Today’s topics

- Iterative solution of linear systems of equations
    - Preconditioned conjugate gradient
    - Multigrid
- Solution of eigenvalues and -vectors

Linear System of Equations: Simple Iterative Solvers

- Simplest iteration is relaxation with residual:

$$x^{k+1} = x^k + \omega(b - Ax^k)$$

$$\Downarrow$$

$$x^{k+1} - x^k = (I - \omega A)(x^k - x^{k-1})$$

Convergent if spectral radius less than one: $\rho(I - \omega A) < 1$

- Next simplest is Jacobi:

$$x^{k+1} = D^{-1}(b - Rx^k), \quad R = A - D$$

$$\Downarrow$$

$$x^{k+1} - x^k = -D^{-1}R(x^k - x^{k-1})$$

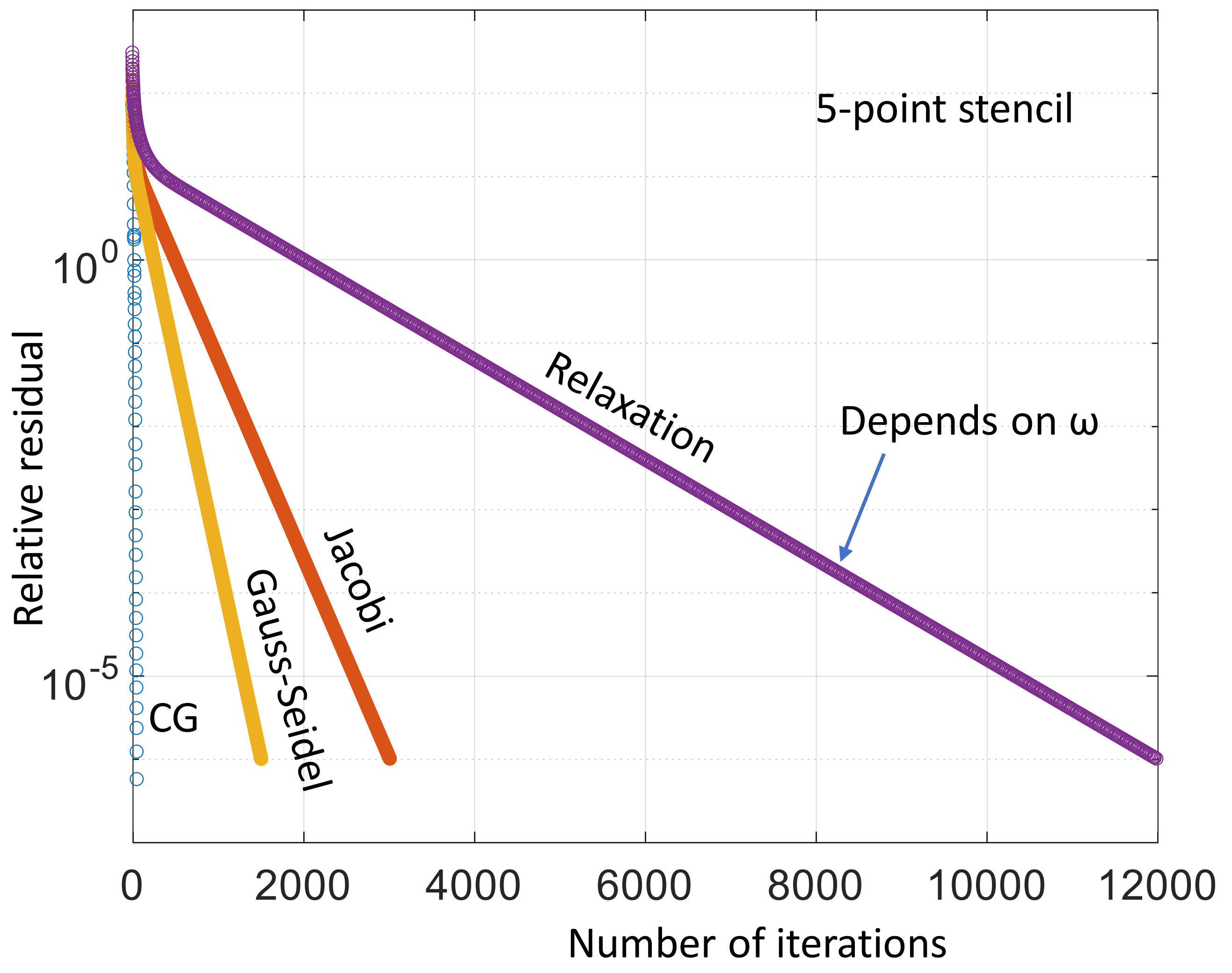
Convergent if : $\rho(-D^{-1}R) < 1$

- Then Gauss-Seidel

$$x^{k+1} = L^{-1}(b - Ux^k), \quad A = L + U$$

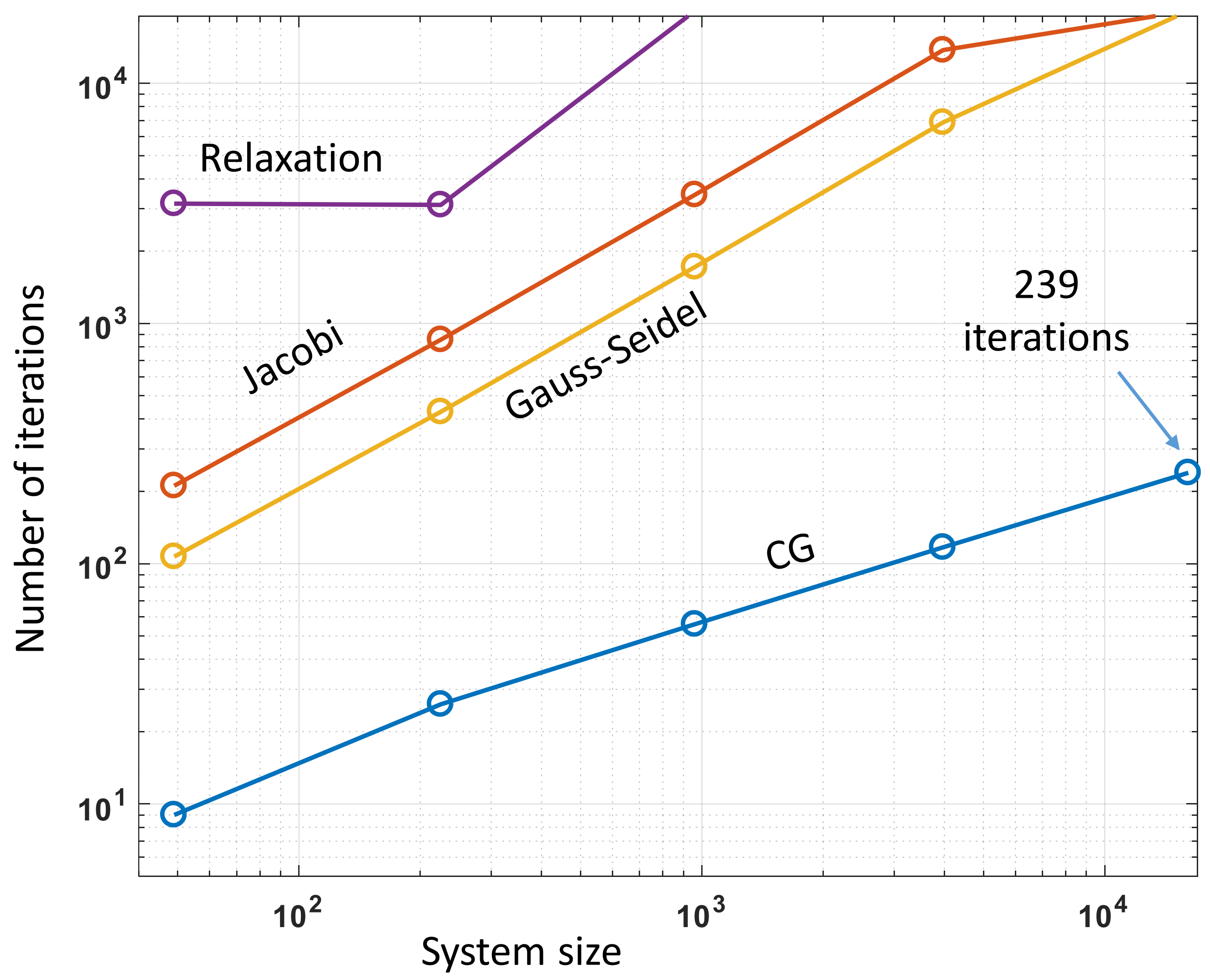
$$x^{k+1} - x^k = -L^{-1}U(x^k - x^{k-1})$$

Convergent if : $\rho(-L^{-1}U) < 1$



Iterations vs. System Size

- The most desirable method would converge in N iterations irrespective of system size
- None of the present ones does but the growth in CG is ”tolerable”



Iterative Methods: Conjugate Gradient(s)

- Suppose that *A* is symmetric, positive definite (spd)
- Solution to *Ax = b* also minimizes $f(x) = \frac{1}{2} x^T A x - b^T x$
- First solution would be to use the search direction $-\nabla f(x) = b - Ax$
- Hmm, this we tried with the relaxation method…

- We need to modify the search directions. Suppose we have a set of vectors $\{p_0, p_1, \dots, p_n\}$ with $p_i^T A p_j \sim \delta_{ij}$ and expand $x_* = \sum \alpha_i p_i$

  Then $b = Ax_* = \sum \alpha_i A p_i$ where $\alpha_k = \frac{p_k^T b}{p_k^T A p_k}$

- The clue of CG is to generate p_i and α_i

Hestenes and Stiefel (1952):
"Methods of Conjugate Gradients for Solving Linear Systems"
Journal of Research of the National Bureau of Standards. 49 (6).

