Numerical Modelling in FORTRAN day 6

Paul Tackley, 2014

Today's Goals

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

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

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

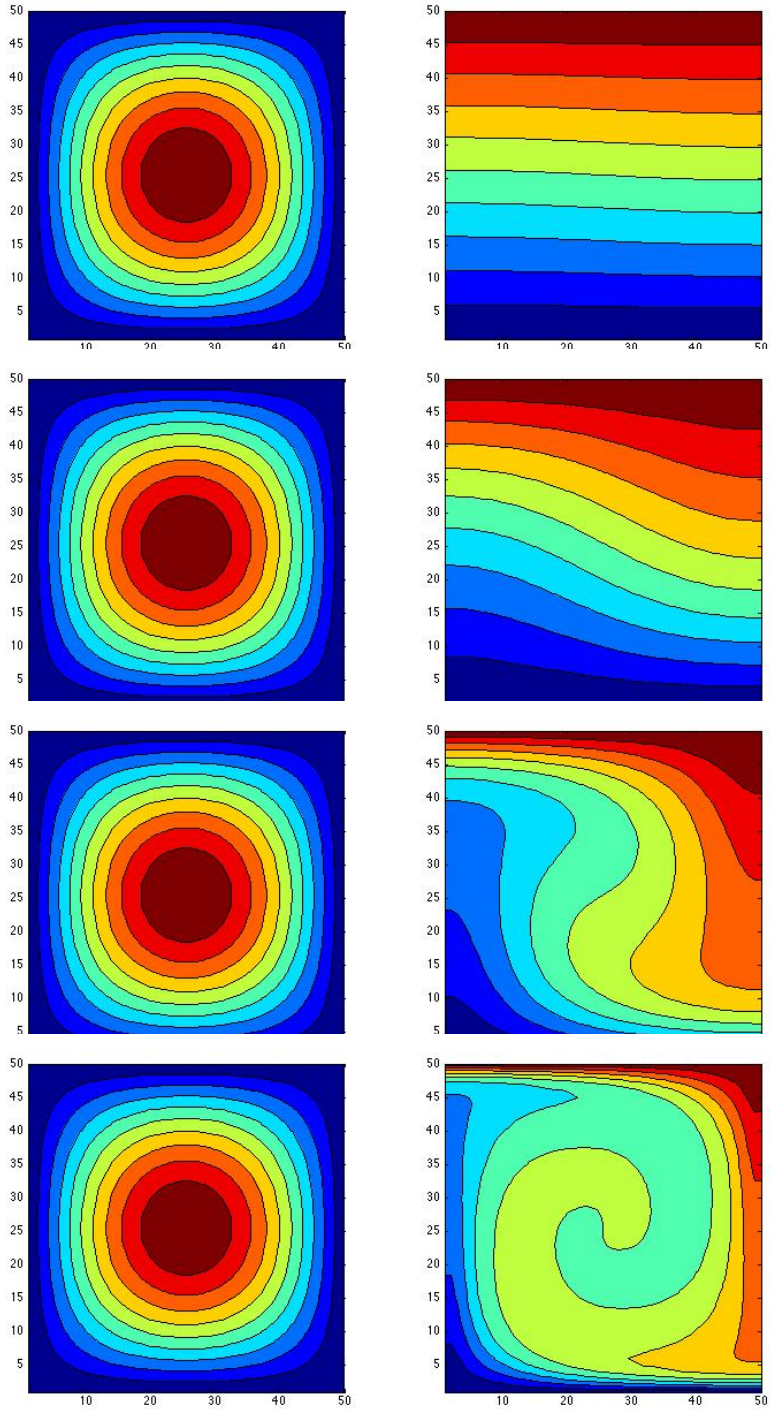
This is what it should look like

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

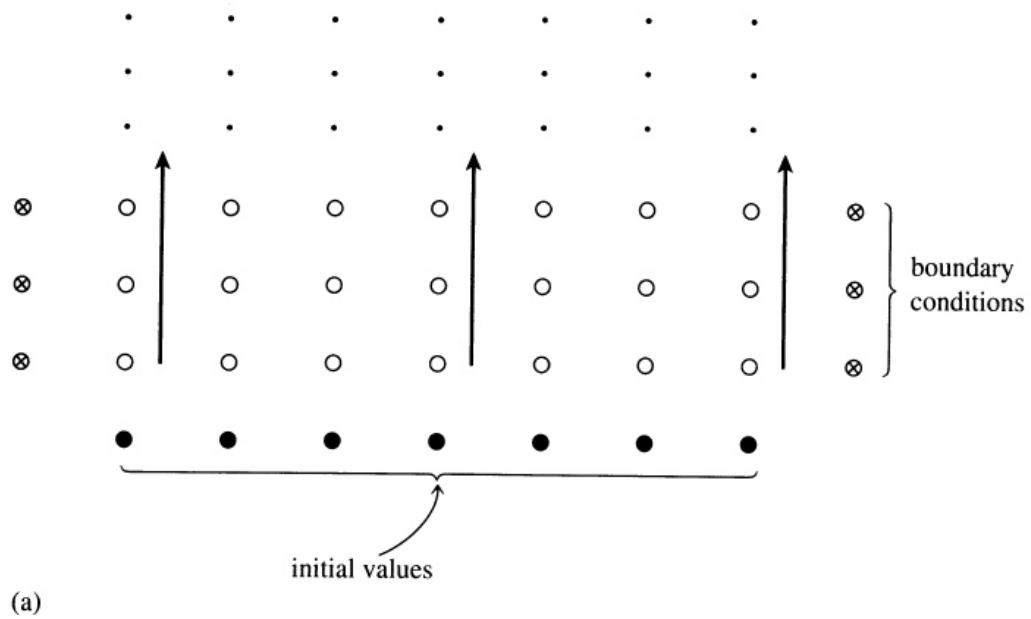
