epcc

Parallel Numerical Algorithms

Solution of Boundary Value Problems

Overview of Lecture

- General solution methods
- Relaxation methods
  - Jacobi algorithm
  - testing for convergence
  - Gauss Seidel
  - over-relaxation
- Notes
  - parallelisation
  - non-linear equations
- Pollution problem
  - solution using relaxation methods
  - 2D equations including wind

Many methods for solving *Au=b*

- Direct methods
  - give the solution after a fixed number of operations
    - Gaussian elimination
    - LU factorisation
- Relaxation methods (this lecture)
  - gradually improve solution, starting from an initial guess
  - stop when the answer is sufficiently accurate
  - simple to implement but may be slow to converge on solution
    - or may fail completely!
- Krylov subspace methods (following lectures)
  - iterative (like relaxation methods) but more sophisticated
  - harder to implement but more efficient and reliable

Why not use Direct Methods?

- Direct methods explicitly operate on the matrix *A*
  - eg decompose it into *L* and *U* factors
- For PDEs, *A* is very sparse indeed
  - may contain 99% zeros so clearly we use compressed storage
  - we want to take advantage of this when we solve equations
- Difficult to exploit sparsity for direct methods
  - eg *L* and *U* may be dense even though *A* is sparse
  - for large systems of equations, we may run out of memory!
- Relaxation and Krylov methods exploit sparsity
  - relaxation methods operate on the equations not the matrix
  - Krylov methods comprise mostly matrix-vector multiplications
    - can write efficient routines to do $y = A\,x$ when *A* is sparse

Relaxation vs Matrix Methods

- Operate directly on the difference equations
  - can forget (almost!) all about the matrix representation $Au = b$ for this lecture
  - it turns out that relaxation methods can usefully be understood in terms of matrix-vector operations (not immediately obvious)
    - See lecture on “Matrix Splitting Techniques”

- For illustrative purposes, look at 1D problem
  - for simplicity with no wind
  - exercise will involve extending this to the 2D problem
    - quite straightforward in practice

Relaxation Methods

- 1D diffusion equations are

$$- u_{i-1} + 2\,u_i - u_{i+1} = 0, \quad i = 1, 2, \ldots N$$

- Equivalently: $u_i = \frac{1}{2}\,(u_{i-1} + u_{i+1})$
  - why not make an initial guess at the solution
  - then loop over each lattice point i and set $u_i = \frac{1}{2}\,(u_{i-1} + u_{i+1})$
  - ie we solve the equation exactly at each point in turn
- Updating u_i spoils solution we just did for u_{i-1}
  - so simply iterate the whole process again and again ...
  - ... and hope we eventually get the right answer!
- This is called the Jacobi Algorithm
  - the simplest possible relaxation method

Jacobi Algorithm

- Use superscript *n* to indicate iteration number
  - *n* counts the number of times we update the *whole* solution
  - equivalent to computer time
- Jacobi algorithm for diffusion equation is:

$$u_i^{(n+1)} = \tfrac{1}{2} \left(u_{i-1}^{(n)} + u_{i+1}^{(n)} \right)$$

- Each iteration, calculate $u^{(n+1)}$ in terms of $u^{(n)}$
  - don't need to keep copies of all the previous solutions
  - only need to remember two solutions at any time: u and u_{new}
    - corresponding to iterations n and $n+1$

