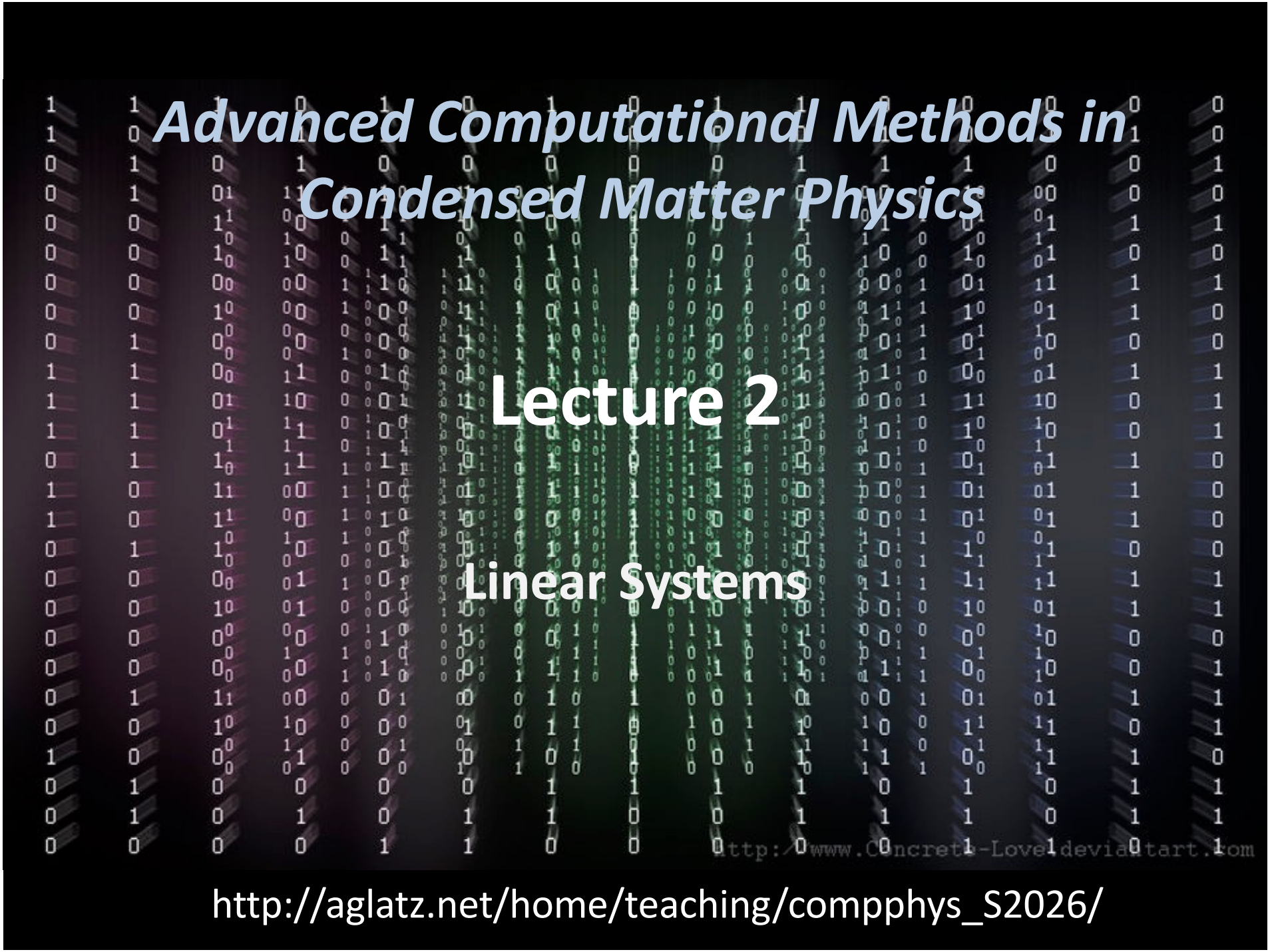

Advanced Computational Methods in Condensed Matter Physics

Lecture 2

Linear Systems

http://aglatz.net/home/teaching/compphys_S2026/

Linear equation systems

Solving linear equation systems is a quite common task during a simulation. Seemingly simple, there are many challenges to do so on a computer.