we need a Poisson solver

- 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)

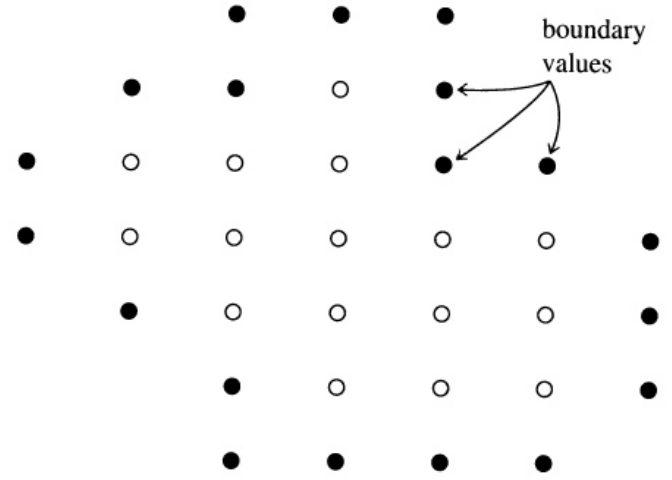
Initial value vs boundary value problems ...often, the problem is both

- (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 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

a is a 'relaxation parameter' of around 1:
a > 1 => 'overrelaxation'
a < 1 => 'underrelaxation'