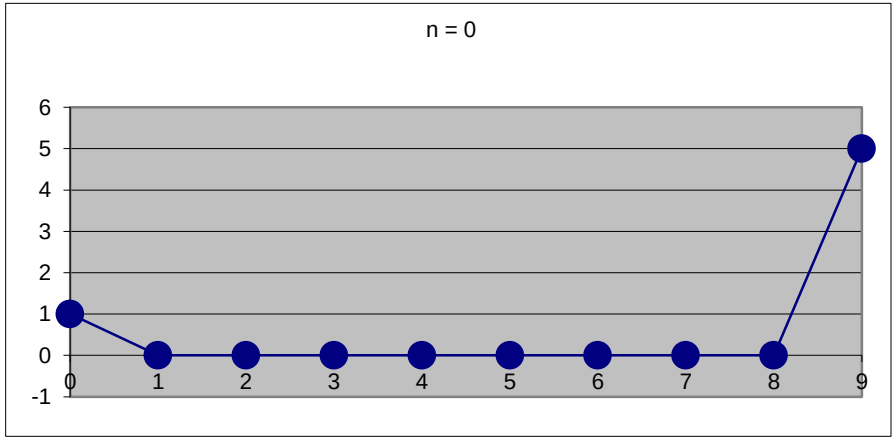
Jacobi Pseudo-Code

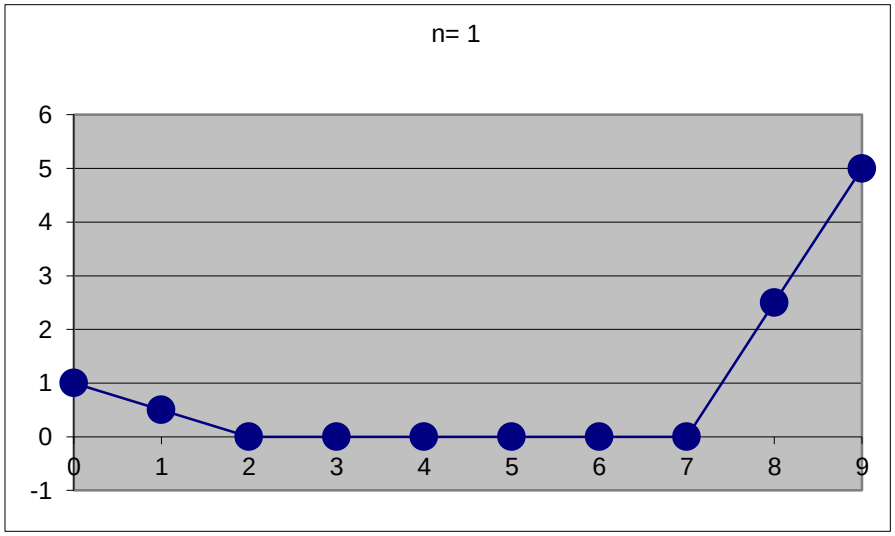
```
declare arrays:     u(0, 1, ..., M+1)
                  unew(0, 1, ..., M+1)

initialise: set boundaries: u(0)   = fixed value u0
                            u(M+1) = fixed value uM+1
            initial guess:  u(1, 2, ..., M) = guess value

loop over n = 1, 2, ...

  update:     loop over internal points: i = 1, 2, ... M
                unew(i) = 0.5*( u(i-1) + u(i+1) )
              end loop over i

  copy back: u(1, 2, ..., M) = unew(1, 2, ..., M)

end loop over n
```

