Solving linear system

Francesco Sanfilippo

Istituto Nazionale di Fisica Nucleare - Sezione Roma Tre

27 - 31 March 2017

Contents of Day 1

Introduction of the problem

1. Direct solution of linear system $A\,x = b$
2. Quadratic functional minimization

Iterative solver

1. Advantages
2. Comparison of efficiency

Contents of Day 2

Checking the solution

1. Limits, stability and efficiency of various algorithms
2. Convergence criterions

Accelerating the convergence

1. Mixed precision algorithms
2. Choosing a starting guess
3. Preconditioning the problem

Solving similar problems at the same time

1. Shifted problems $A + \sigma \mathrm{Id}$
2. Deflating the problem

Contents of Day 3

Review of Parallelisation

1. Distributed memory
2. Shared memory
3. Vectors

Gather/scatter approaches

$\rightarrow$ 1+2 different examples of gathering of non-local data

Contents of Day 4

More specifically on parallelisation

1. Communication/computation overlap
2. Multithreading
3. Vectorization

An example of a physical application

$\rightarrow$ Lattice QCD

Implicit and explicit residue

Implicit residue

- The value of the residue can be implicitly computed looking $|r_k|$
- This vector is automatically computed implicitly with: $r_{k+1} = r_k - \alpha_k p_k$

Explicit residue

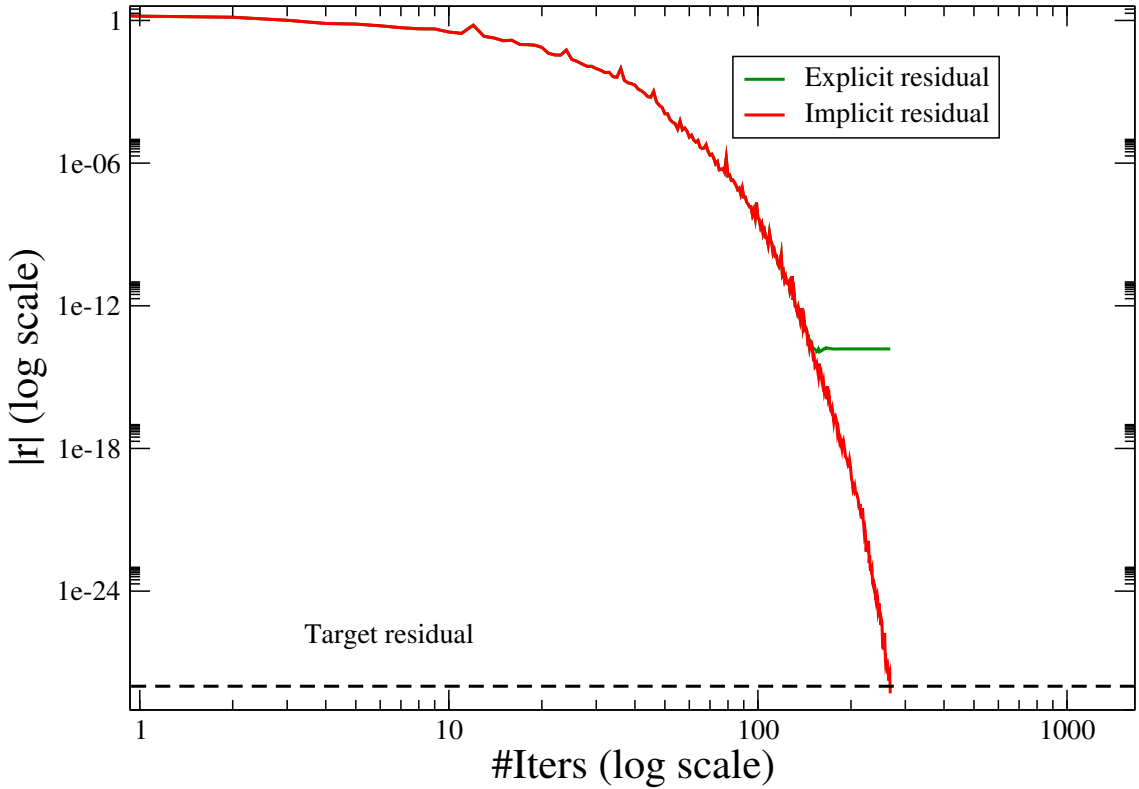
- One can also compute $r_k^{expl} = \boldsymbol{A}x_k - b$ at the cost of an **additional application**
- This way is possible to compute F as well