2 types of iterations: Jacobi & Gauss-Seidel

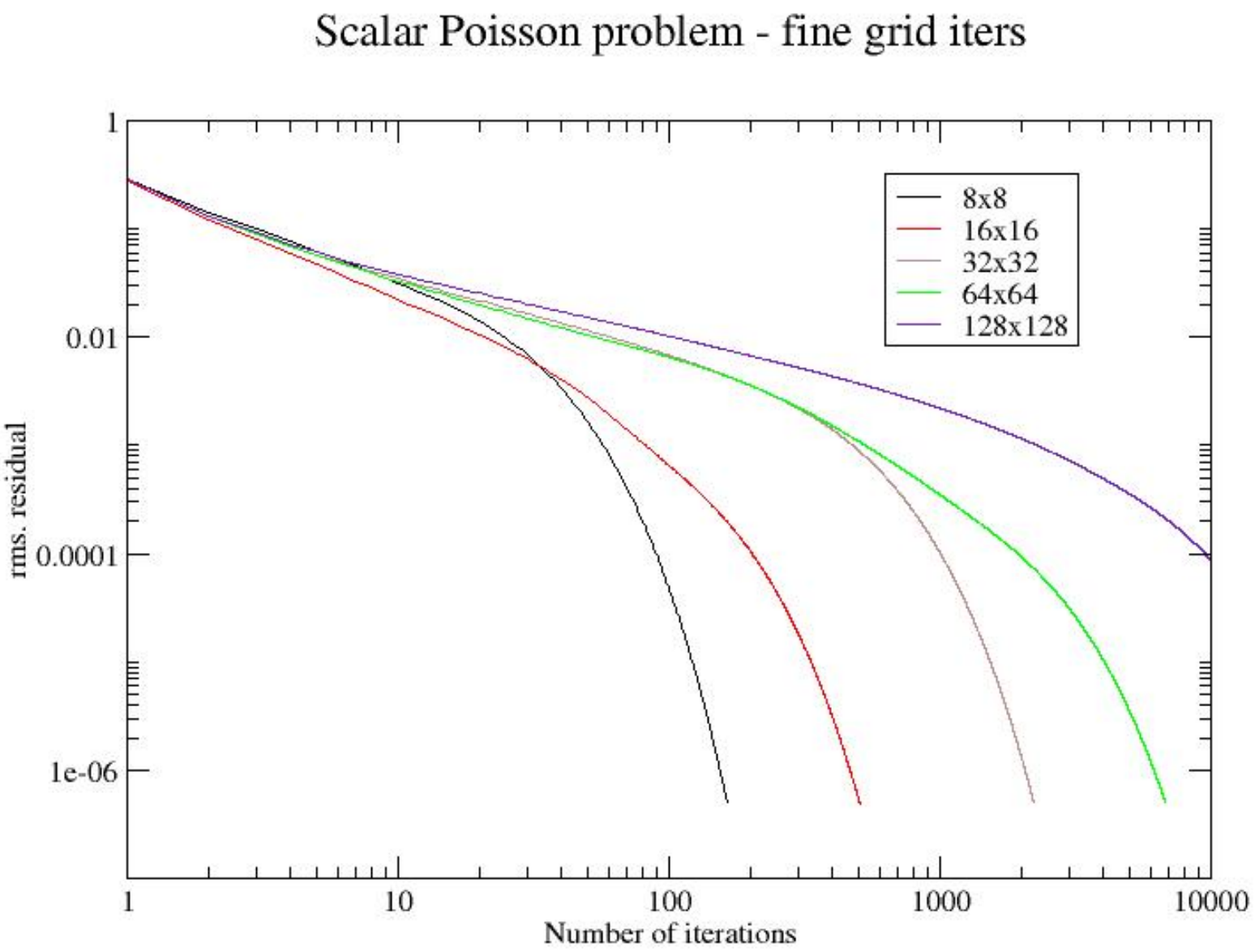
- Gauss-Seidel: update point at same time as residue calculation
  - do (all points)
  - residue=...
  - u(i,j) = u(i,j) + f(residue)
  - end do
- Jacobi: calculate all residues first then update all points
  - do (all points)
  - residue(i,j)=...
  - end do
  - do (all points)
  - u(i,j) = u(i,j) + 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 alpha about 1 for GS, 0.7 for Jacobi
- Only problem: GS not possible on multiple CPUs. Solution: red-black iterations.
- Conclusion: use Gauss-Seidel iterations

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

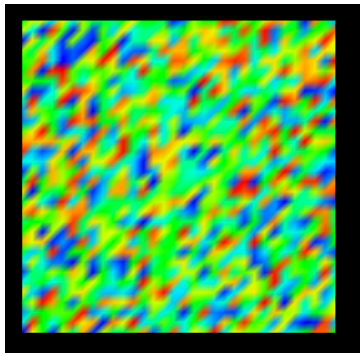
- The residue becomes smooth as well as smaller (=>short-wavelength solution converges fastest)
- #iterations increases with #grid points. How small must R be for the solution to be ‘good enough’ (visually)?
- Effect of α :
  - smaller => slower convergence, smooth residue
  - larger => faster convergence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



- Higher N => slower convergence

Now 2D Poisson 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:

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

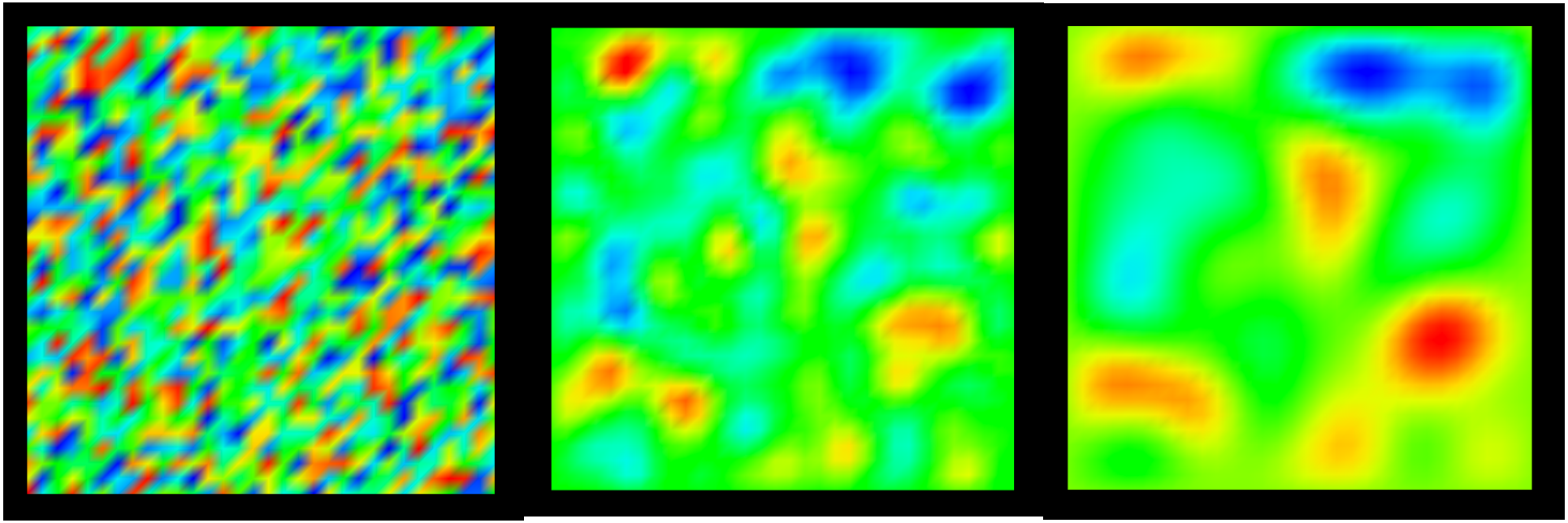
Where Rij is the **residue** (“error”): $R = \nabla^2 \tilde{u} - f$

Residue after repeated iterations

Start
rms residue=0.5

5 iterations
Rms residue=0.06

20 iterations
Rms residue=0.025



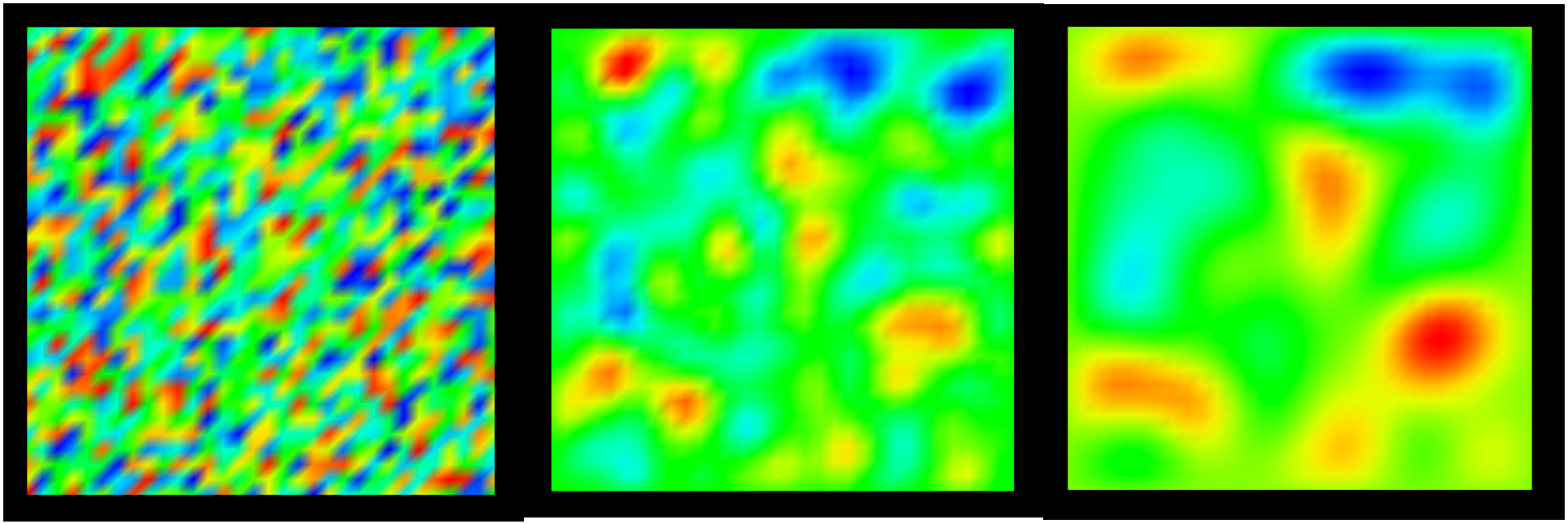
Residue gets **smoother** => iterations are like a diffusion process