Conjugate Gradient

- Let's have a look:

```
$r_0 = b - Ax_0$
$p_0 = r_0$
loop
    $\alpha_k = \frac{r_k^T r_k}{p_k^T A p_k}$
    $x_{k+1} = x_k + \alpha_k p_k$
    $r_{k+1} = r_k - \alpha_k A p_k$
    $\beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
    $p_{k+1} = r_{k+1} + \beta_k p_k$
    $k = k + 1$
end
```

$$f(x) = \frac{1}{2}x^T A x - b^T x$$

Search direction p_k, choose α_k to
1. Minimize *f(x)* along p_k
2. Maintain orthogonality of r_k

Calculate new residual $r_{k+1} = b - Ax_{k+1}$

Choose β_k such that p_k remain conjugate

Further, since

$$r_{k+1} = r_0 - \sum_{s=0}^{k} \alpha_s A p_s$$

$$\Downarrow$$

$$p_j^T r_{k+1} = p_j^T r_0 - \alpha_j p_j^T A p_j =$$

$$= p_j^T (r_0 - r_j) = \sum_{s=0}^{j-1} \alpha_s p_j^T A p_s = 0$$

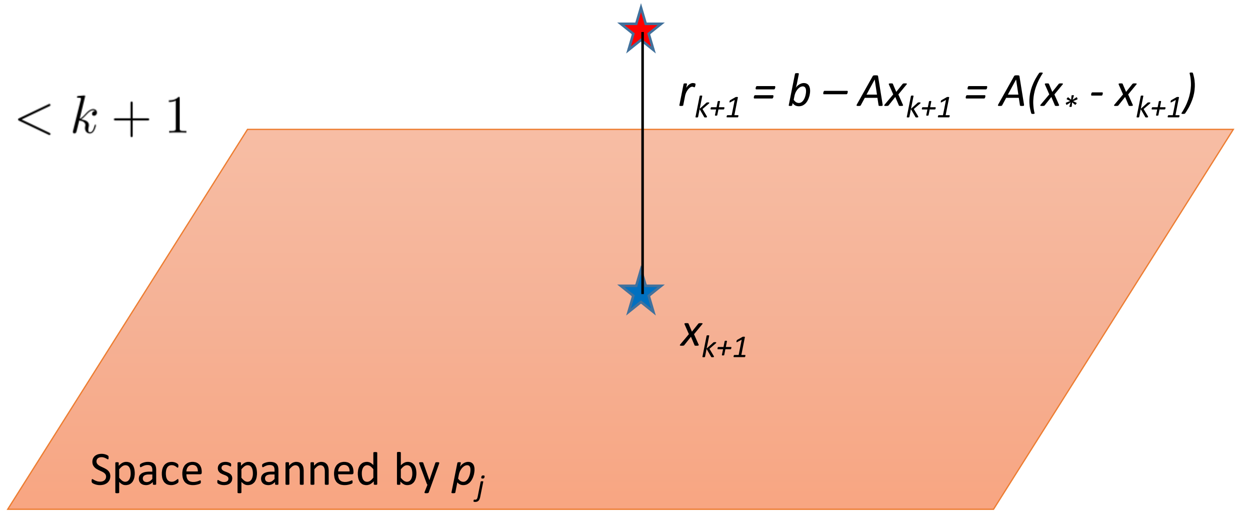
$$\Downarrow$$

$$p_j^T r_{k+1} = 0, \quad j < k+1$$

Conjugate Gradient: What You See is What You Get

```
$r_0 = b - Ax_0$
$p_0 = r_0$
loop
    $\alpha_k = \frac{r_k^T r_k}{p_k^T A p_k}$
    $x_{k+1} = x_k + \alpha_k p_k$
    $r_{k+1} = r_k - \alpha_k A p_k$
    $\beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
    $p_{k+1} = r_{k+1} + \beta_k p_k$
    $k = k + 1$
end
```

$$r_{k+1}^T p_j = 0, \quad j < k+1$$



CG is superior to Jacobi and Gauss-Seidel since it seeks for the globally optmal solution at each step

If $x_0 = 0$ as usually is the case the space is spanned by $A^j b$. This defines the Krylov subspace

$$\mathcal{K}_j(A, b) = \operatorname{span}\{b, Ab, A^2b, \ldots, A^{j-1}b\}$$

Conjugate Gradient: The Energy View

CG method was designed to minimize along the search directions p_j

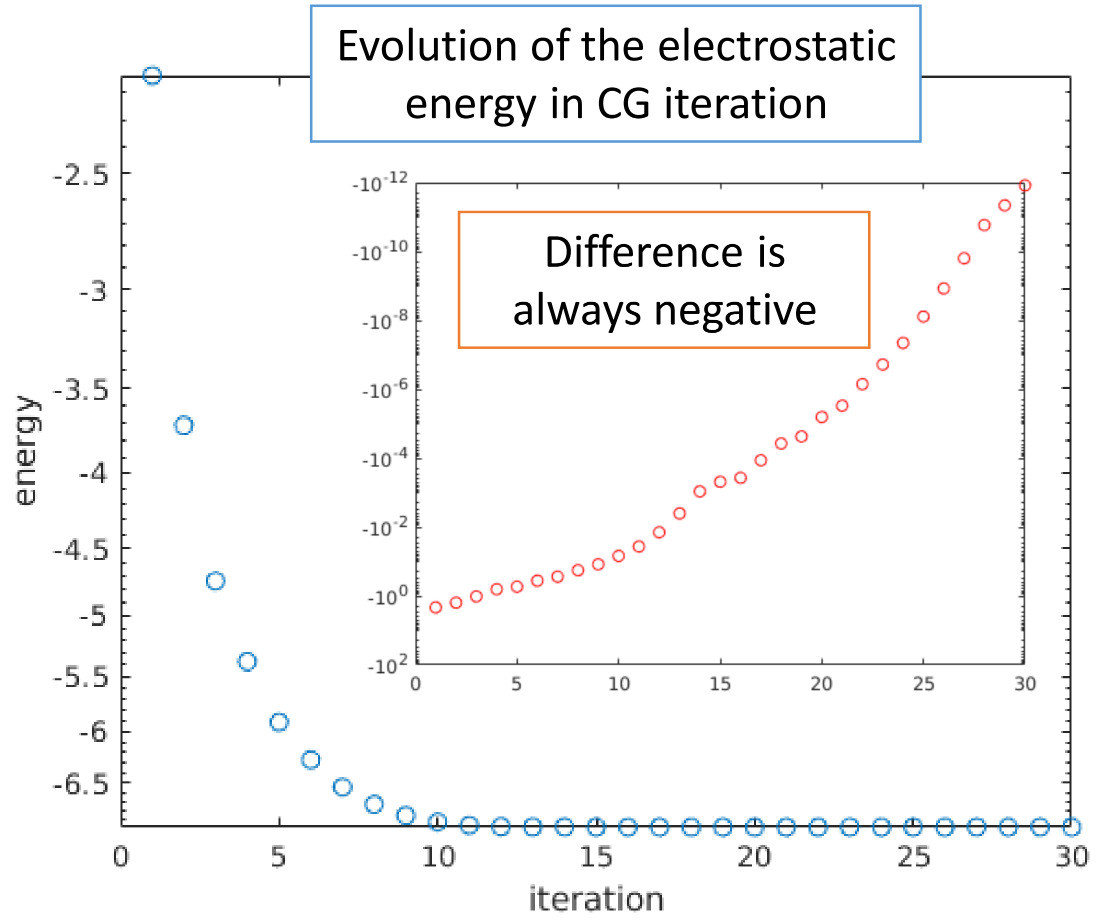
$$f(x) = \frac{1}{2}x^T A x - b^T x$$

However, set $u = \sum x_i \phi_i$

$$f(x) = \frac{1}{2}x^T A x - b^T x = \frac{1}{2}\int \nabla u \cdot \nabla u \, d\mathbf{r} - \int \rho u \, d\mathbf{r}$$

$$= \underbrace{\frac{1}{2}\int \vec{E}\cdot\vec{E}\, d\mathbf{r}}_{\text{field energy}} - \underbrace{\int \rho u \, d\mathbf{r}}_{\text{potential energy}}$$

Minimize difference between field energy and potential energy



Preconditioned Conjugate Gradient

- Convergence can accelerated by solving $P^{-1}Ax = P^{-1}b$ for $P \approx A$

$r_0 = b - Ax_0 \quad z_0 = P^{-1}r_0 \quad p_0 = z_0$

```
loop
    $\alpha_k = \frac{r_k^T z_k}{p_k^T A p_k}$
    $x_{k+1} = x_k + \alpha_k p_k$

    $r_{k+1} = r_k - \alpha_k A p_k$
    $z_{k+1} = P^{-1} r_{k+1}$

    $\beta_k = \frac{r_{k+1}^T z_{k+1}}{r_k^T z_k}$
    $p_{k+1} = z_{k+1} + \beta_k p_k$
    $k = k + 1$
end
```

Search direction p_k, choose α_k to
1. Minimize $f(x)$ along p_k
2. Maintain P-orthogonality of z_k

Calculate new residual $r_{k+1} = b - Ax_{k+1}$
Precondition the residual to improve the search direction

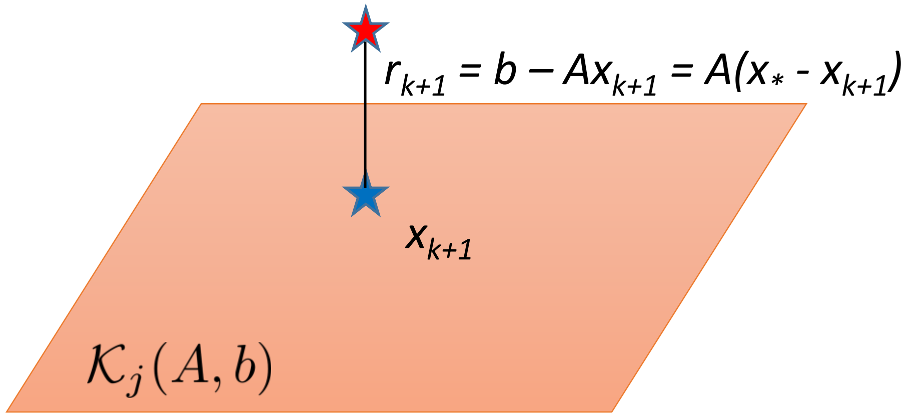
Choose β_k such that p_k remain conjugate

Requirements for *P*:
1. Symmetric
2. Positive definite
3. Static (same for all *k*)
4. Easy to implement P^{-1}

Krylov Subspace Methods

- With CG we got extremely lukcy having far too many orthogonalities
- With a general matrix A much less remains
- But the main idea remains
  1. Construct the Krylov subspace $\mathcal{K}_j(A,b) = \operatorname{span}\{b, Ab, A^2b, \dots, A^{j-1}b\}$
  2. Project the solution to the subspace
     - Get something more complicated: GMRES, BiCG(Stab), (TF)QMR, MINRES,…

Discuss: How could you project?