Numerical rounding

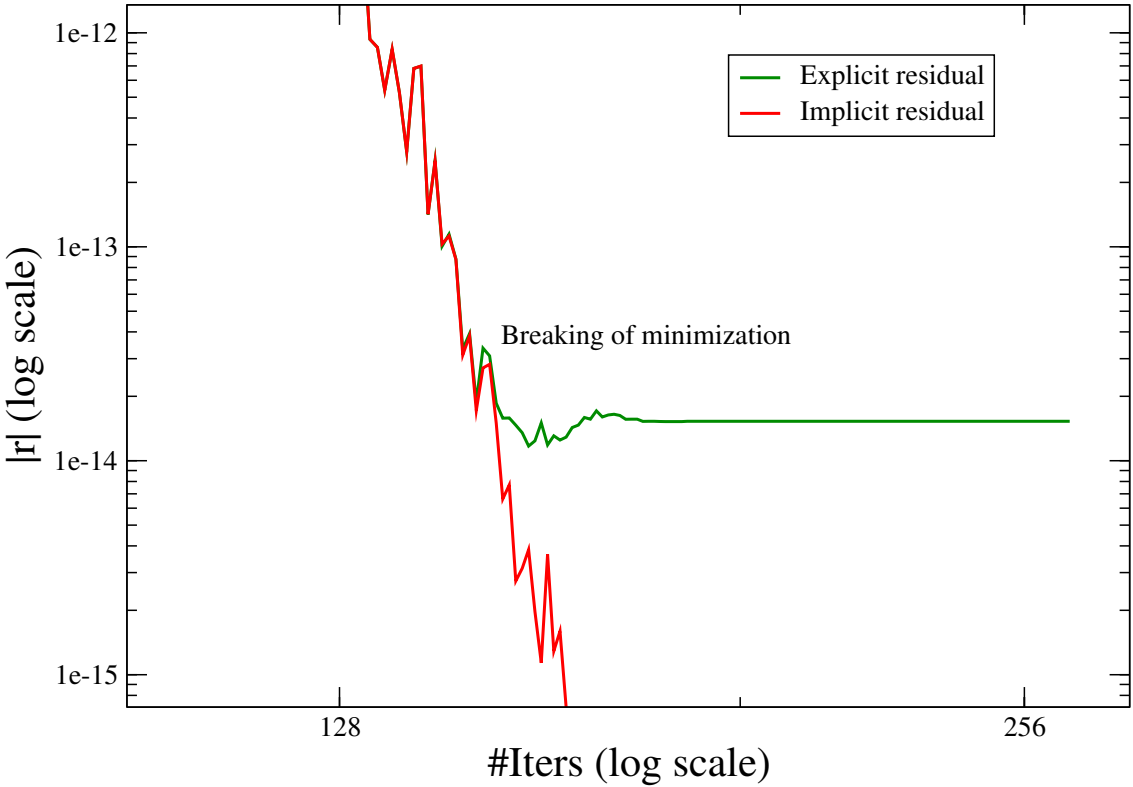
- As an effect of error accumulation, $\left| r_k^{expl} - r_k^{impl} \right|$ will grow with k
- This manifests the increase breaking of automatic conjugacy of generated p_k
- After a certain number of iterations the whole mechanism of minimization of F breaks

You cannot decrease $|r|$ arbitrarily!

Implicit and explicit residue in Conjugate gradient, $|r|_{targ} = 10^{-28}$, $n = 140$



Implicit and explicit residue in Conjugate gradient, $|r|_{targ} = 10^{-28}$, $N = 140$



Improving the result beyond machine precision

Split the system $\boldsymbol{A}x = b \quad \rightarrow \quad \boldsymbol{A}(x_0 + x_1 + \ldots x_{n-1}) = b$

- First set $b_0 = b$ and obtain an approximated solution x_0 such that

$$\boldsymbol{A}x_0 = b_0 \quad (+r_0)$$

with a target relative residual $|r_0| / |b_0| = \hat{r}_{targ}$ reachable by the conjugate gradient