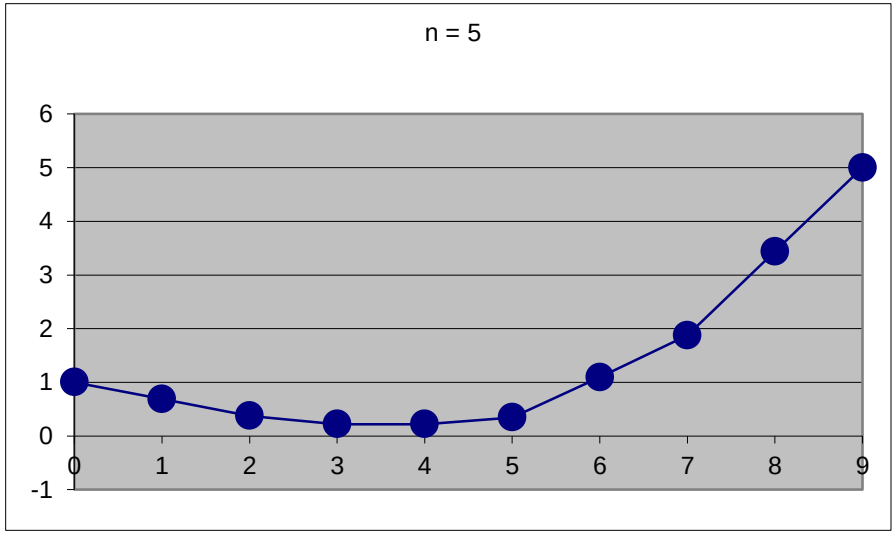
Implementation Notes

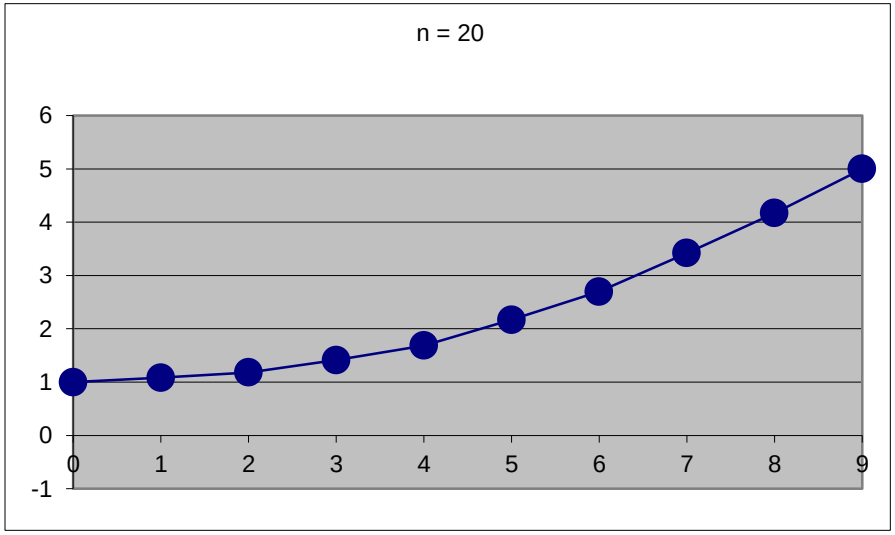
- Array declarations
  - Fortran: `real, dimension(0:M+1) :: u`
  - Java: `float[] u = new float[M+2];`
  - C: `float u[M+2];`
- Arrays explicitly contain boundaries u_0 and u_{M+1}
  - we set them according to boundary conditions
    - but we NEVER update them!
  - eg when we copy u_{new} back to u, only copy internal values
  - in pseudo-code, boundary values for u_{new} are never set
    - complete solution is therefore only ever present in u
    - might be more elegant to set boundaries in u_{new} as well
- What to choose for initial guess $u_i^{(0)}$?
  - for a simple implementation just set interior values to zero

Progress of Solution









When to Stop the Iterative Loop

- The solution *appears* to be getting better
  - must quantify this!
- For dense systems we used the residual
  - we tried to solve $Ax=b$, so $r = b-Ax$ should be a zero vector
  - in practice, there is a numerical error in solution of each equation
  - error in equation i is the value of r_i
    - residual is computed from the sum of the squares of r_i
- Can do the same thing for relaxation methods
  - compute the sum of the squares of the error in each equation
  - do this at the end of each iterative loop over n
    - stop if this is small enough

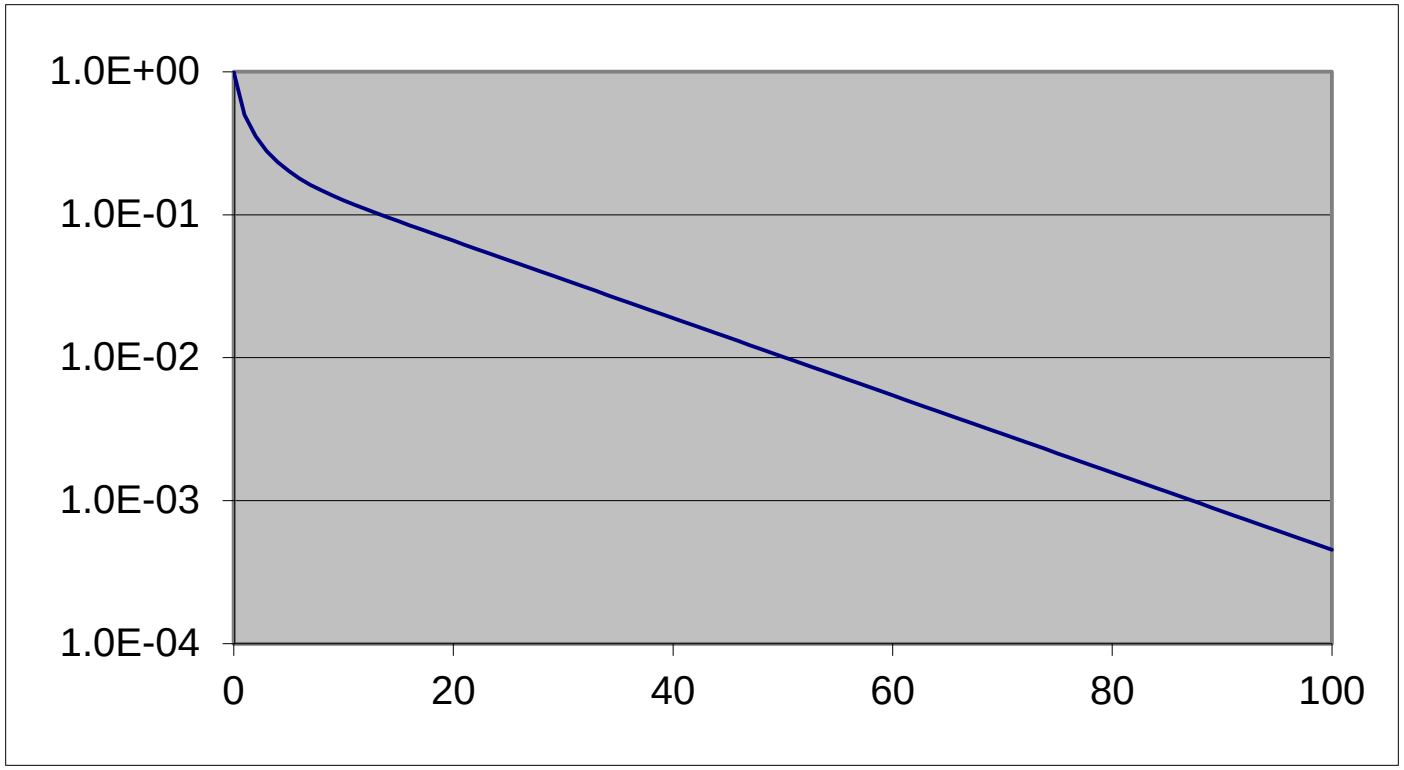
Pseudocode for Residual Calculation

```
loop over n = 1, 2, ...
  update:     ...
  copy back:  ...

  compute residue: rnorm = 0.0
                   loop over i = 1, 2, ..., M
                     rnorm = rnorm + (-u(i-1)+2*u(i)-u(i+1))^2
                   end loop over n
                   rnorm = sqrt(rnorm)

   normalise:      res = rnorm / bnorm

   if (res < tolerance) finish

end loop over n
```