Homework 10

$$\begin{cases} -\Delta u(x,y) + 2u(x,y) = \exp(-\frac{(x-0.5)^2+(y-0.5)^2}{10}), & (x,y) \in [0,1]\times[0,1] \\ u(x,y) = 0, & \text{on the boundary} \end{cases}$$

a) Implement a discretization of your choice for the above problem. Solvers based on the finite difference method or the finite element method can both be used. (See also Homework 9.) Solve the resulting linear system of equations using the conjugate gradient method without preconditioning. You can use the provided skeleton. What is the condition number of your matrix A? (2 p.)

b) Improve the convergence by introducing preconditioners of increasing complexity. In each case report the number of iterations needed to solve the problem and the condition number of the preconditioned system.

i) Diagonal preconditioner: $P = \mathrm{diag}(A)$.

ii) Lower-upper preconditioner $P = P_1P_2$ where $P_1 = L_*$ is the lower triangle of A including diagonal and $P_2 = U_*$ is the upper triangle of A including the diagonal. (Yes, the diagonal gets counted twice.) If you are using matlab check out the functions `tril, triu`.

iii) Incomplete Cholesky factorization of A without fill-in, IC(0). In the IC(0) preconditioner the matrix A is factored approximately $A \approx \tilde{L}\tilde{L}^T$ where $\tilde{L}$ is an incomplete Cholesky factor of A. In the case of no fill-in only non-zero entries of A are included in $\tilde{L}$. For matlab see the function `ichol`.

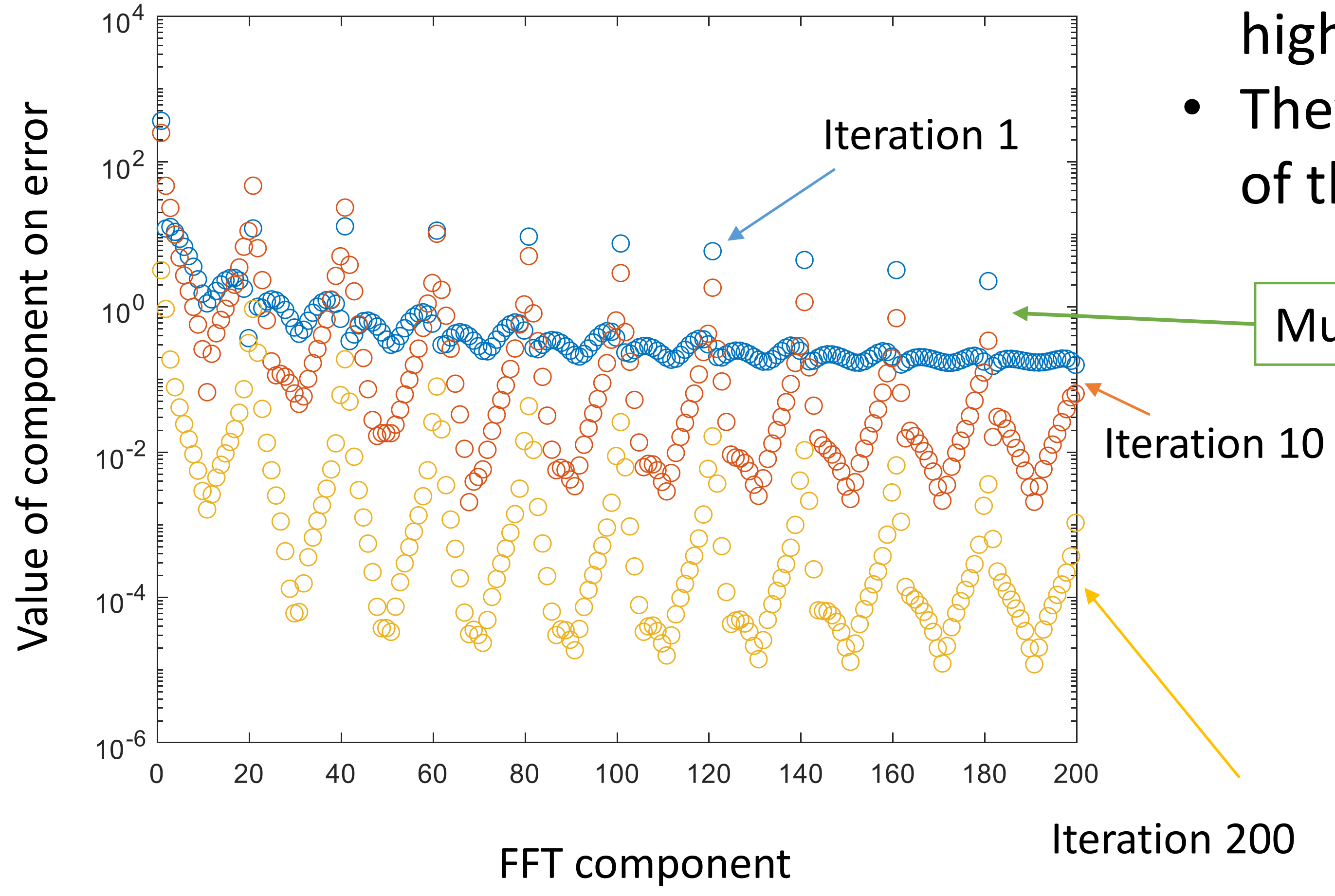
Multigrid

- Matrix form of Poisson $Au = b$ where A is discrete Laplacian
- A solution *w* has residual *r* defined as $r = b - Aw$
- Simple iterative methods first remove short range errors leaving only long range errors
- Multigrid:
  - Move to a coarser grid and solve $Ad = r$
  - Do this recursively as long as you can
  - Correct on the coarser grid by *w+d*: $A(w + d) = Aw + Ad = b - r + r = b$

If you don't get the idea from this, read for example http://www.cs.berkeley.edu/~demmel/cs267/lecture25/lecture25.html

Failure of Simple Iterative Methods

FFT of the error x - x_k



- Simple methods reduce first the high-frequency components
- They lack the global picture → slow-down of the convergence after a few iterations

Multigrid aims to capitalize on this gap

Discuss: Why does going to coarser grids help?

Recursive function example

```
program example_factorial
  implicit none
  integer :: i, f
  i=10
  f=factorial(i)
  write(*,*) "Main part:", i, f
contains
  recursive function factorial(n) result(f)
    integer :: f, n
    write(*,*) n, "comes in"
    if (n>1) then
       f=n*factorial(n-1)
    else
       f=1
    end if
    write(*,*) n, " came in and out goes", f
  end function factorial
end program example_factorial
```

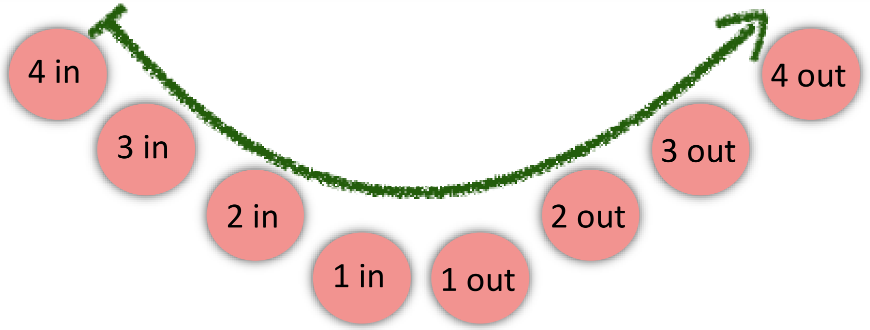
Call a function to calculate factorial

This is a recursive function

For inputs larger than one, the same function is called again

```
10 comes in
 9 comes in
 8 comes in
 7 comes in
 6 comes in
 5 comes in
 4 comes in
 3 comes in
 2 comes in
 1 comes in
 1  came in and out goes          1
 2  came in and out goes          2
 3  came in and out goes          6
 4  came in and out goes         24
 5  came in and out goes        120
 6  came in and out goes        720
 7  came in and out goes       5040
 8  came in and out goes      40320
 9  came in and out goes     362880
10  came in and out goes    3628800
Main part:          10     3628800
```

Notice the order of the outputs: recursion goes to smaller and smaller values of n before it moves back to larger values, V-cycle



Multigrid

From the main part of the code, we call this subroutine as many times as needed to reach convergence.