- Then solve for $b_1 = -r_0$:

$$\boldsymbol{A}x_1 = b_1 \quad (+r_1)$$

where the relative residual $|r_1| / |b_1| = |r_0| / |b_0|$ is reachable by the conjugate gradient

- Iterate until $|r_{n-1}| / |b| = \hat{r}_{targ}$
- The solution is given by $x = x_0 + x_1 + \cdots + x_{n-1}$

Problem

- When the residue $|r| \ll |b|$ (high precision reached) r comes from a big cancellation
- If $|r| / |b| \sim \epsilon$ (machine precision $\sim 10^{-16}$), r_0^{expl} is badly computed and differs from true r_0
- x_1 obtained from $\boldsymbol{A}x_1 = b_1$ will not to improve x_0:

$$\boldsymbol{A}(x_0 + x_1) = b + r_0 - r_0^{expl} \neq b$$

Mixed precision

Higher precision

- Modern libraries offer support for emulated higher precision algebra
- e.g. gnu complier __float128 type implements quadruple precision (128 bit), $\epsilon \sim 10^{-32}$

Efficiency

- Emulation has a cost (order of magnitudes slower than hardware implementation)
- Running the whole solver in quadruple precision is costly and inefficient

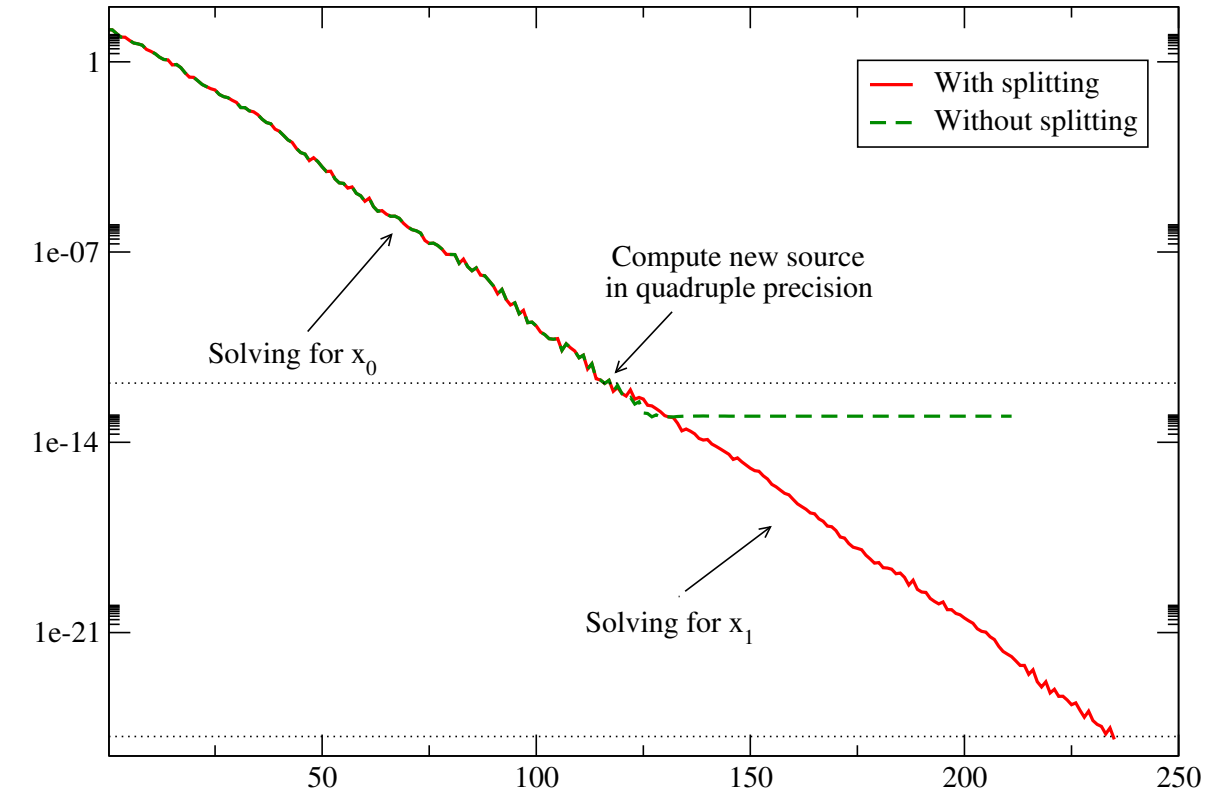
Solution

- Problem comes from the fact that $r_0^{expl} \neq r_0$, so that

$$\boldsymbol{A}(x_0 + x_1) = b + r_0 - r_0^{expl} \neq b$$

- Only need to compute r_0^{expl} accurately, so that $\left| r_0 - r_0^{expl} \right| \ll |r_0|$
- Quadruple precision is used only once in a while:
  - to compute $b = -r_i$
  - to sum together $x = x_0 + x_1 + \ldots x_{n-1}$
- Ordinary precision is used through all the rest (inner solver)

Mixed precision



Mixed precision

Acceleration via lower precision

The same game can be played the other way around:

- Use single (or even lower) precision in the inner solver
- Accumulate the external solution in higher precision

Advantages

- Some architectures (most notably, GPU) lacks hardware support to double precision, or have much faster single precision
- Data needed to compute $\boldsymbol{A}x$ occupy less memory, so
  - more data coherence (optimize cache access)
  - faster loading from memory
- When data is split across different computing nodes, less communication is needed

Disadvantages

- Need to store more information (lower and higher precision)
- At each restart Krylov space must be regenerated, can introduce unneeded overhead