Notes on Residual

- For a perfect solution, residue will be zero
  - in practice we will get a finite value
  - usually stop when it is “small”, eg a tolerance of res < 10^{-6}
  - there will be a limit to how small the residual can get
    - can easily hit the limits of single precision
    - use double precision everywhere (or at least perform residual calculation using doubles)
- Normalisation
  - need to divide by the norm of the *b* vector
  - we saw before that *b* corresponds to the boundary values
  - in 1D: `bnorm = sqrt(u(0)*u(0) + u(M+1)*u(M+1))`
    - in 2D, need to sum values of squares of $u_{i,j}$ over all edges

Residual Against Iteration



- Decreases exponentially
  - with a zero initial guess for *u*, should equal 1.0 at iteration zero

Parallelisation

- Very simple for Jacobi
- Decompose the problem domain regularly across processes/threads
  - for MPI we need halo regions due to *i*+1, *i*-1 references
  - halos are 1 cell wide for 5-point stencil
  - could be wider for larger stencils
  - swap halos between neighbouring processes every iteration
- Require global sums for, eg, residue calculation

Relaxation Methods

- About to cover some variations on Jacobi
  - which we hope will be faster!
- How can we tell if a method will work at all?
- Necessary (but not sufficient) condition
  - if the method arrives at the correct solution it must stay there

- Is this true for Jacobi? $u_i^{(n+1)} = \frac{1}{2} (u_{i-1}^{(n)} + u_{i+1}^{(n)})$

  - for a solution: $-u_{i-1}^{(n)} + 2 u_i^{(n)} - u_{i+1}^{(n)} = 0$, ie $\frac{1}{2} (u_{i-1}^{(n)} + u_{i+1}^{(n)}) = u_i^{(n)}$
  - so, $u_i^{(n+1)} = u_i^{(n)}$ and we stay at the solution
    - worth checking this for other methods

Gauss Seidel

- Why do we need both u_{new} and u ?