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, ...

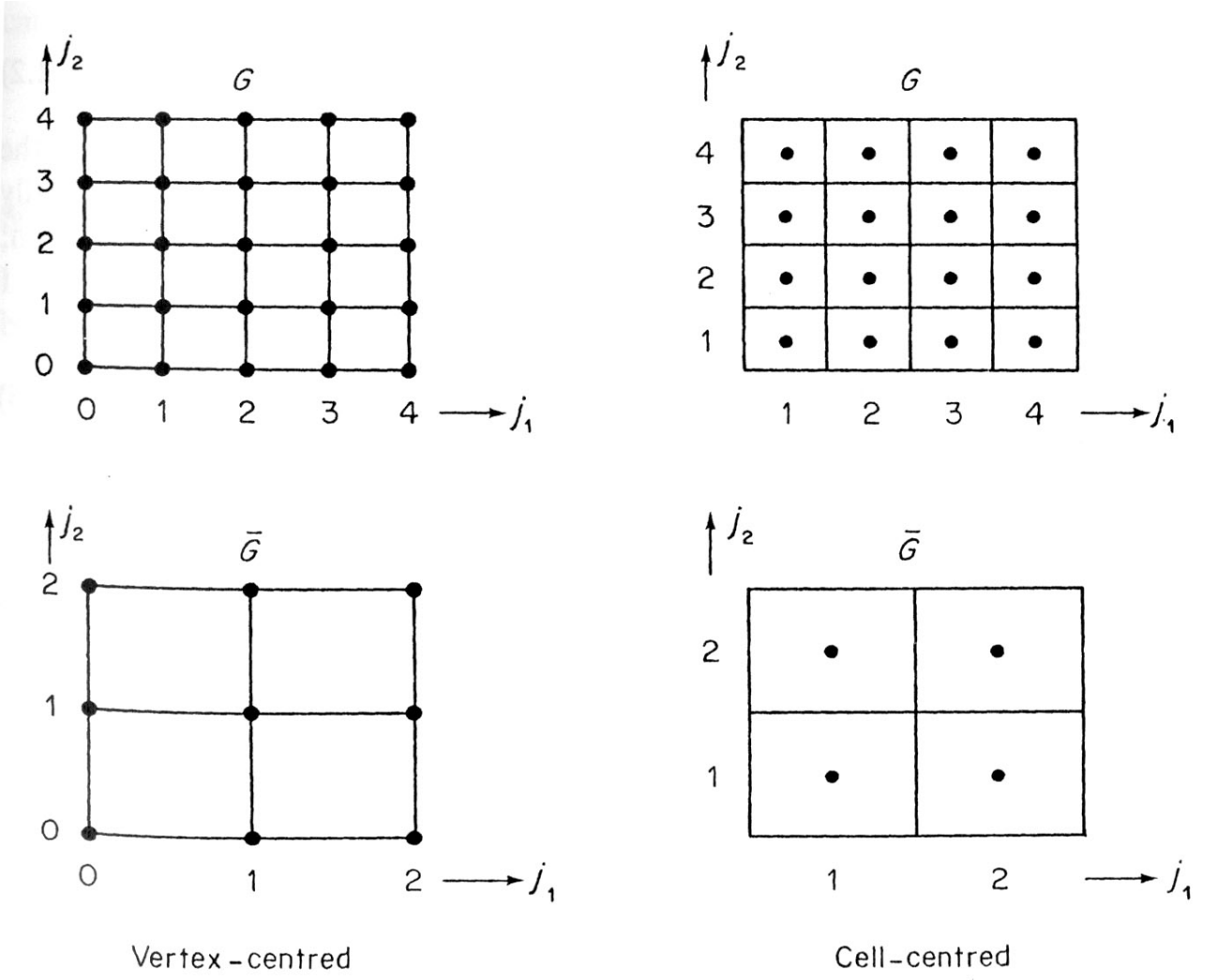


Figure 5.1.2 Vertex-centred and cell-centred coarsening in two dimensions. (● grid points.)