- m linear equations with n unknowns, x_i:

$$\sum_{j=1}^{m} a_{ij} x_j = b_i, \;\; i = 1, \ldots, m$$

- In matrix form:

$$A\mathbf{x} = \mathbf{b}$$

where the coefficients a_{ij} form the matrix $A \in \mathbb{C}^{m \times n}$, and the rhs b_i the vector $\mathbf{b} \in \mathbb{C}^m$.

Solution methods

Here we consider real square matrices with rank(A)=n, i.e., det(A)≠0

Formal solution (Cramer's rule): $$x_j = \frac{\Delta_j}{\det(A)},\ j = 1, \ldots, n$$

Where Δ_j is the determinant of the matrix obtained by replacing column j of A by **b**. If this would be implemented, if would be an O((n+1)!) algorithm!!! Or take 10^{46} years to solve 50 equations on a modern computer (100Gflop/s)….

The problem to solve this equation system is related to the problem of inverting a square matrix, since the solution can be written as

$$\mathbf{x} = A^{-1}\mathbf{b}$$

What methods exist to solve it?

alternatives to Cramer's rule:

1. ***direct methods****:* yield the solution of the system in a finite number of steps
2. ***iterative methods:*** require (theoretically) an infinite number of steps.

The choice between a direct and an iterative method depends

- on the theoretical efficiency of the scheme
- the particular type of matrix
- on memory storage requirements
- on the architecture of the computer

Accuracy?

Warning: Solving a linear system by a numerical method invariably leads to the introduction of rounding errors.
→ *We will discuss this in the chapter about linear stability.*

Outlook: An important measure for the accuracy of the numerical solution is the condition number of a matrix:

$$K(A) \equiv ||A|| \cdot ||A^{-1}|| \geq 1$$

For the Euclidean norm and symmetric, positive definite matrices:

$$K(A) = \lambda_{max}/\lambda_{min}$$

with λ_{max} and λ_{min} being the largest/smallest eigenvalue of A.

If the condition number is close to one, the matrix is well conditioned → its inverse can be computed with good accuracy.

Direct methods

Triangular matrices

Consider the non-singular, lower triangular 3x3 matrix:

$$\begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

→

$$\begin{aligned} x_1 &= b_1/l_{11}, \\ x_2 &= (b_2 - l_{21}x_1)/l_{22}, \\ x_3 &= (b_3 - l_{31}x_1 - l_{32}x_2)/l_{33} \end{aligned}$$

Can be extended to systems n × n: *forward substitution algorithm*

$$\begin{aligned} x_1 &= \frac{b_1}{l_{11}}, \\ x_i &= \frac{1}{l_{ii}} \left(b_i - \sum_{j=1}^{i-1} l_{ij} x_j \right), \quad i = 2, \ldots, n \end{aligned}$$

$O(n^2)$ algorithm

...

Equivalent for upper triangular matrix [U**x**=**b**]: *backward substitution*

$$x_n = \frac{b_n}{u_{nn}},$$

$$x_i = \frac{1}{u_{ii}} \left(b_i - \sum_{j=i+1}^{n} u_{ij} x_j \right), \quad i = n-1, \ldots, 1$$

Algorithms (MatLab code)

```
function [b]=forward_col(L,b)
[n]=mat_square(L);
for j=1:n-1,
    b(j)= b(j)/L(j,j); b(j+1:n)=b(j+1:n)-b(j)*L(j+1:n,j);
end; b(n) = b(n)/L(n,n);
```

```
function [b]=backward_col(U,b)
[n]=mat_square(U);
for j = n:-1:2,
    b(j)=b(j)/U(j,j); b(1:j-1)=b(1:j-1)-b(j)*U(1:j-1,j);
end; b(1) = b(1)/U(1,1);
```

Gaussian elimination (GE)

Gaussian Elimination:

- Reduce A**x**=**b** to an equivalent system (that is, having the same solution) of form U**x**=$\mathcal{b}$
  U: upper triangular matrix, $\mathcal{b}$: updated right side vector.
- The latter system can then be solved by backward substitution
- Let us denote the original system by $A^{(1)}\mathbf{x} = \mathbf{b}^{(1)}$

1. Introduce the multipliers:

$$m_{i1} = \frac{a_{i1}^{(1)}}{a_{11}^{(1)}}, \quad i = 2, 3, \ldots, n$$

2. Eliminate the unknown x_1 in the following rows i below row 1:

$$a_{ij}^{(2)} = a_{ij}^{(1)} - m_{i1} a_{1j}^{(1)}, \quad i, j = 2, \ldots, n,$$

$$b_i^{(2)} = b_i^{(1)} - m_{i1} b_1^{(1)}, \quad i = 2, \ldots, n,$$

➔

$$\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \ldots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \ldots & a_{2n}^{(2)} \\ \vdots & \vdots & & \vdots \\ 0 & a_{n2}^{(2)} & \ldots & a_{nn}^{(2)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(2)} \\ \vdots \\ b_n^{(2)} \end{bmatrix} \quad \Leftrightarrow A^{(2)}\mathbf{x} = \mathbf{b}^{(2)}$$

...