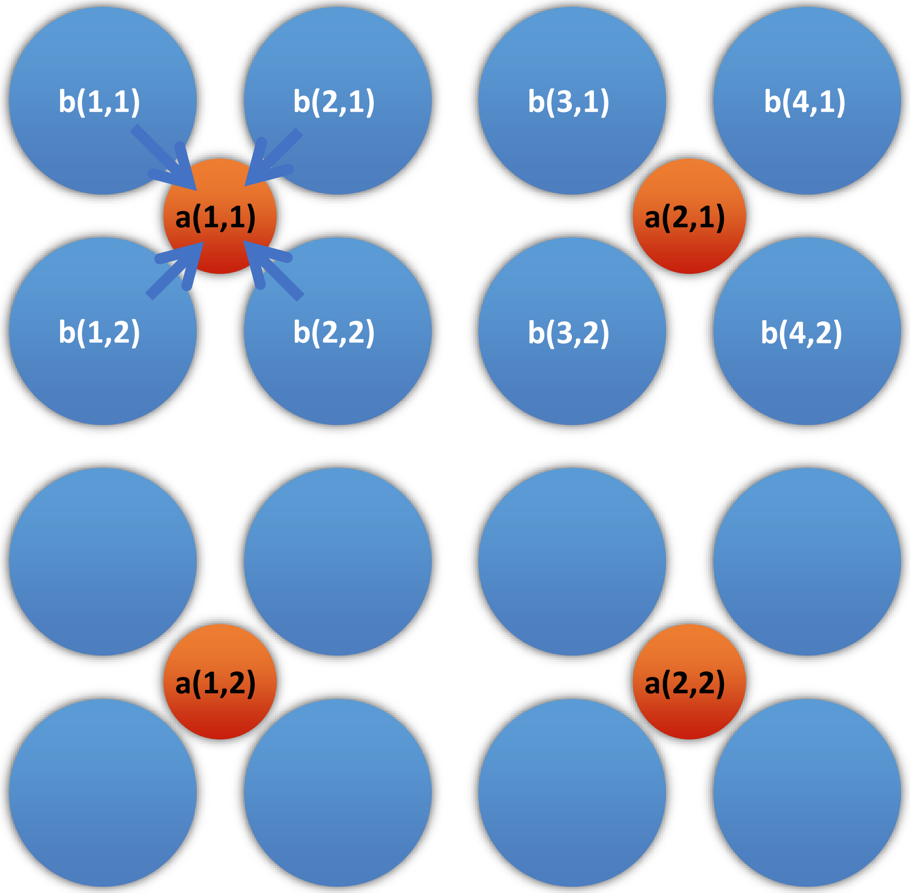
```
recursive subroutine MG_r(w,r)
  ! This is the 'real' multigrid part.
  ! Does a V-cycle.
  real(kind(1.d0)) :: w(:,:), r(:,:)
  real(kind(1.d0)), dimension(size(w,1)/2,size(w,2)/2) :: sw, sr
  integer :: i, j, Nx, Ny
  Nx=size(w,1)
  Ny=size(w,2)

  do i=1, 10
     call do_gs(w,r) ! Gauss-Seidel pre-relaxation
  end do
  if (Nx>2) then
     sw=0.d0
     sr=to_coarse(Laplace_p(w)+r)
     call MG_r(sw,sr*4.d0) ! We have to multiply residual, 'double grid'
     w=w+to_fine(sw)
     do i=1, 10
        call do_gs(w,r) ! Gauss-Seidel post-relaxation
     end do
  else
     do i=1, 10
        call do_gs(w,r) ! Exact solution on the coarsest grid
     end do
  end if
end subroutine MG_r
```

Note that we move coarser grid until we reach 2x2 lattice and after that we start to get to the “out” phase of the previous recursive function example.

Then we move to finer and finer grids and do more Gauss-Seidel steps

On the coarsest grid (2x2) we solve the problem ”exactly”. This is not really necessary here but if the coarsest grid is larger then this step is mandatory.



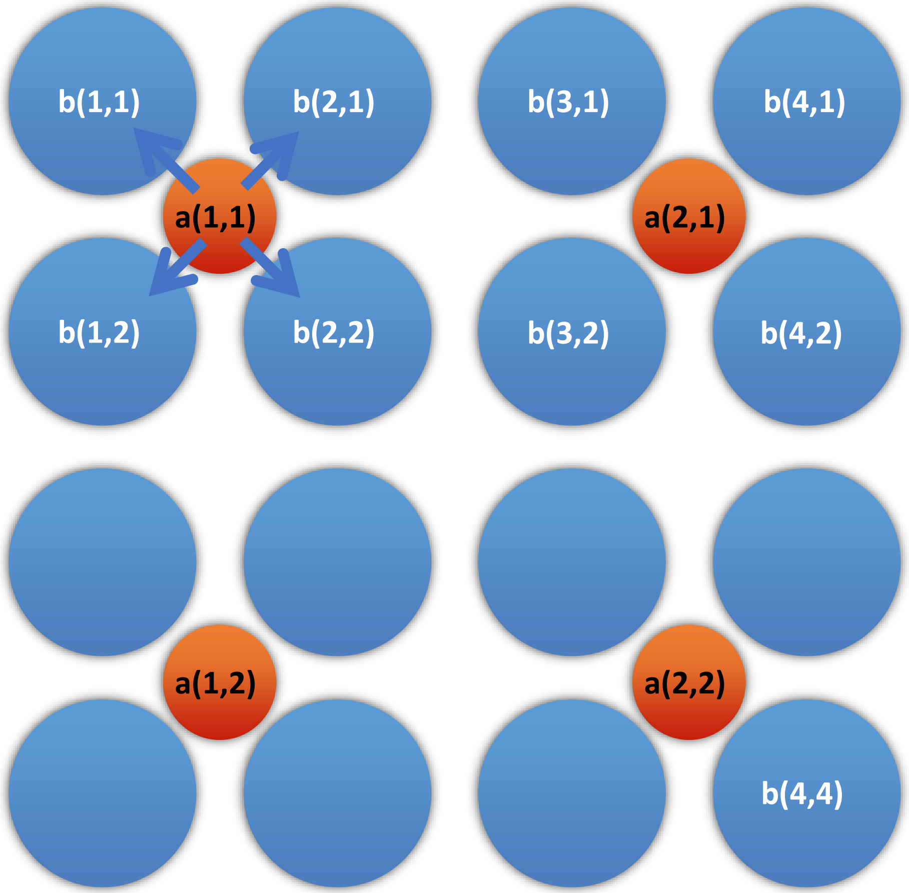
Multigrid

```
function to_coarse(b) result(a)
 ! This moves data to coarser grid
 real(kind(1.d0)) :: b(:,:)
 real(kind(1.d0)), dimension(size(b,1)/2,size(b,2)/2) :: a
 integer :: i, j, Nx, Ny
 Nx=size(a,1)
 Ny=size(a,2)
 do i=1, Nx
   do j=1, Ny
     a(i,j)=sum(b((2*i-1):(2*i),(2*j-1):(2*j)))/4.d0
   end do
 end do
end function to_coarse
```

```
a(1,1)=sum(b(1:2,1:2)/4
a(1,2)=sum(b(1:2,3:4)/4
a(2,1)=sum(b(3:4,1:2)/4
a(2,2)=sum(b(3:4,3:4)/4
```

Simply an average of the four neighbours.

Notice that in fortran we don't have to know the size of the incoming matrix



Multigrid

```
function to_fine(a) result(b)
 ! This moves data to finer grid
 real(kind(1.d0)) :: a(:,:)
 real(kind(1.d0)), dimension(size(a,1)*2,size(a,2)*2) :: b
 integer :: i, j, Nx, Ny
 Nx=size(b,1)
 Ny=size(b,2)
 do i=1, Nx
   do j=1, Ny
     b(i,j)=a((i+1)/2,(j+1)/2)
   end do
 end do
end function to_fine
```

Integer division!

b(1,1)=a(1,1)
b(1,2)=a(1,1)
b(2,1)=a(1,1)
…
b(4,4)=a(2,2)

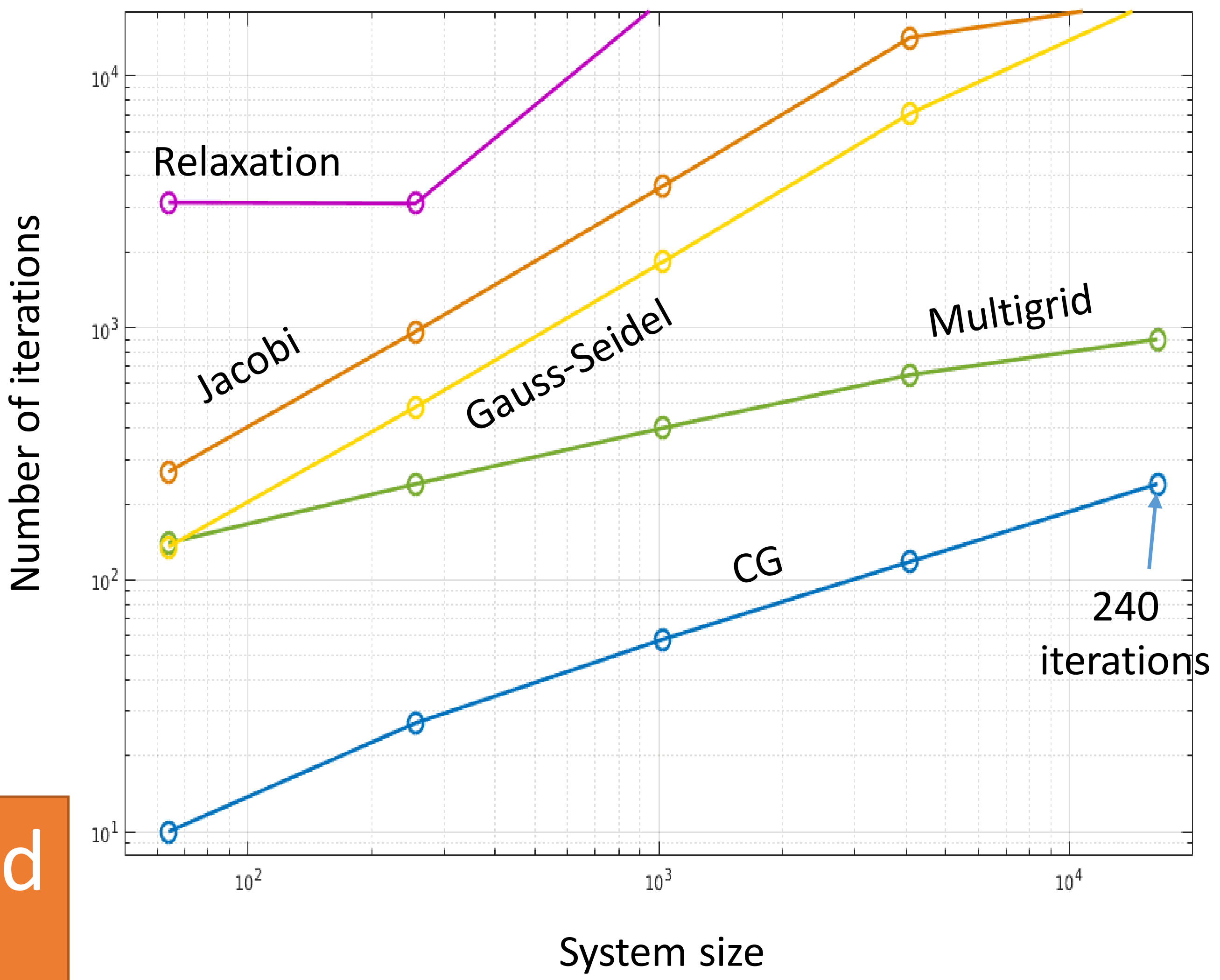
Copy the data to the four nearest points.

