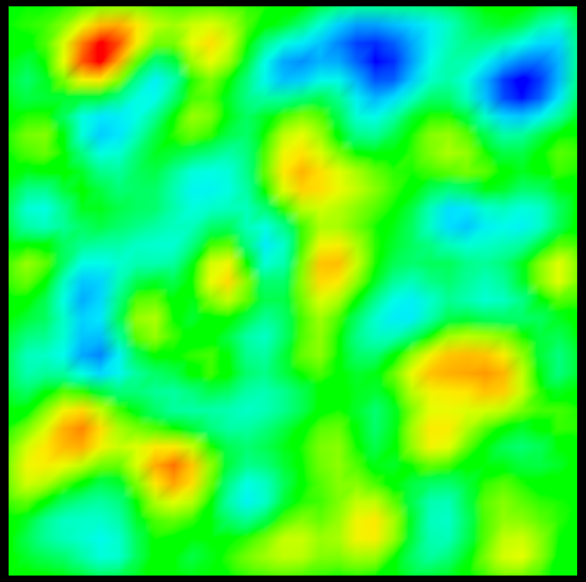
Residue gets **smoother** => iterations are like a diffusion process

Iterations smooth the residue
=>**solve R on a coarser grid**
=>faster convergence

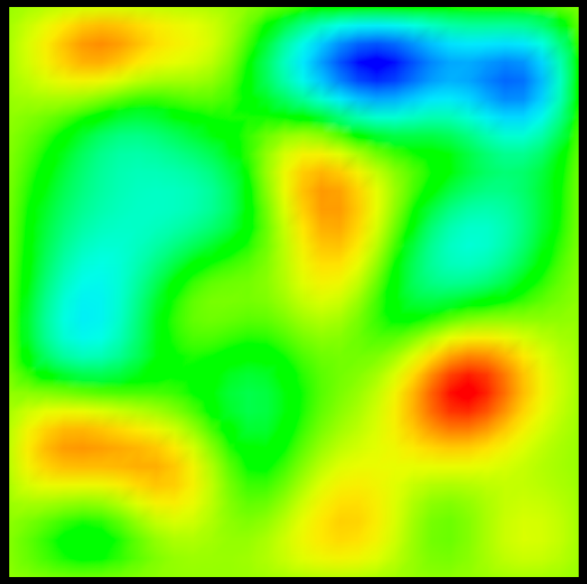
Start
rms residue=0.5



5 iterations
Rms residue=0.06



20 iterations
Rms residue=0.025



2-grid Cycle

- Several iterations on the fine grid
- Approximate (“**restrict**”) R on coarse grid
- Find coarse-grid solution to R (=correction to u)
- Interpolate (“**prolongate**”) correction=>fine grid and add to u
- Repeat until low enough R is obtained

Coarsening: vertex-centered vs. cell-centered

We will use the grid on the left. Number of points has to be a power-of-two plus 1, e.g., 5,9,17,33,65,129, 257,513,1025, 2049, 4097, ...