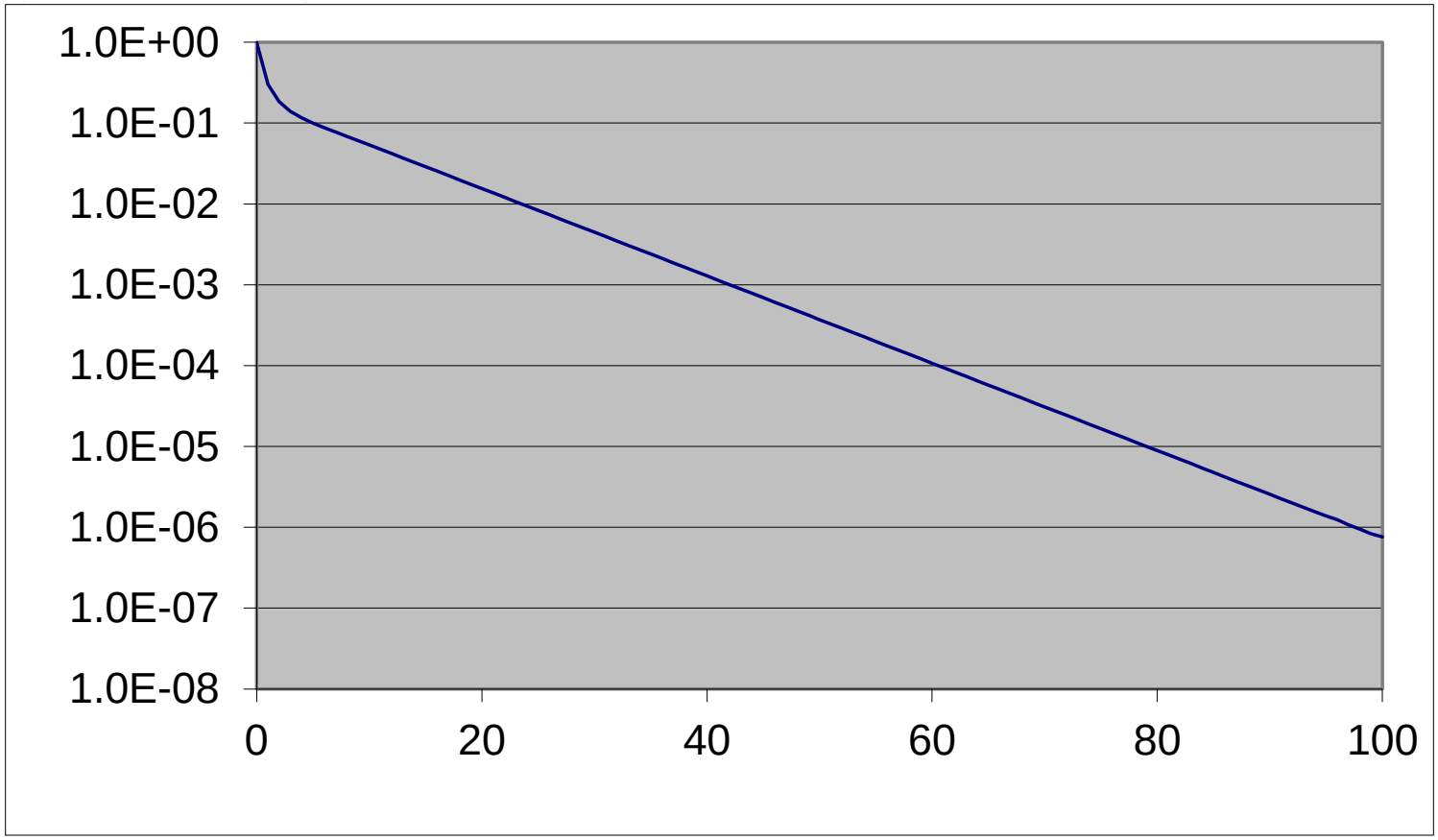
```
update:     loop over internal points: i = 1, 2, ... M
              unew(i) = 0.5*( u(i-1) + u(i+1) )
            end loop over i
copy back: u(1, 2, ..., M) = unew(1, 2, ..., M)
```

- Why not do the update in place?

```
update:     loop over internal points: i = 1, 2, ... M
              u(i) = 0.5*( u(i-1) + u(i+1) )
            end loop over i
```

  - this is called the *Gauss-Seidel* method

Convergence of Gauss-Seidel



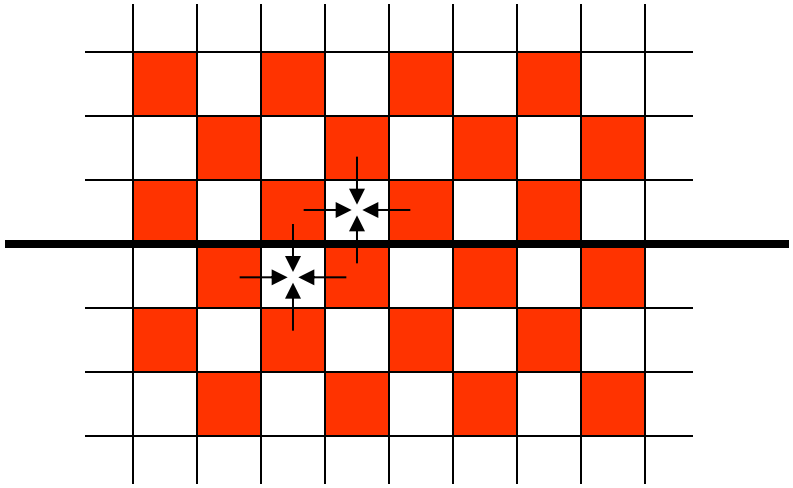
- Converges twice as fast as Jacobi
  - for less work and less storage!