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

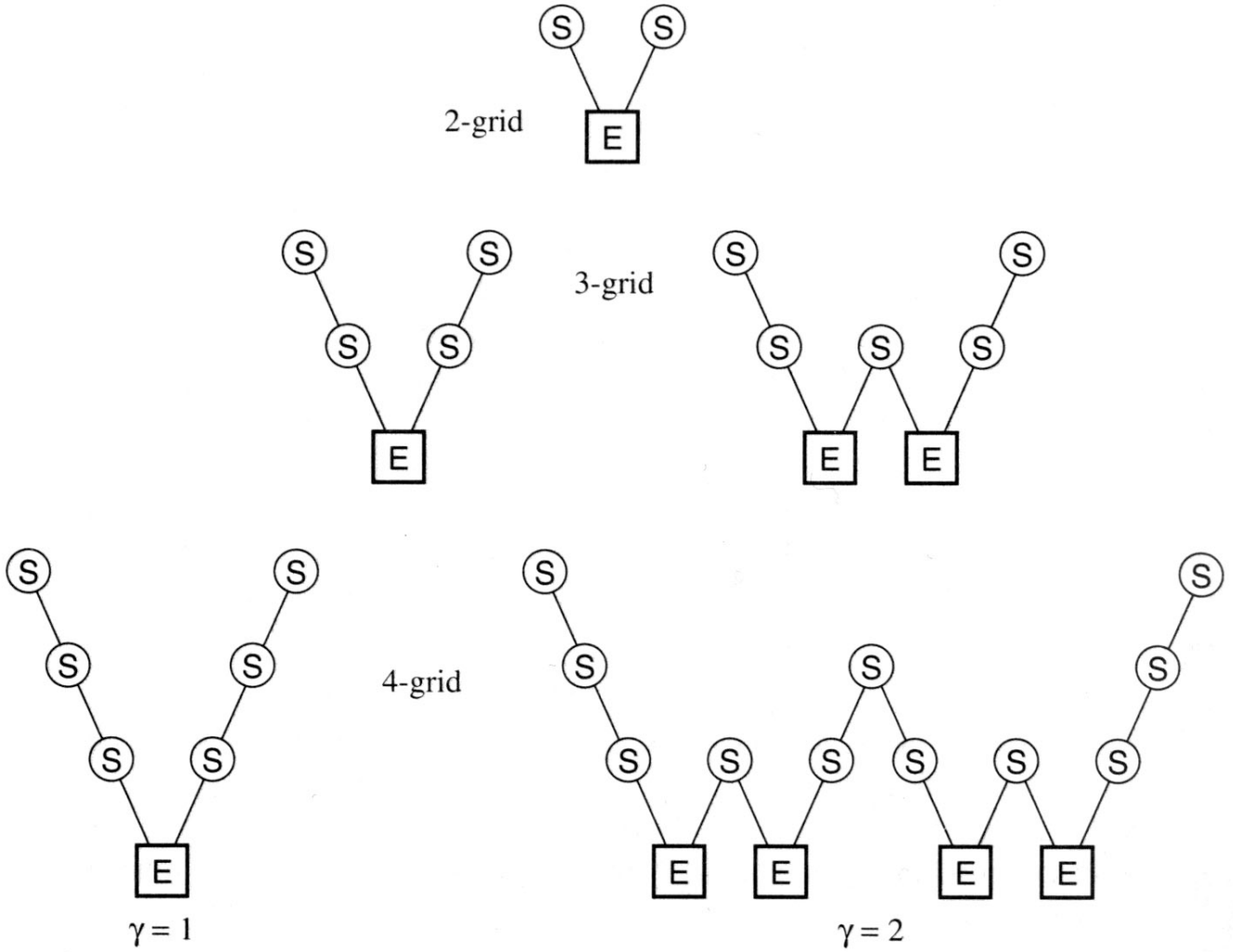
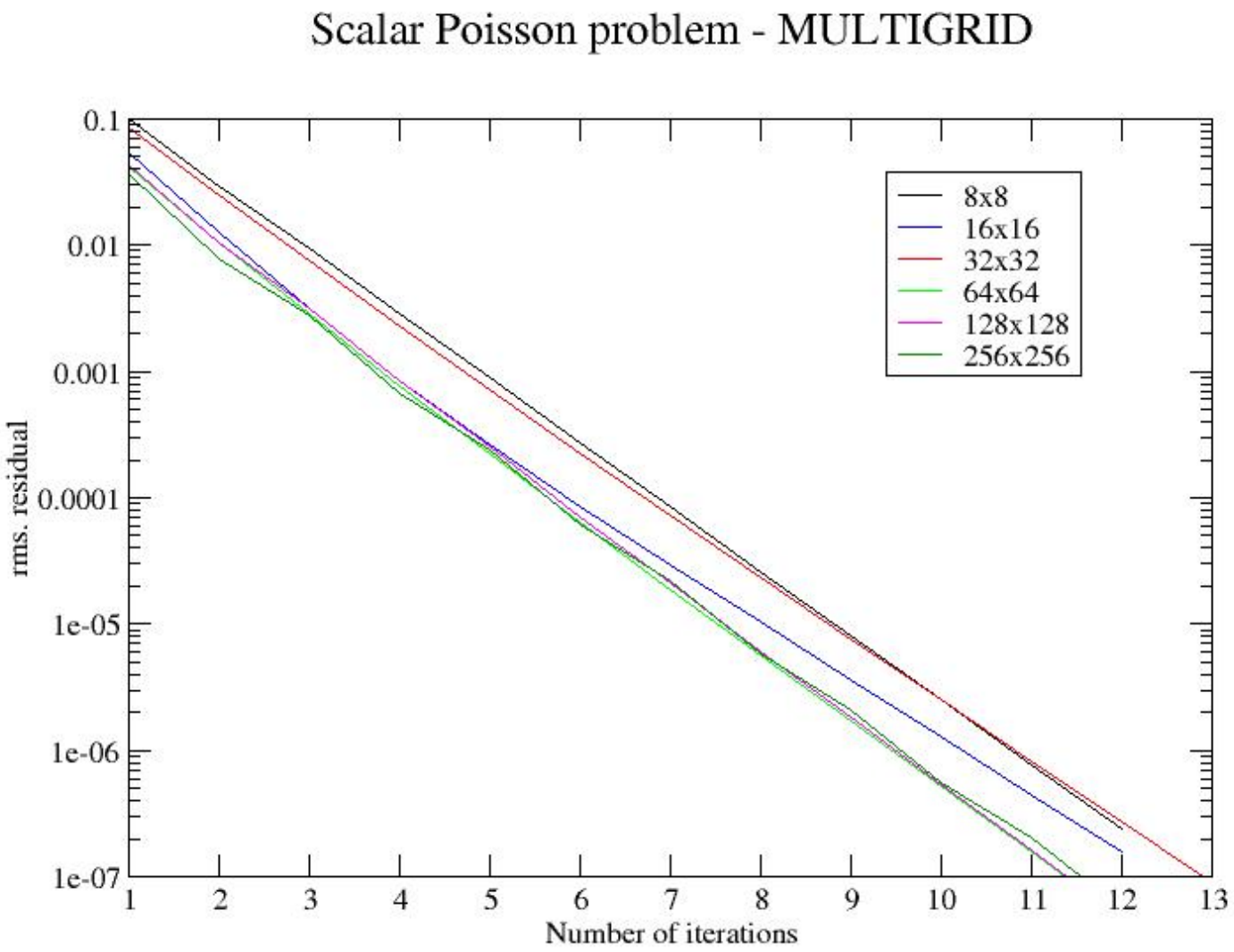