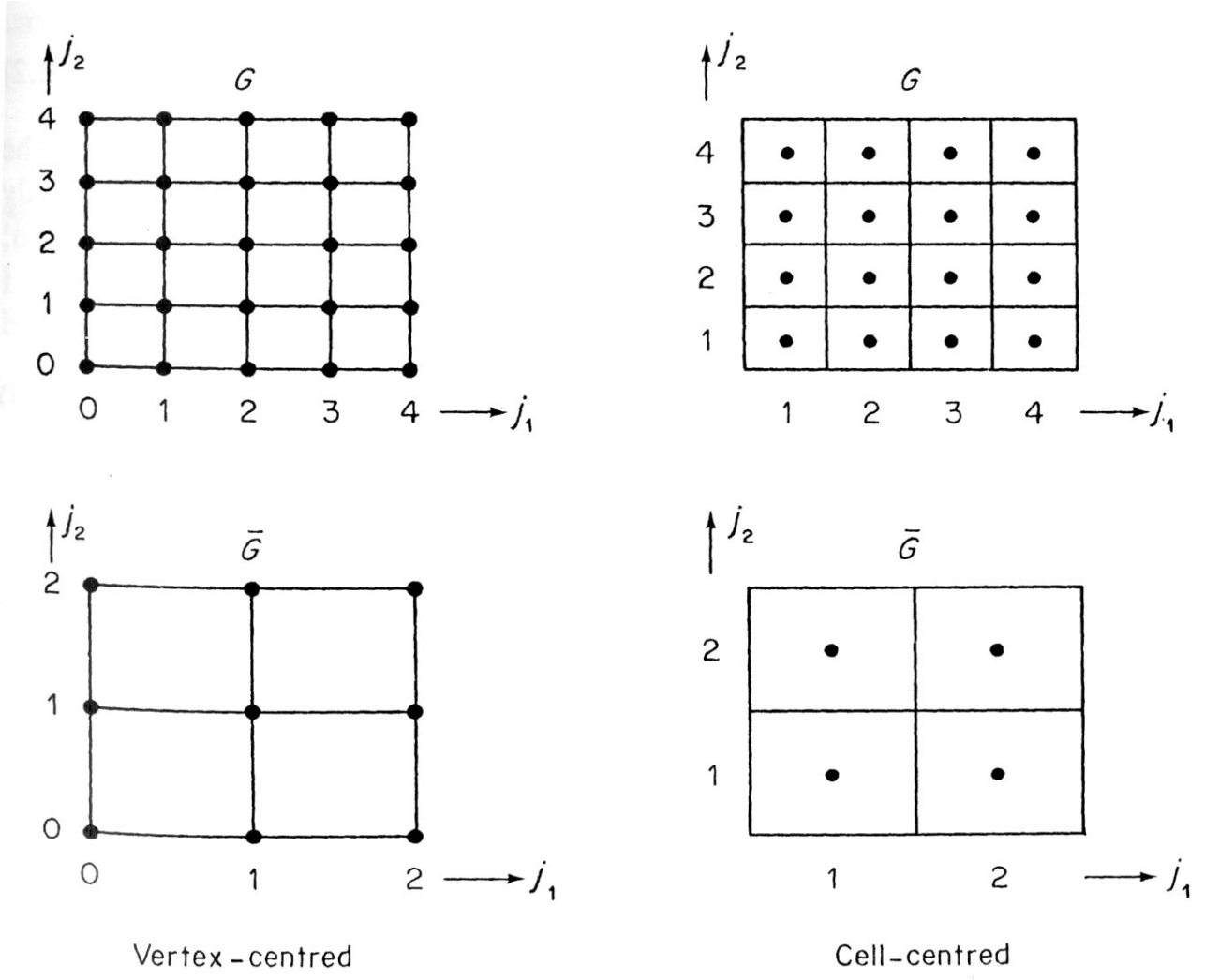


Figure 5.1.2 Vertex-centred and cell-centred coarsening in two dimensions. (● grid points.)

Multigrid cycle

- Start as 2-grid cycle, but keep going to coarser and coarser grids, on each one calculating the correction to the residue on the previous level
- Exact solution on coarsest grid (~ few points in each direction)
- Go from coarsest to finest level, at each step interpolating the correction from the next coarsest level and taking a few iterations to smooth this correction
- All lengthscales are relaxed @ the same rate!