For integer division in many computer languages, 3/2=1 etc.

Iterations vs. System Size

- The most desirable method would converge in N iterations irrespective of system size
- None of the present ones does but the growth in CG and MG is ”tolerable”
- Iterations in MG are G-S smoothing steps → comparable cost

Discuss: How are Multigrid and CG related?



Solution of an Eigenvalue Problem

- Basic problem: $H\psi_i = \epsilon_i \psi_i$
- Or, in generalized form $H\psi_i = \epsilon_i S \psi_i$
- Since S is in practise always s.p.d. one writes

$$S = LL^T \qquad \tilde{\psi}_i = L^T \psi_i \quad \Longrightarrow \quad L^{-1} H L^{-T} \tilde{\psi}_i = \tilde{H} \tilde{\psi}_i = \epsilon_i \tilde{\psi}_i$$

$\Longrightarrow$ For start, let’s not mind about the generalized problem.

Solution of an Eigenvalue Problem: Direct method

- Basic problem: $H\psi_i = \epsilon_i\psi_i$
- Assume symmetric *H*:
  1. Transform *H* into tridiagonal *T* with Householder reflections: $P = I - \tau v v^T$
     - This works one column / row at a time and the entire matrix needs to be treated → slow
  2. Solve eigenvalues and –vectors of *T* with, e.g., *QR*-algorithm
     - *QR* algorithm is very fast for eigenvalues of tridiagonal *T*, not so for general *H* or if eigenvectors are desired
  3. Refine eigenvalues and –vectors if needed
  4. Backtransform the eigenvectors with the reflections from 1.
- Drawback: All of *H* needs to be treated in 1. even in the case when only a few eigenvalues are needed
- In 2. only part of the eigenvalues could be solved but this doesn't really save very much time and effort unless eigenvectors are required.

Eigenvalue Problem: Simple Iterative Methods

- Power method:
    - Start with random vector ψ^0

$$v = \frac{\psi^k}{||\psi^k||} \Rightarrow \psi^{k+1} = Hv \Rightarrow \epsilon^{k+1} = v^T\psi^{k+1}$$

    - Converged when $||\psi^{k+1} - \epsilon^{k+1}v|| < \text{tol} \cdot \epsilon_{k+1}$

Converges to the largest eigenvalue ε_{MAX}

- Inverse power iteration
    - Apply power method to $(H - \sigma I)^{-1}$
    - Set $\epsilon = \sigma + \frac{1}{\text{result}}$

Converges to the eigenvalue closest to σ

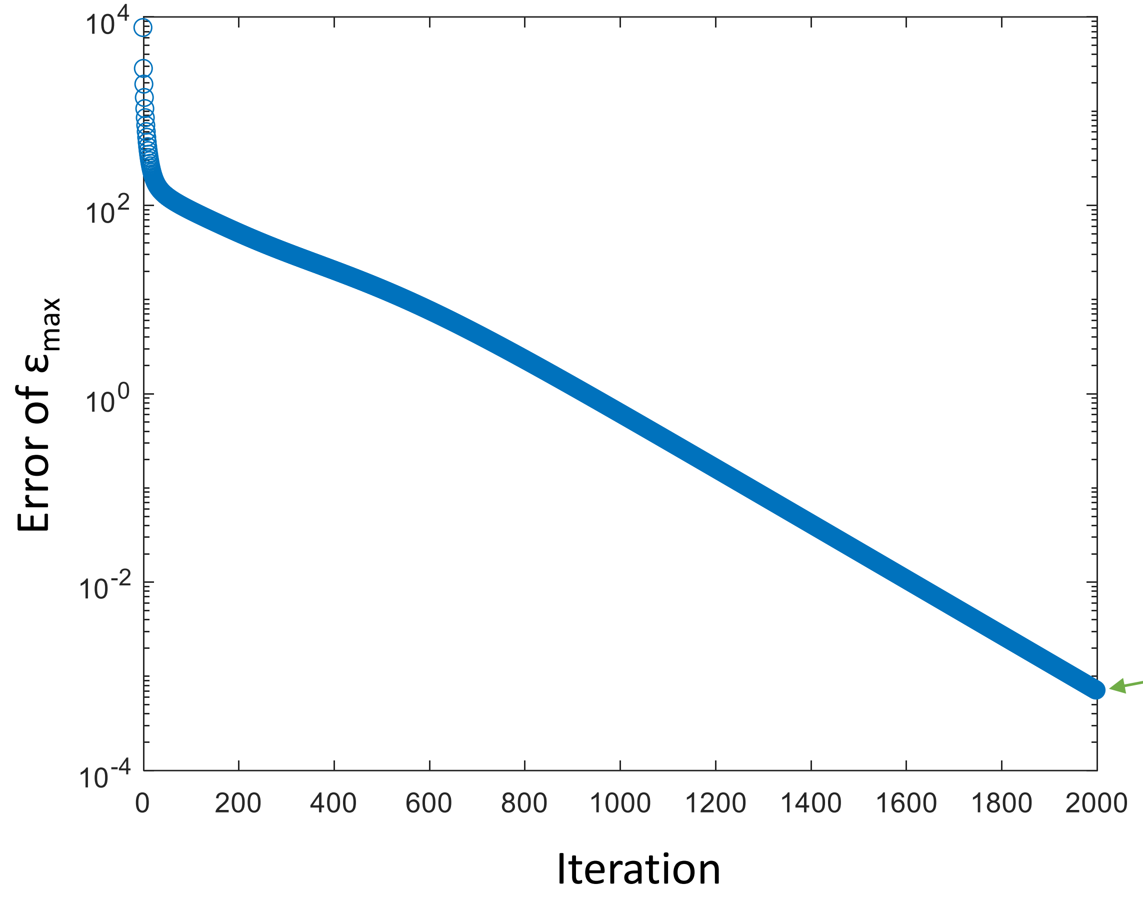
- QR-algorithm:
    - Set $H^0 = H$

$$H^k = Q^k R^k \Rightarrow H^{k+1} = R^k Q^k$$
$$= (Q^k)^T H^k Q^k$$

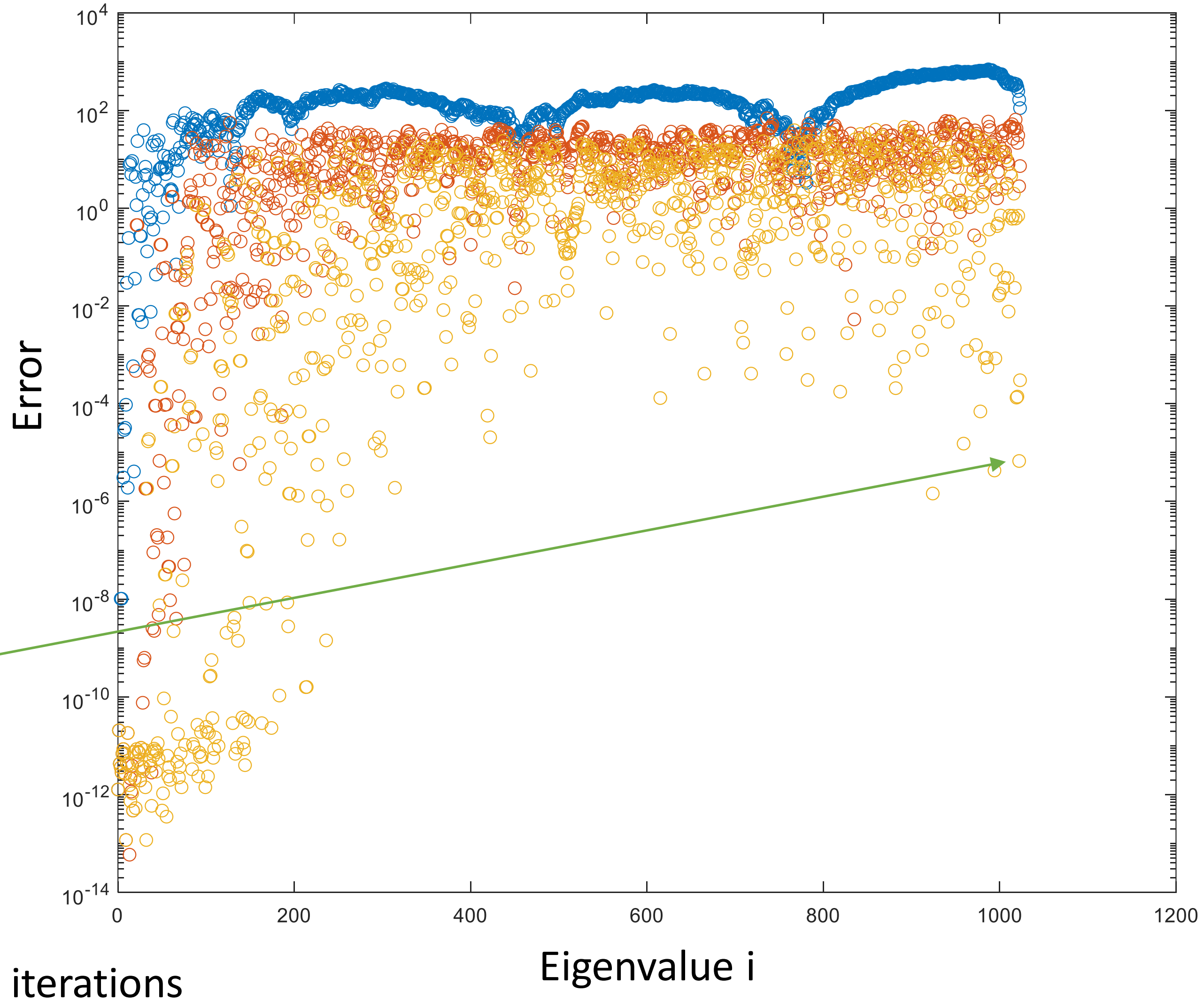
Converges to a diagonal matrix with eigevalues of *H* on the diagonal. Use inverse power iteration to find eigenvectors