Figure 19.6.1. Structure of multigrid cycles. S denotes smoothing, while E denotes exact solution on the coarsest grid. Each descending line \ denotes restriction ($\mathcal{R}$) and each ascending line / denotes prolongation ($\mathcal{P}$). The finest grid is at the top level of each diagram. For the 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))

     !---------- 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)
```

```
     !----- 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 nx,ny, flags to indicate what type of source and iterations, and 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!

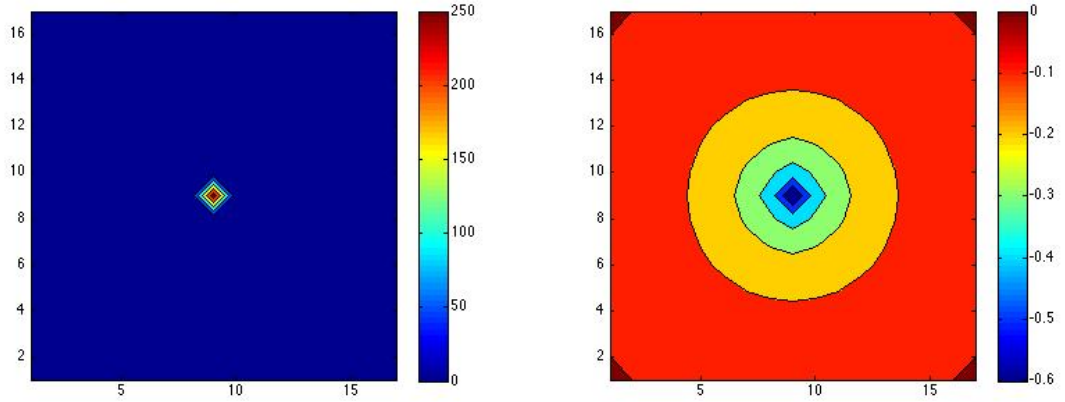
Tests

- Record number of iterations 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?

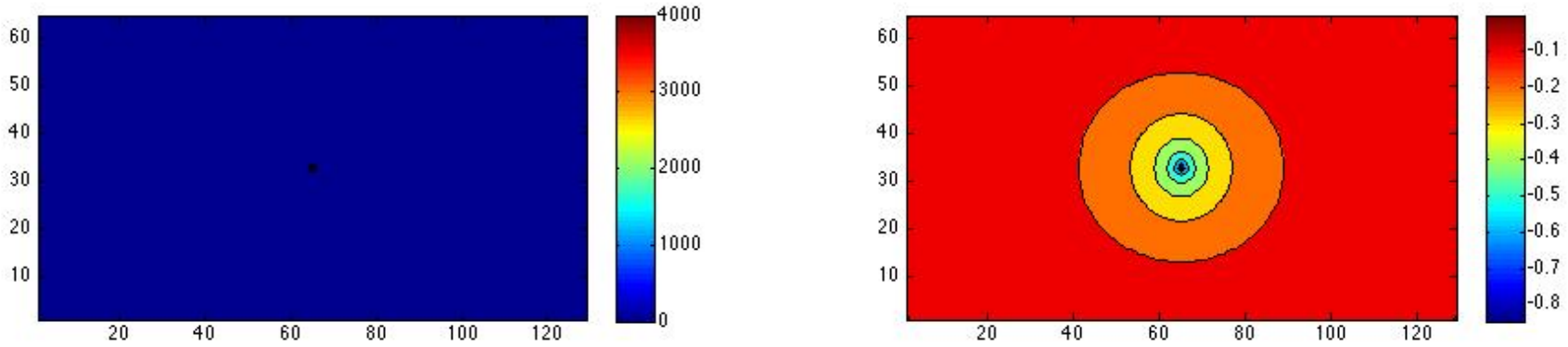
Example solutions

- 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
- Results of your tests
  - #iterations vs. grid size
  - effect of alpha
  - maximum #points)
- Plots of your solutions for a delta-function source