V-cycles and W-cycles

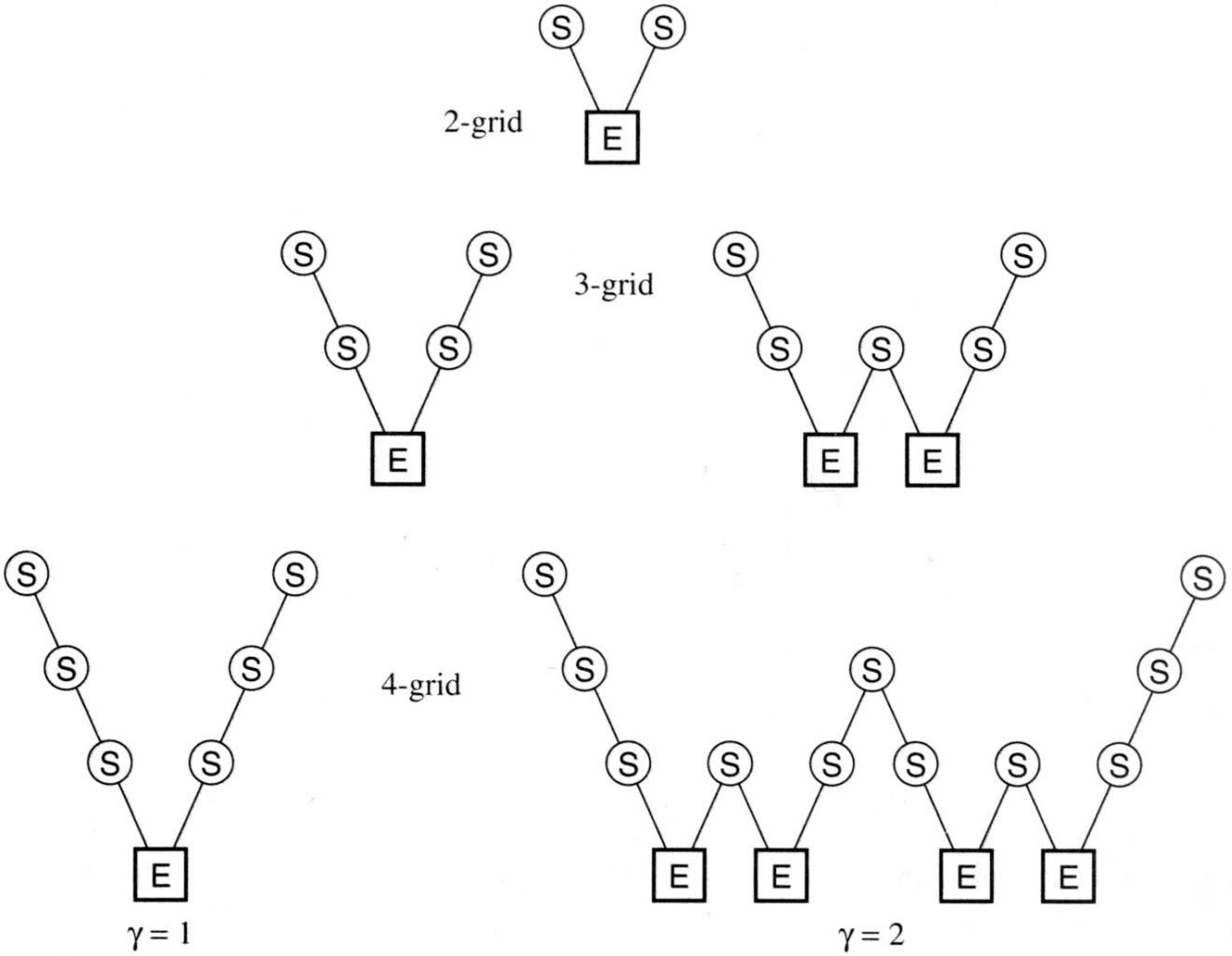
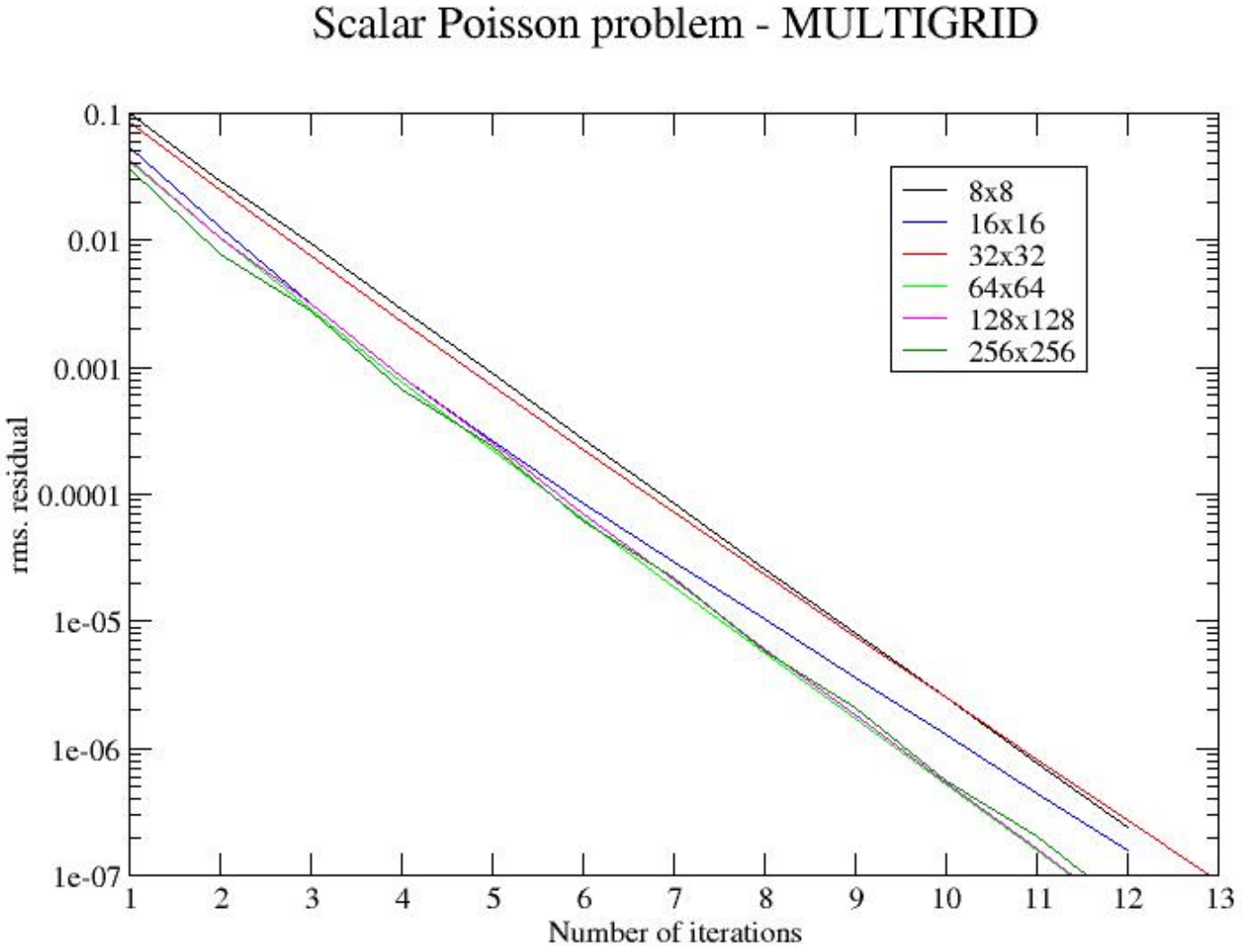