Convergence for 5-point stencil

Power method, largest eigenvalue $\varepsilon_{max} = 8.6923 \cdot 10^3$



QR-algorithm, 1024 eigenvalues, 100, 500, 2000 iterations



Inverse Power Iteration converges in 12 iterations
Shifted iteration to the middle of the spectrum coverges in 4 iterations

Eigenvalue Problem: More Iterative Methods

- As for the solution of *Ax=b* we have to look global: Lanczos method

```
Initialize:  $r = \psi^0$   $\beta_0 = ||r||$
loop until convergence
    $v_k = r/\beta_{k-1}$
    $r = H v_k$
    $r = r - v_{k-1}\beta_{k-1}$
    $\alpha_k = v_k^T r$
    $r = r - v_k \alpha_k$
    $\beta_k = ||r||$
    compute eigenvalues of $T_k = Y\Theta Y^T$   where
end loop
compute eigenvectors as $\Psi = V_k Y$   where $V_k = [v_1|v_2|\dots|v_k]$   $v_k \in \mathcal{K}_k(H, \psi^0)$
```

Get a new direction

$$\left.\begin{aligned} r &= r - v_{k-1}\beta_{k-1} \\ \alpha_k &= v_k^T r \\ r &= r - v_k\alpha_k \end{aligned}\right\}$$

Orthogonalize against previous vectors

$$T_k = \begin{pmatrix} \alpha_1 & \beta_1 & & \\ \beta_1 & \alpha_2 & \ddots & \\ & \ddots & \ddots & \beta_{k-1} \\ & & \beta_{k-1} & \alpha_k \end{pmatrix}$$

In addition:

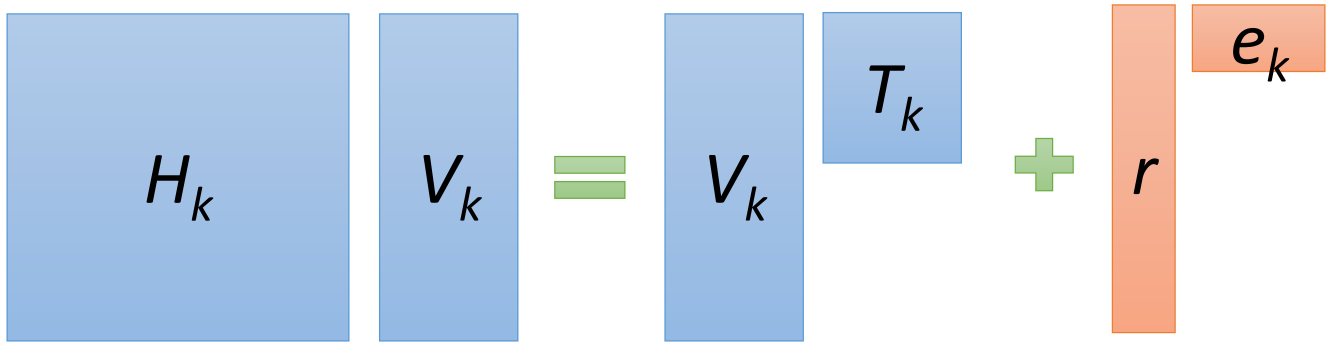
$$HV_k = V_k T_k + r e_k^T$$

$$V_k^T r = 0$$

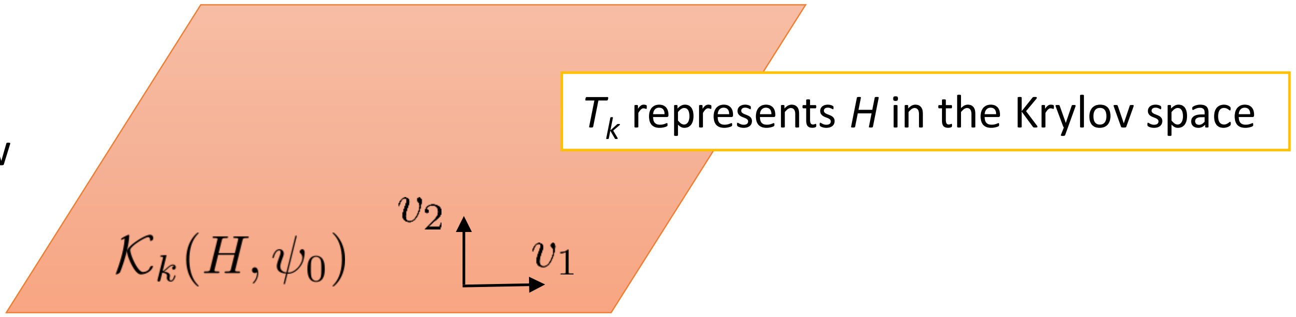
Eigenvalue Problem: Lanczos

- Views on Lanczos

1. Block matrix view



2. Krylov space view



T_k represents H in the Krylov space
$\mathcal{K}_k(H, \psi_0)$
v_2
v_1

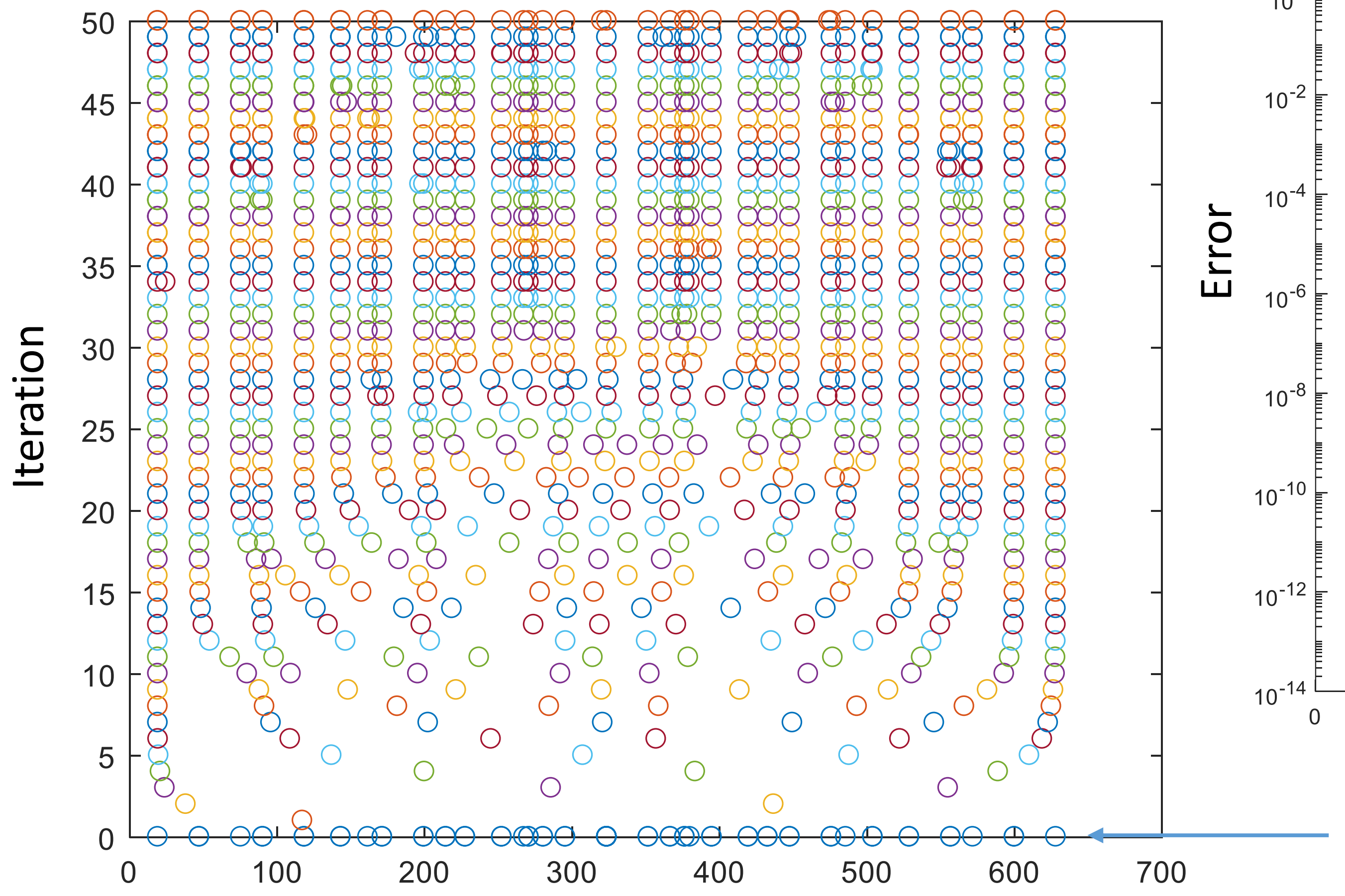
3. Matrix algebra view

$$\begin{cases} V_k = [v_1|v_2|\dots|v_k] \\ V_k^T H V_k = T_k \end{cases} \qquad T_k y_j = \theta_j y_j \Rightarrow \begin{cases} \epsilon_j \approx \theta_j \\ \psi_j \approx V_k y_j \end{cases}$$