Notes on Gauss Seidel

- Order of the update loop is now significant
  - we used normal (*lexicographic*) order: other orderings possible
- Red-black order divides grid into chequerboard
  - update all the red squares first then all the black ones
  - enables Gauss Seidel method to be parallelised

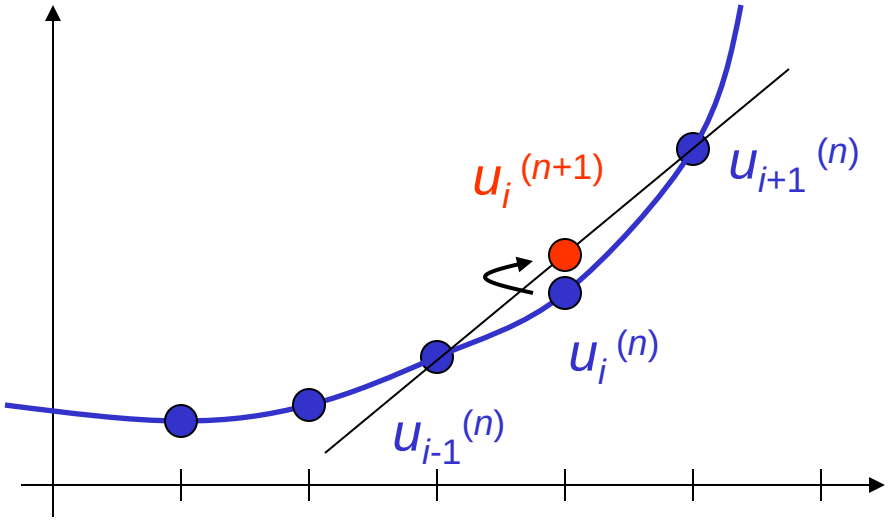
Processor 1

Processor 2



Over Relaxation

- Recall how Jacobi solution progressed



  - we have increased the value of u_i by a small amount
    - but we know the real solution is even higher
  - why not increase by more than suggested
    - ie multiply the change by some factor $w > 1$

Over-Relaxed Gauss Seidel

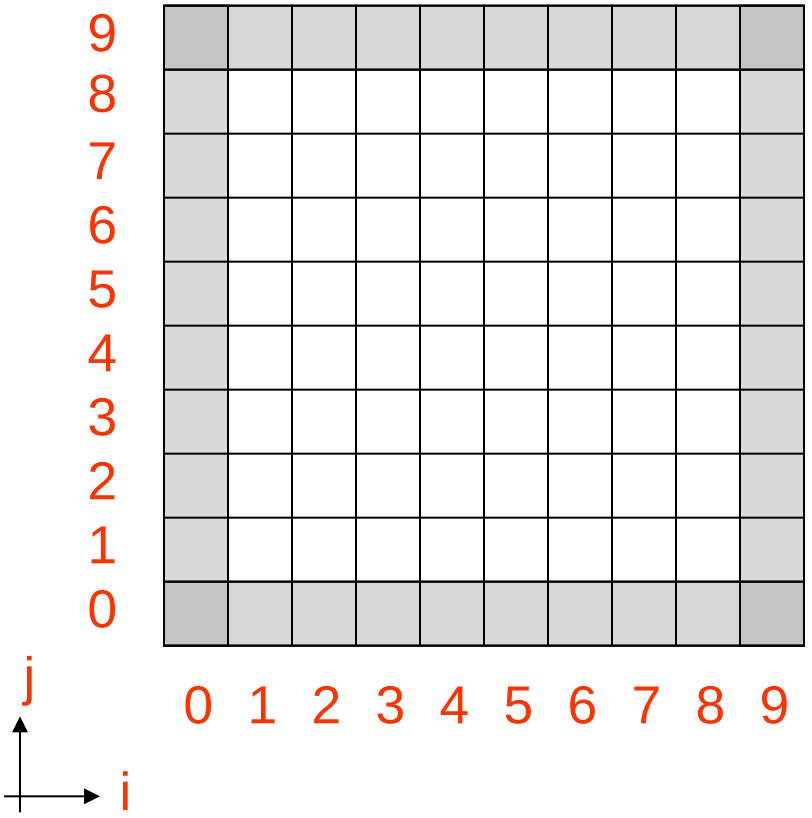
- Gauss-Seidel method: $u_i = \frac{1}{2} (u_{i-1} + u_{i+1})$
  - ie: $u_i = u_i + [\frac{1}{2} (u_{i-1} - 2 u_i + u_{i+1})]$
- Multiply change (in square brackets) by w
  - over-relaxed update: $u_i = u_i + \frac{1}{2} w (u_{i-1} - 2 u_i + u_{i+1})$
  - or $u_i = (1-w) u_i + \frac{1}{2} w (u_{i-1} + u_{i+1})$
- Notes
  - original method corresponds to $w = 1$
  - if we get to a solution we stay there for any value of w

Non-Linear Equations

- Relaxation methods deal directly with equations
  - doesn't matter that we cannot express them as $Au = b$
  - equally valid for non-linear equations (eg fluid dynamics)
- Non-linear equations can be very unstable
  - may need to under-relax to get convergence, ie $w < 1$