Figure 19.6.1. Structure of multigrid cycles. S denotes smoothing, while E denotes exact solution on the coarsest grid. Each descending line \ denotes restriction ($\mathcal{R}$) and each ascending line / denotes prolongation ($\mathcal{P}$). The finest grid is at the top level of each diagram. For the V-cycles ($\gamma = 1$) the E step is replaced by one 2-grid iteration each time the number of grid levels is increased by one. For the W-cycles ($\gamma = 2$), each E step gets replaced by two 2-grid iterations.



- Convergence rate **independent of grid size**
- =>#operations scales as #grid points

Programming multigrid V-cycles

- You are given a function that does the steps in the V-cycle
- The function is recursive, i.e., it calls itself. It calls itself to find the correction at the next coarser level.
- It calls various functions that you need to write: doing an iteration, calculating the residue, restrict or prolongate a field
- Add these functions and make it into a module
- Boundary conditions: zero

```
recursive function Vcycle_2DPoisson(u_f,rhs,h) result (resV)
  implicit none
  real resV
  real,intent(inout):: u_f(:,:)  ! arguments
  real,intent(in)   :: rhs(:,:),h
  integer           :: nx,ny,nxc,nyc, i,j  ! local variables
  real,allocatable:: res_c(:,:),corr_c(:,:),res_f(:,:),corr_f(:,:)
  real              :: alpha=1.0, res_rms

  nx=size(u_f,1); ny=size(u_f,2)  ! must be power of 2 plus 1
  nxc=1+(nx-1)/2; nyc=1+(ny-1)/2  ! coarse grid size

  if (min(nx,ny)>5) then  ! not the coarsest level

     allocate(res_f(nx,ny),corr_f(nx,ny), &
          corr_c(nxc,nyc),res_c(nxc,nyc))

     !---------- take 2 iterations on the fine grid-------------
     res_rms = iteration_2DPoisson(u_f,rhs,h,alpha)
     res_rms = iteration_2DPoisson(u_f,rhs,h,alpha)

     !---------- restrict the residue to the coarse grid --------
     call residue_2DPoisson(u_f,rhs,h,res_f)
     call restrict(res_f,res_c)

     !---------- solve for the coarse grid correction -----------
     corr_c = 0.
     res_rms = Vcycle_2DPoisson(corr_c,res_c,h*2) ! *RECURSIVE CALL*

     !---- prolongate (interpolate) the correction to the fine grid
     call prolongate(corr_c,corr_f)

     !---------- correct the fine-grid solution -----------------
     u_f = u_f - corr_f

     !---------- two more smoothing iterations on the fine grid---
     res_rms = iteration_2DPoisson(u_f,rhs,h,alpha)
     res_rms = iteration_2DPoisson(u_f,rhs,h,alpha)

     deallocate(res_f,corr_f,res_c,corr_c)

  else
```

```
      !---------- solve for the coarse grid correction -----------
      corr_c = 0.
      res_rms = Vcycle_2DPoisson(corr_c,res_c,h*2) ! *RECURSIVE CALL*

      !---- prolongate (interpolate) the correction to the fine grid
      call prolongate(corr_c,corr_f)

      !---------- correct the fine-grid solution -----------------
      u_f = u_f - corr_f

      !---------- two more smoothing iterations on the fine grid---
      res_rms = iteration_2DPoisson(u_f,rhs,h,alpha)
      res_rms = iteration_2DPoisson(u_f,rhs,h,alpha)

      deallocate(res_f,corr_f,res_c,corr_c)

   else

      !----- coarsest level (ny=5): iterate to get 'exact' solution

      do i = 1,100
         res_rms = iteration_2DPoisson(u_f,rhs,h,alpha)
      end do

   end if

   resV = res_rms   ! returns the rms. residue

end function Vcycle_2DPoisson
```