Initial guess

- So far we assumed to start from an initial guess $x_0 = 0$ and iterate
- How can we incorporate a preliminary knowledge of an approximate solution x_{appr}, ?

Splitting the problem again

Decompose the total solution

$$\boldsymbol{A}(x - x_{appr} + x_{appr}) = b$$

so that the problem can be rewritten as:

$$\boldsymbol{A}\Delta x = c$$

with:

$$c = b - r_{appr}, \quad r_{appr} = \boldsymbol{A}x_{appr}, \quad \Delta x = x - x_{appr}$$

Recombine the problem

- After solving for Δx we sum $x = x_{appr} + \Delta x$
- Equivalent to set from the beginning $x_0 = x_{appr}$

Which guess

For example we could have solved a similar system, such as:

$$\boldsymbol{A}x = b' \quad \text{or} \quad \boldsymbol{A}'x = b$$

Non symmetric definite positive matrices

Condition for convergence

- Minimization of the functional is guaranteed by the fact that $F(x) = \frac{1}{2}x^T \boldsymbol{A}x - bx$ represents a paraboloid in a N-dimensional space, and $x = \boldsymbol{A}^{-1}b$ locate the minimum
- Eigenvalues of $\boldsymbol{A}$ must be real and positive ($\boldsymbol{A}$ is symmetric and definite positive)
- What to do if $\boldsymbol{A}$ does not satisfy the conditions?

Conjugate Gradient Normal Equation (CGNE)

- For whatever $\boldsymbol{A}$ (even rectangular) the matrix $\boldsymbol{B} = \boldsymbol{A^t A}$ is symmetric and definite positive, in facts eigenvalues of $\boldsymbol{B}$ are the square modulo of those of $\boldsymbol{A}$
- Equation obtained multiplying both sides of $\boldsymbol{A}x = b$ by $\boldsymbol{A^t}$ has the same solution x:

$$\boldsymbol{A^t A}x = \boldsymbol{A^t}b$$

- Get x by solving $\boldsymbol{B}$ for the vector $c = \boldsymbol{A}^t b$, as: $\boldsymbol{B}x = c$

Disadvantages

- Matrix $\boldsymbol{B} = \boldsymbol{A^t A}$ condition number is the square of that of $\boldsymbol{A}$
- Applying $\boldsymbol{B}$ cost: twice of applying $\boldsymbol{A}$ to x

Variation: biconjugate (stabilized) gradient algorithm

Problem

- When $\boldsymbol{A}$ is not symmetric & positive definite $\boldsymbol{A}$ is not a quadratic form
- (when the space is complex, symmetric$\rightarrow$ hermitian)
- Minimization breaks down