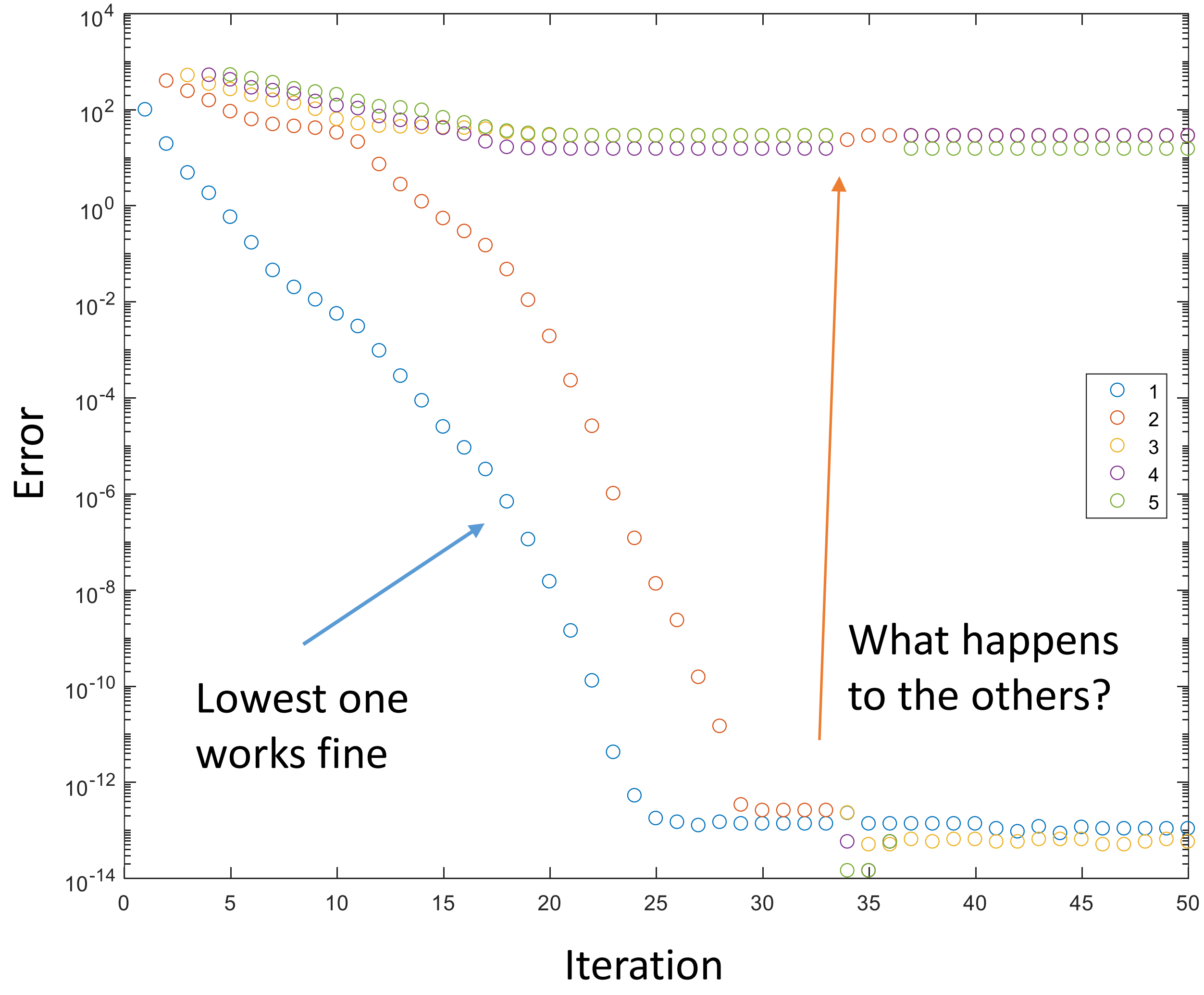
Eigenvalue Problem: Lanczos

- Convergence of Lanczos

Convergence of the entire spectrum

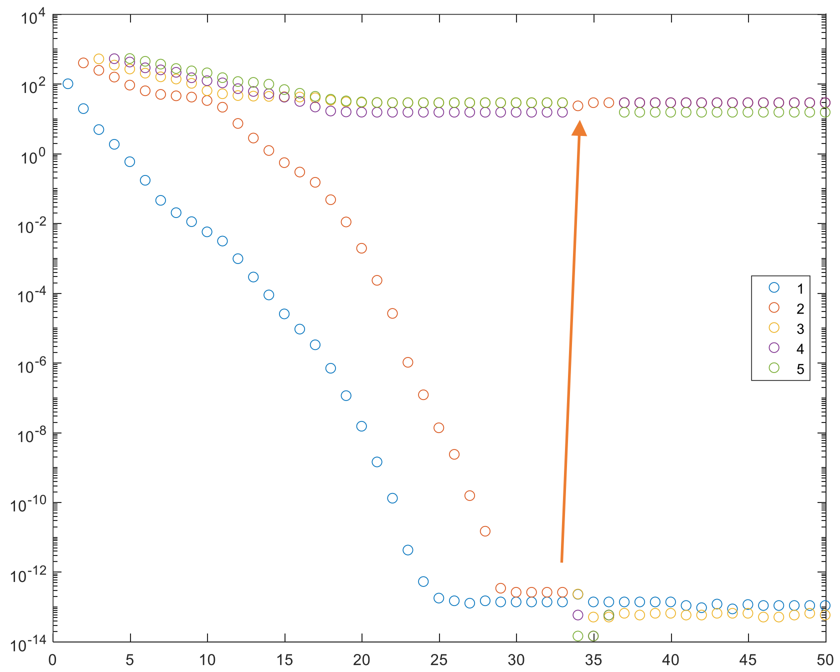


Convergence of the lowest eigenvalues



Lanczos: Loss of Orthogonality

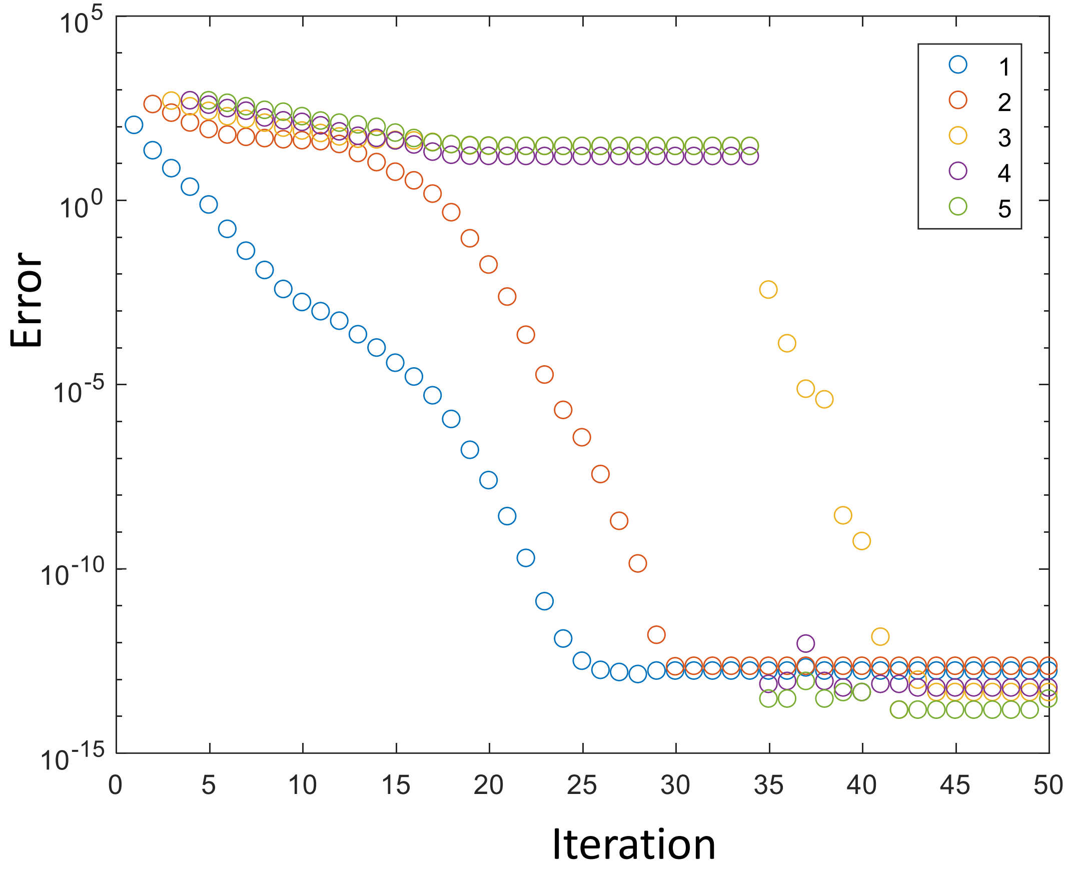
Convergence of the lowest eigenvalues



The reason behind the failure to convegre is loss of orthogonality among vectors v_k

The solution is to
- Orthogonalize v_k
  a) At every iteration
  b) When loss of orthogonality is detected
- Restart Lanczos

Full orthogonalization in progress



A Variant: Jacobi-Davidson

```
Initialize $t = \psi_0$
loop until convergence
    for i = 1,…,k-1
        $t = t - (v_i^T t) v_i$
    end
    $v_k = t/||t||$   $V_k = [v_1|v_2|\dots|v_k]$
    $M_k = V_k^T H V_k$
    compute largest eigenpair of $M_k = Y \Theta Y^T$
    $\psi_k = V_k y$
    $r = H\psi_k - \theta_k \psi_k$
    if || r || is small enough, stop
    solve (approximately) t from $(I - \psi_k \psi_k^T)(H - \theta_k I)(I - \psi_k \psi_k^T) t = -r$
end loop
```

Orthogonalize against previous vectors in V_k

Same thing as in Lanczos but the space V_k is constructed differently

The point in Jacobi-Davidson is that the new vector in V_k is not a Krylov vector but orthogonal to ψ and solved from an equation.