The use of **result** in functions

- Avoids use of the function name in the code. Instead, another variable name is used to set the result
- Required by some compilers for recursive functions
- Example see this code

```
recursive function Vcycle_2DPoisson(u_f,rhs,h) result (resV)
  implicit none
  real resV
  real,intent(inout):: u_f(:,:)  ! arguments
  real,intent(in)   :: rhs(:,:),h
  integer           :: nx,ny,nxc,nyc, i,j  ! local variables
  real,allocatable:: res_c(:,:),corr_c(:,:),res_f(:,:),corr_f(:,:)
  real              :: alpha=1.0, res_rms

  nx=size(u_f,1); ny=size(u_f,2)  ! must be power of 2 plus 1
  nxc=1+(nx-1)/2; nyc=1+(ny-1)/2  ! coarse grid size

  if (min(nx,ny)>5) then  ! not the coarsest level

     allocate(res_f(nx,ny),corr_f(nx,ny), &
          corr_c(nxc,nyc),res_c(nxc,nyc))




     !----- coarsest level (ny=5): iterate to get 'exact' solution

     do i = 1,100
        res_rms = iteration_2DPoisson(u_f,rhs,h,alpha)
     end do

  end if

  resV = res_rms   ! returns the rms. residue

end function Vcycle_2DPoisson
```

Exercise

- Make a Poisson solver module by adding necessary functions to the provided V-cycle function
- Write a main program that tests this, taking multiple iterations until the residue is less than about 1e-5 of the right-hand side f
  - NOTE: may need 64-bit precision to obtain this accuracy
- The program should have the option of calling the iteration function directly, or the V-cycle function, so that you can compare their performance

Functions to add

- function iteration_2DPoisson(u,f,h,alpha)
  - does one iteration on u field as detailed earlier
  - is a function that returns the rms. residue
- subroutine residue_2DPoisson(u,f,h,res)
  - calculates the residue in array res
- subroutine restrict(fine,coarse)
  - Copies every other point in fine into coarse
- subroutine prolongate(coarse,fine)
  - Copies coarse into every other point in fine
  - Does linear interpolation to fill the other points

Convergence criterion

Equation to solve: $\nabla^2 u = f$

Residue: $R = \nabla^2 \tilde{u} - f$

Iterate until $\frac{R_{rms}}{f_{rms}} < err$ Where err is e.g. 10^{-5}

Rms=root-mean-square: $f_{rms} = \sqrt{\frac{\sum \left(A_{ij} \right)^2}{n_x n_y}}$

Test program details

- Using namelist input, read in (see example par file)
  - nx,ny
  - Init (='random' or 'spike')
  - Multigrid. (=.true. or .false.; switches on multigrid)
  - alpha
- Initialise:
  - h=1/(ny-1)
  - source field f to random, spike, etc.
  - u(:,:)=0
- Repeatedly call either iteration_2DPoisson or Vcycle_2DPoisson until "converged" (according to rms. residue compared to rms. source field f)
- Write f and u to files for visualisation!