Biconjugate stabilized

- Instead of minimizing F we can find other functional forms
- Biconjugate stabilized algorithm minimizes also non-symmetric definite positive matrix
- Pro:
  - Works for generic matrix
  - Convergence typically requires less iterations
- Con:
  - Two applications of $\boldsymbol{A}$ per iteration needed
  - Is more turbulent (residue can jump)
  - Convergence is not guaranteed in all the cases

Other options

- Minimum residual method (MINRES), Generalized minimal residual method (GMRES) works to minimize $|r_k|$ at each iteration
- Symmetric LQ method (SYMMLQ)

Convergence can be faster, but typically is never guaranteed for generic problem

Preconditioning the problem

Preconditioner

Introduce an operator $\boldsymbol{P}$ such that:

- $\kappa\left(\boldsymbol{A}\boldsymbol{P}^{-1}\right) < \kappa\left(\boldsymbol{A}\right)$ the product $\boldsymbol{A}\boldsymbol{P}^{-1}$ has a lower condition number of $\boldsymbol{A}$
- computing $\boldsymbol{A}\boldsymbol{P}^{-1}x$ is not much more expensive than computing $\mathbf{A}x$

Preconditioned equation

$$\boldsymbol{A}\boldsymbol{P}^{-1}\,\boldsymbol{P}x = b$$

- Defining $y = \boldsymbol{P}x$ solve: $\boldsymbol{A}\boldsymbol{P}^{-1}y = b$
- Then solve: $\boldsymbol{P}x = y$ and obtain x

Definite-positivity

- If $\boldsymbol{P}^{-1}$ does not commute with $\boldsymbol{A}$, $\boldsymbol{A}\boldsymbol{P}^{-1}$ is not symmetric (even if $\boldsymbol{A}$ is)
- Solve the problem for $\boldsymbol{C} = \boldsymbol{P}^{-1}\boldsymbol{A}\boldsymbol{P}^{-1}$ and $d = \boldsymbol{P}^{-1}b$:

$$\boldsymbol{C}y = d$$

Alternative precontioned equation: just multiply the matrix from the left

$$\boldsymbol{P}^{-1}\boldsymbol{A}x = \boldsymbol{P}^{-1}b = c$$

Which preconditioner?

Desiderata

- System for $\boldsymbol{A}\boldsymbol{P}^{-1}$ (or $\boldsymbol{P}^{-1}\boldsymbol{A}$) must be simpler to be solved than $\boldsymbol{A}$
- $\boldsymbol{P}$ should therefore approximate $\boldsymbol{A}$ in some sense
- Computing $\boldsymbol{P}^{-1}v$ must be cheap

Generic possibilities

- Jacobi preconditioner $\boldsymbol{P} = \mathrm{diag}\,(\boldsymbol{A})$
  good if the system is dominated by its diagonal
- Incomplete LU factorization: we force matrix $\boldsymbol{U}$ to be zero where $\boldsymbol{A}$ was
  good for sparse matrices

More specific possibilities for sparse matrices

- Consider a simplified version of the matrix retaining its "physical" features
- Domain decomposition: consider separately sub-blocks of $\boldsymbol{A}$
  (of size possibly related to a physical property of the system)

Focus on a particular problem - Laplace equation in D dimensions

In the continuum

Setting $\sigma = const$, $\Delta = \sum_\mu \nabla_\mu^2$:

$$(\sigma - \Delta) f(x) = b(x)$$

Discretization

$$\begin{cases} f(x) & \to & f_i \\ b(x) & \to & b_i \\ \left(\nabla_\mu^2 f\right)(x) & \to & \frac{1}{2}\left(f_{i+\hat{\mu}} + f_{i-\hat{\mu}} - 2f_i\right) \end{cases}$$

Discrete problem

$$\boldsymbol{M}_{ij} f_j = b_i$$

Reduced to a linear system for matrix $\boldsymbol{M}$:

$$\boldsymbol{M}_{ij} = (\sigma + D)\,\boldsymbol{\delta}_{ij} - \frac{1}{2}\sum_\mu \left(\boldsymbol{\delta}_{i,j+\hat{\mu}} + \boldsymbol{\delta}_{i,j-\hat{\mu}}\right)$$