Idea: Find *t* that is orthogonal to ψ_k and

$$H(\psi_k + t) = \epsilon(\psi_k + t)$$

$$\Downarrow$$

$$(H - \epsilon I)t = -(H - \epsilon I)\psi_k$$

H in the orthogonal subspace is

$$(I - \psi_k \psi_k^T) H (I - \psi_k \psi_k^T)$$

and hopefully $\varepsilon \approx \theta_k$

Jacobi-Davidson in Action

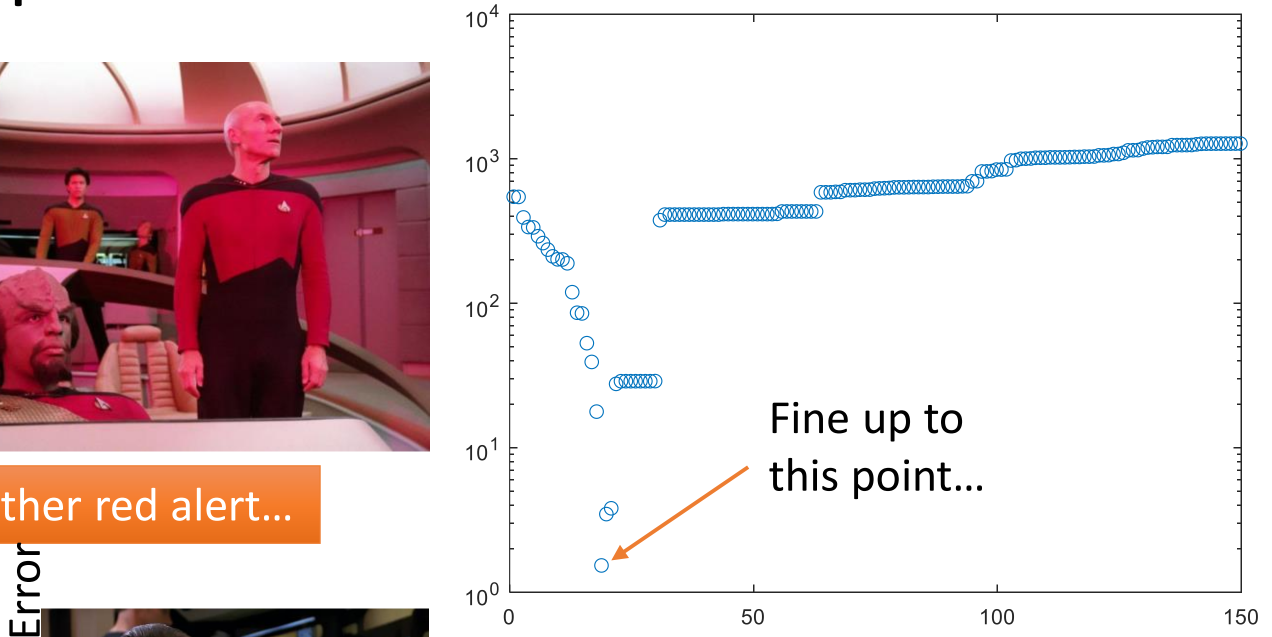
```
Initialize $t = \psi_0$
loop until convergence
    for i = 1,…,k-1
        $t = t - (v_i^T t) v_i$
    end
    $v_k = t/||t||$   $V_k = [v_1|v_2|\ldots|v_k]$
    $M_k = V_k^T H V_k$
    compute largest eigenpair of $M_k = Y \Theta Y^T$
    $\psi = V_k y$
    $r = H\psi - \theta\psi$
    if || r || is small enough, stop
    solve (approximately) t from $(I - \psi\psi^T)(H - \theta I)(I - \psi\psi^T)t = -r$
end loop
```

Run this loop twice

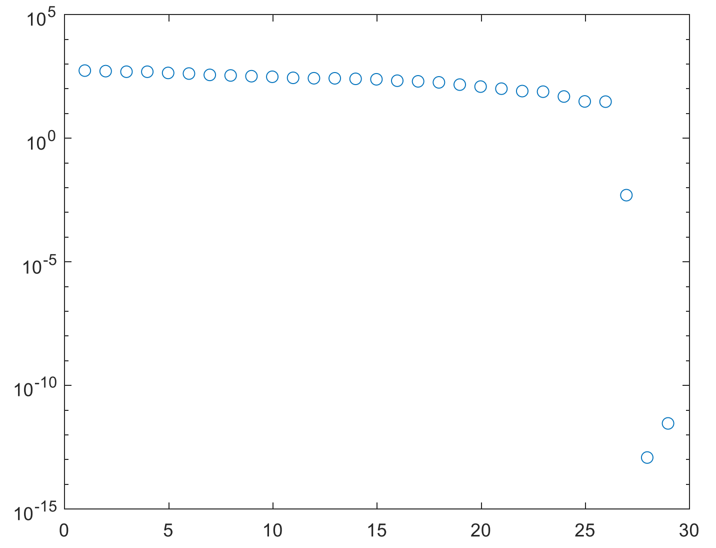
Another red alert…

Seems we have lost orthogonality again

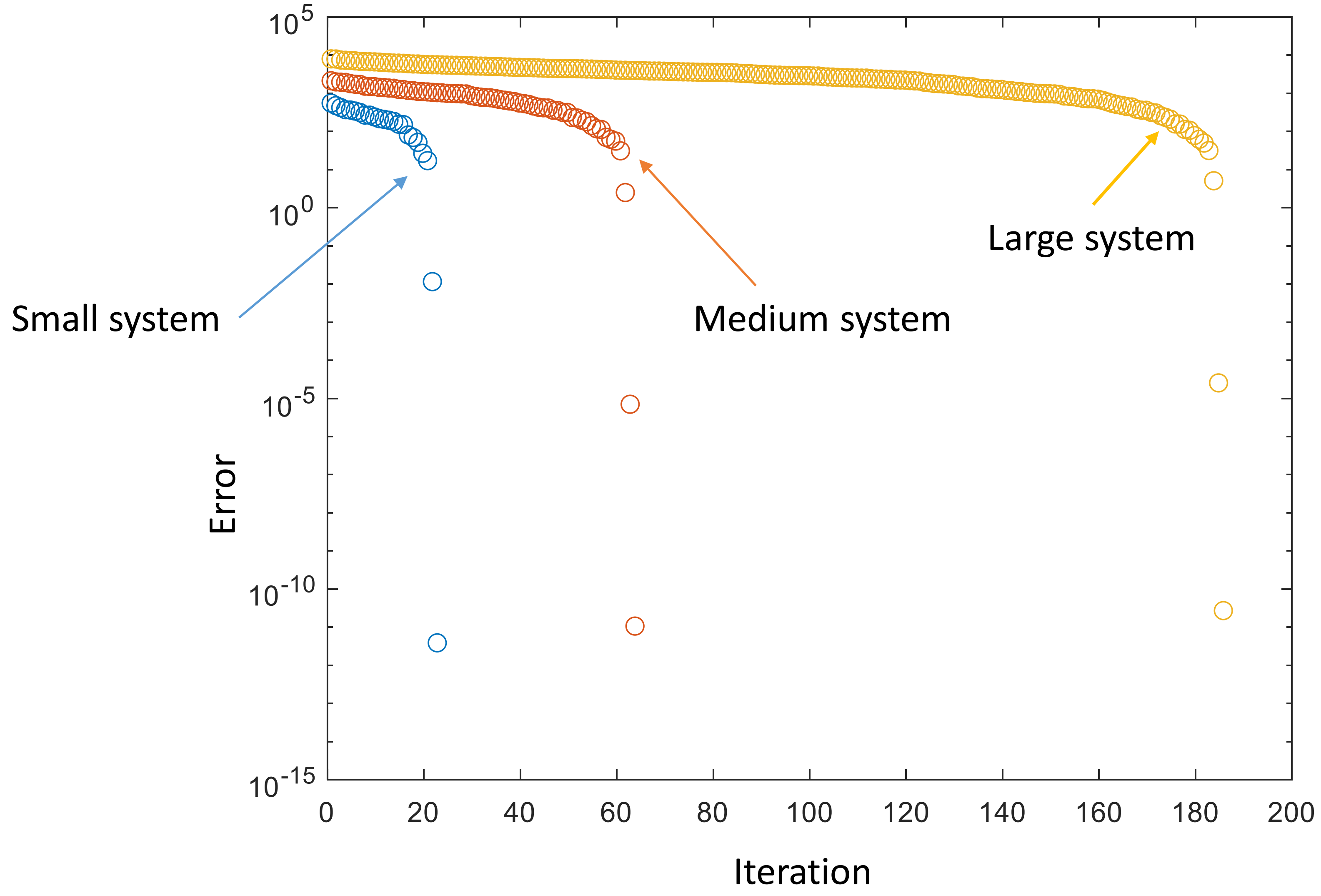
First try:



After reorthogonalization:



Jacobi-Davidson in Action: Size Invariance



Contrary to Lanczos, there is a plateau after which solution is found fast → Jacobi-Davidson benefits from a good starting guess

Eigenvalue Problems: More Solutions

- So far we have seen ways to compute maximal / extremal eigenpairs. However, this is not always enough.
  1. If interior eigenvalues are needed, shift-and-invert techniques should be used, i.e., solve eigenpairs of $(H - \sigma I)^{-1}$
  2. If more eigenpairs are needed
     - Converged eigenvalues can be deflated: $\tilde{H} = H - \epsilon_i \psi_i \psi_i^T$
     - Block versions of the algorithms can be used but then orthogonality must be maintained
  3. If errors / residuals diverge
     - Reorthogonalization helps
     - Restarting is a viable option since better starting vectors imply better convergence

Discuss: What are the differences between *$Ax = b$* and *$H\psi = \varepsilon\psi$*?