Extending to 2 Dimensions

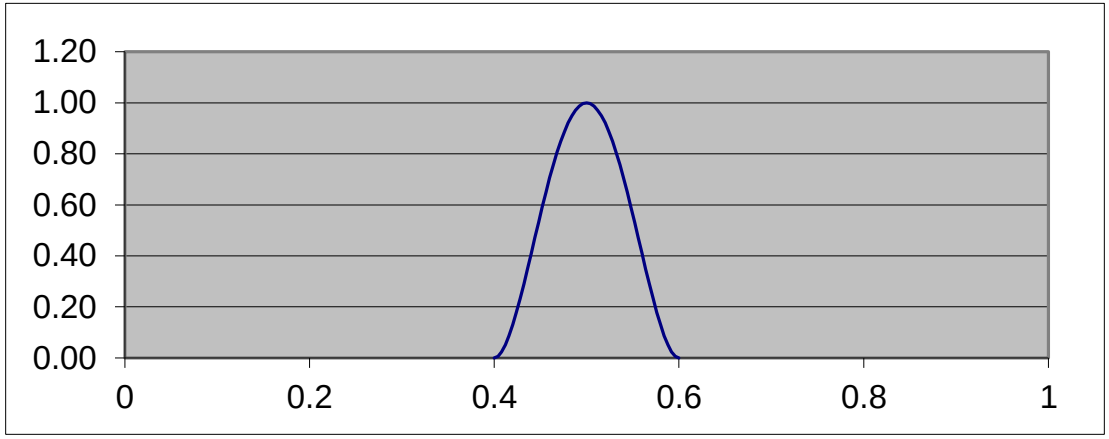
- Initialise
  - set boundary values (grey)
    - zero on top, bottom and left
    - hump function on right
  - zero interior (white)
- Loop over interior
    - $i = 1, 2, ..., M$
    - $j = 1, 2, ..., M$
  - update $u_{i,j}$ as appropriate
- Repeat until converged
- Write results
  - include boundaries so that the solution looks nice!



2D example with $M = 8$

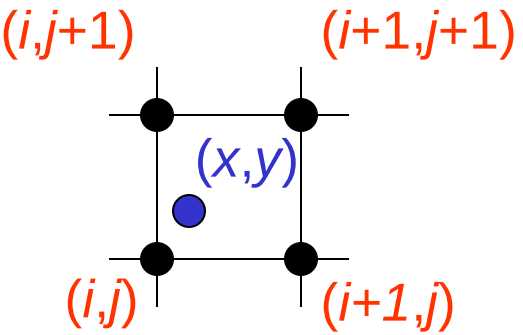
Notes (1)

- How do we convert from (*i*,*j*) to (*x*,*y*) coordinates?
  - for a domain of size 1x1:
    - $x = i * h$ and $y = j * h$
- What is the hump function?
  - $u(1.0,y) = k * (y_2 - y)^2 * (y - y_1)^2$
  - a peak, centred at $(y_2+y_1)/2$, dropping to zero for $y < y_1$ and $y > y_2$
  - for this example, take $y_1 = 0.4$ and $y_2 = 0.6$



Notes (2)

- How do we convert from (x,y) to (i,j) coordinates?
  - eg what lattice point do we look at to find $u(0.20,0.33)$?
  - (0.20,0.33) is unlikely to fall exactly on a lattice point
  - the four nearest neighbours are:
    - $i = \text{int}(x/h)$
    - $j = \text{int}(y/h)$



  - do weighted average of these four values (see exercise notes)

Convection-Diffusion Equations

- 1D Gauss-Seidel update

$$u_i = \left(\frac{1}{2+ah}\right)\left(u_{i-1} + (1+ah)\,u_{i+1}\right)$$

- 1D Over-Relaxed update

$$u_i = (1-w)u_i + w\left(\frac{1}{2+ah}\right)\left(u_{i-1} + (1+ah)\,u_{i+1}\right)$$