Provided par file relaxIN.txt

```
&in
        nx=129
        ny=257
        init='spike'
        alpha=0.7
        multigrid=.true.
/
```

Your program should run with this par file!

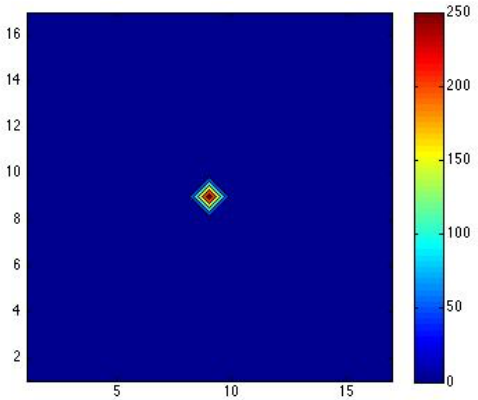
Tests

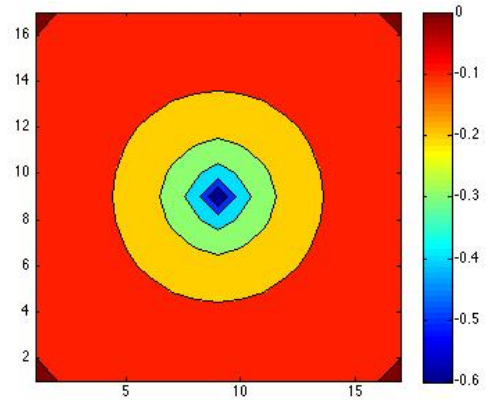
- Record number of iterations (V-cycles when using multigrid) needed for different grid sizes for each scheme, from 17x17 to at least 257x257
- What is the effect of alpha on iterations? (for multigrid it is hard-wired)
- For multigrid, what is the maximum number of grid points that fit in your computer, and how long does it take to solve the equations?

Example solutions

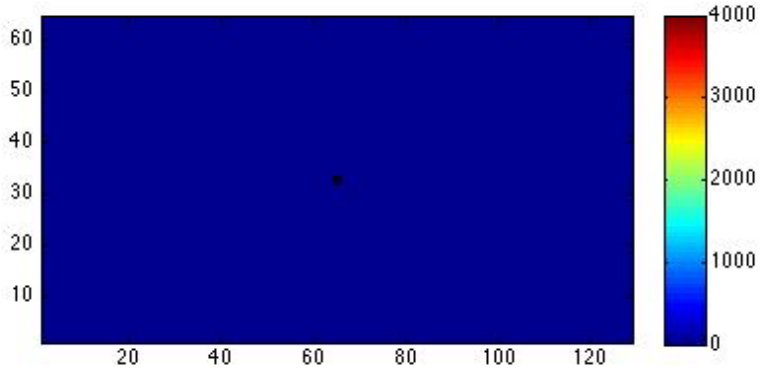
- For a delta function (spike) in the center of the grid, i.e.,
  - f(nx/2+1,ny/2+1)=1/dy**2, otherwise 0
  - (1/dy**2 is so that the integral of this=1)

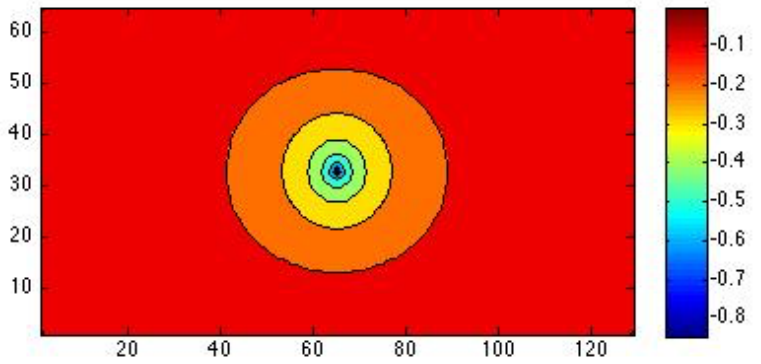
17x17 grid





129x65 grid





Hand in by email

- Your complete program (.f90 files)
- Results of your tests
  - #iterations or #V-cycles vs. grid size
  - effect of alpha
  - maximum #points
- Plots of your 2 solutions for a delta-function source