Laplace equation in D dimensions: exact solution

In x space

$$\boldsymbol{M}_{ij} f_j = b_i, \qquad \boldsymbol{M}_{ij} = (\sigma + D)\,\boldsymbol{\delta}_{ij} - \frac{1}{2}\sum_{\mu} \left(\boldsymbol{\delta}_{i,j+\hat{\mu}} + \boldsymbol{\delta}_{i,j-\hat{\mu}}\right)$$

Passing to Fourier space

$$\begin{cases} f_j = \frac{1}{V}\sum_k e^{ijk} f\,(k) \\ \boldsymbol{M}_{jl} = \frac{1}{V}\sum_k e^{i(j-l)k} \end{cases} \quad \left(\sigma + D - \sum_{\mu} \cos k_{\mu}\right)$$

Exact solution

$$f_j = \sum_k e^{-ijk} \boldsymbol{M}^{-1}\,(k)\, b\,(k) = \sum_k \frac{e^{-ijk} b\,(k)}{\sigma + D - \sum_{\mu} \cos k_{\mu}}$$

Good to check approximate solution!

Properties of the discrete Laplace operator

Eigenvalues, eigenvectors

- $\boldsymbol{M}_{ij} = (\sigma + D)\,\boldsymbol{\delta}_{ij} - \frac{1}{2}\sum_{\mu}\left(\boldsymbol{\delta}_{i,j+\hat{\mu}} + \boldsymbol{\delta}_{i,j-\hat{\mu}}\right)$ is symmetric
- Due to translation invariance, plane waves are eigenvectors
- Eigenvalues labelled by corresponding momenta:

$$\lambda(k) = \sigma + D - \sum_{\mu} \cos k_{\mu}$$

- If $\sigma = 0$ the system has a zero eigenvalue

Matricially

For $N = 6$, $D = 1$ ($d = \sigma + D$, $s = -1/2$)

$$\begin{pmatrix} d & s & 0 & 0 & 0 & s \\ s & d & s & 0 & 0 & 0 \\ 0 & s & d & s & 0 & 0 \\ 0 & 0 & s & d & s & 0 \\ 0 & 0 & 0 & s & d & s \\ s & 0 & 0 & 0 & s & d \end{pmatrix}$$

Decomposing the problem - Even Odd precondition

Splitting parity coupling (red-black)

$$\boldsymbol{M} = \boldsymbol{M}^{ee} + \boldsymbol{M}^{oe} + \boldsymbol{M}^{eo} + \boldsymbol{M}^{oo}$$

where e.g. M^{eo} is different from 0 only between even and odd sites:

$$\boldsymbol{M}^{eo}_{i,j} = \boldsymbol{M}_{i,j}\delta_{p(i),e}\delta_{p(j),o}, \quad p(i) = \left(\sum_{\mu} x_i\right) \bmod 2, \quad e = 0,\ o = 1$$

and similarly

$$v = v^e + v^o$$

Rewriting the system

In this way the system is rewritten as $\begin{cases} \boldsymbol{M}^{ee}x^e + \boldsymbol{M}^{eo}x^o &= b^e \\ \boldsymbol{M}^{oe}x^e + \boldsymbol{M}^{oo}x^o &= b^o \end{cases}$

Decoupling even solution

- Some algebra shows: $\begin{cases} \left(\boldsymbol{M}^{ee} - \boldsymbol{M}^{eo}\frac{1}{\boldsymbol{M}^{oo}}\boldsymbol{M}^{oe}\right)x^e = b^e - \boldsymbol{M}^{eo}\frac{1}{\boldsymbol{M}^{oo}}b^o \\ x^o = \frac{1}{\boldsymbol{M}^{oo}}\left(b^o - \boldsymbol{M}^{oe}x^e\right) \end{cases}$
- The equation for x^e decouples from that of x_0

Advantages of solving x^e

$$\left(\boldsymbol{M}^{ee} - \boldsymbol{M}^{eo}\frac{1}{\boldsymbol{M}^{oo}}\boldsymbol{M}^{oe}\right)x^e = b^e - \boldsymbol{M}^{eo}\frac{1}{\boldsymbol{M}^{oo}}b^o$$

Cost

Application of $\boldsymbol{M}^{eo}\frac{1}{\boldsymbol{M}^{oo}}\boldsymbol{M}^{oe}$ and $\boldsymbol{M}^{ee}$ has the same cost of $\boldsymbol{M}$ but:

- $\boldsymbol{M}^{oo}_{i,j} = \delta_{i,j}d$ so $\frac{1}{\boldsymbol{M}^{oo}_{i,j}} = \frac{1}{d}\boldsymbol{M}_{i,j}$
- half of the data is necessary (less communications, more data coherence)
- the condition number has decreased

Further preconditioning

- Additional optimization can be achieved factorizing $\boldsymbol{M}^{ee}$

$$\left(1 - \frac{1}{\boldsymbol{M}^{ee}}\boldsymbol{M}^{eo}\frac{1}{\boldsymbol{M}^{oo}}\boldsymbol{M}^{oe}\right)x^e = \frac{1}{\boldsymbol{M}^{ee}}\left(b^e - \boldsymbol{M}^{eo}\frac{1}{\boldsymbol{M}^{oo}}b^o\right)$$

Solver

The system is still symmetric, so conjugate gradient solver usable

Visualization

Reordering

Putting first all even sites and then odd,

$$M = \begin{pmatrix} M^{ee} & M^{oe} \\ M^{eo} & M^{oo} \end{pmatrix}, \quad v = \begin{pmatrix} v_e \\ v_o \end{pmatrix}$$

Good for performance

Not necessary to perform this reordering, but improve performance

$$\begin{pmatrix} d & s & 0 & 0 & 0 & s \\ s & d & s & 0 & 0 & 0 \\ 0 & s & d & s & 0 & 0 \\ 0 & 0 & s & d & s & 0 \\ 0 & 0 & 0 & s & d & s \\ s & 0 & 0 & 0 & s & d \end{pmatrix} \rightarrow \begin{pmatrix} d & 0 & 0 & 0 & s & s \\ 0 & d & 0 & s & 0 & s \\ 0 & 0 & d & s & s & 0 \\ 0 & s & s & d & 0 & 0 \\ s & 0 & s & 0 & d & 0 \\ s & s & 0 & 0 & 0 & d \end{pmatrix}$$

Access to a more compact piece of memory

Domain decomposition

- What we saw is just an example of a Schur decomposition on the 2×2 blocked matrix
- Other decompositions can come to your mind
- In our case for example

$$\left(\begin{array}{cccccc} d & s & 0 & 0 & 0 & s \\ s & d & s & 0 & 0 & 0 \\ \underline{0} & \underline{s} & \underline{d} & \underline{s} & \underline{0} & \underline{0} \\ 0 & 0 & s & d & s & 0 \\ 0 & 0 & 0 & s & d & s \\ s & 0 & 0 & 0 & s & d \end{array}\right) = \left(\begin{array}{cc} D_{11} & \Omega_{12} \\ \Omega_{21} & D_{22} \end{array}\right)$$

- In this case

$$\left(\boldsymbol{D}_{11} - \boldsymbol{\Omega}_{12} \frac{1}{\boldsymbol{D}_{22}} \boldsymbol{\Omega}_{21}\right) x^1 = b^1 - \boldsymbol{\Omega}_{12} \frac{1}{\boldsymbol{D}_{22}} b^2$$

- Computing $t = \frac{1}{\boldsymbol{D}_{22}} v$ requires actually to solve the $\boldsymbol{D}_{22} t = v$ (micro-system) but
  - the system is much smaller
  - in a parallel program, no communication between different domains needed
- Can be seen as "gluing together" the solution obtained on subdomain, and correcting it

  Pro if sub-blocks size correctly chosen, few macro iteration needed

  Con have to know the scale at which the micro-solution approximate well the total one

Solving multiple similar systems

Naive solution

Use the solution of a problem as first guess for the next one

Shifted solver

If the problem is of the kind

$$(\boldsymbol{A} + \sigma)\, x = b$$

multi-shifted solver exists, e.g. M-CG, that can solve all the system for σ_1, σ_2, ... simultaneously with very little overhead

Deflation

If we are interested in multiple source b_1, b_2 ... use algorithm (EigCG...) that

- while solving $\boldsymbol{A}x = b$ finds the lowest eigenvector oft the system
- for each sources b_1, b_2, ... one eigenvector is found
- and can be removed by the spectra
- accelerating following solution