- 2D Discrete Equations

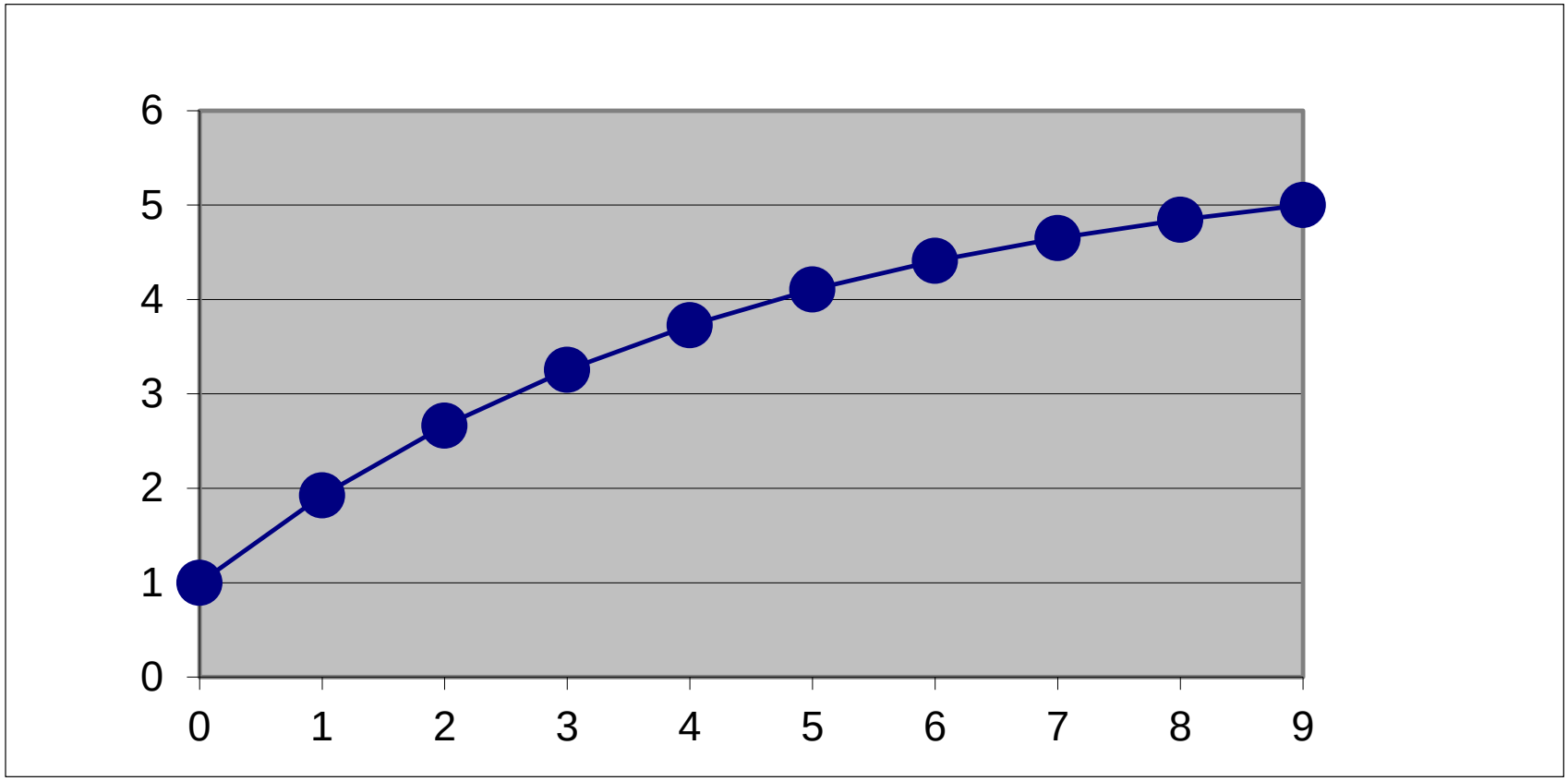
$$-u_{i,j-1} - u_{i-1,j} + (4 + (a_x + a_y)h)u_{i,j} - (1+a_x h)u_{i+1,j} - (1+a_y h)u_{i,j+1} = 0$$

(a_x, a_y) = wind strength from *x* (East) and *y* (North) respectively

Notes

- Have multiplied all the equations by h^2
  - equations now explicitly depend on h for a non-zero wind a
  - straightforward to derive update equations for 2D case
- A different convention for Krylov methods
  - maintain the $1/h^2$ factor in matrix A
    - therefore need to multiply RHS by same factor
    - happens to be more convenient
- Finite wind
  - matrix A is now non-symmetric
  - in 1D, lower-diagonal elements are $(1+ah)$, upper elements are 1
  - gives some minor technical issues when normalising the residue
    - see notes
    - if correctly normalised, residue at zero iterations will *always* be 1.0 if the initial guess is a zero solution

Sample solution: *N*=8 and *a*=2.0



Summary

- Relaxation methods
  - guess at an initial solution
  - update many times and stop when residue is small enough
- Update rule is very straightforward
  - solve exactly for each individual u_i
    - obtain formula by rearranging difference equations so u_i is on the LHS
- Interior points updated according to the PDE
  - boundary points set by the boundary conditions
- Jacobi is the simplest method
  - Gauss Seidel acts “in-place” and requires roughly half the iterations
  - appropriate over-relaxation can accelerate this even more
    - finding the best value of *w* requires some experimentation!