Then eliminate x_2 from rows 3,…,n, etc.
In general after k-1 elimination steps, we have a system: $\mathrm{A}^{(k)}\mathbf{x} = \mathbf{b}^{(k)}, \quad 1 \le k \le n,$

$$\mathrm{A}^{(k)} = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \dots & \dots & \dots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & & & & a_{2n}^{(2)} \\ \vdots & & \ddots & & & \vdots \\ 0 & \dots & 0 & a_{kk}^{(k)} & \dots & a_{kn}^{(k)} \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \dots & 0 & a_{nk}^{(k)} & \dots & a_{nn}^{(k)} \end{bmatrix}$$

And finally, we get:

$$\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \dots & \dots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & & & a_{2n}^{(2)} \\ \vdots & & \ddots & & \vdots \\ 0 & & & \ddots & \vdots \\ 0 & & & & a_{nn}^{(n)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(2)} \\ \vdots \\ \vdots \\ b_n^{(n)} \end{bmatrix} \Leftrightarrow \mathrm{U}\mathbf{x} = \mathcal{b}$$

We assumed $a^{(i)}_{ii} \neq 0$ (i=1,…,n-1). These elements are called pivots.

$O(n^3)$ algorithm

GE Example

3x3 Hilbert matrix:

$$(\mathrm{A}^{(1)}\mathbf{x} = \mathbf{b}^{(1)}) \quad \begin{cases} x_1 + \frac{1}{2}x_2 + \frac{1}{3}x_3 = \frac{11}{6} \\ \frac{1}{2}x_1 + \frac{1}{3}x_2 + \frac{1}{4}x_3 = \frac{13}{12} \\ \frac{1}{3}x_1 + \frac{1}{4}x_2 + \frac{1}{5}x_3 = \frac{47}{60} \end{cases}$$

$m_{21}=1/2$, $m_{31}=1/3$:

$$(\mathrm{A}^{(2)}\mathbf{x} = \mathbf{b}^{(2)}) \quad \begin{cases} x_1 + \frac{1}{2}x_2 + \frac{1}{3}x_3 = \frac{11}{6} \\ 0 + \frac{1}{12}x_2 + \frac{1}{12}x_3 = \frac{1}{6} \\ 0 + \frac{1}{12}x_2 + \frac{4}{45}x_3 = \frac{31}{180} \end{cases}$$

$m_{32}=1$:

$$(\mathrm{A}^{(3)}\mathbf{x} = \mathbf{b}^{(3)}) \quad \begin{cases} x_1 + \frac{1}{2}x_2 + \frac{1}{3}x_3 = \frac{11}{6} \\ 0 + \frac{1}{12}x_2 + \frac{1}{12}x_3 = \frac{1}{6} \\ 0 + 0 + \frac{1}{180}x_3 = \frac{1}{180} \end{cases}$$

→ $x_3=1$, $x_2=1$, $x_1=1$

General Hilbert matrix: $h_{ij}=1/(i+j-1)$; $i,j=1,\ldots,n$

pivots

GE only works if the pivots are finite.

There are classes of matrices, when GE is “safe”

- A is diagonally dominant by rows
- A is diagonally dominant by column
- A is symmetric and positive definite

If zero (or small) pivots are encountered, one can reorder the remaining rows of $A^{(k)}$ [$\mathbf{b}^{(k)}$ elements accordingly] in order to move the largest (absolute value) element to the pivot position and continue.

Pseudocode for GE with pivoting

```
for k = 1 ... m:
  //Find pivot for column k:
  i_max  := argmax (i = k ... m, abs(A[i, k]))
  if A[i_max, k] = 0
    error "Matrix is singular!"
  swap rows(k, i_max)
  //Do for all rows below pivot:
   for i = k + 1 ... m:
     //Do for all remaining elements in current row:
     for j = k ... n:
       A[i, j]  := A[i, j] - A[k, j] * (A[i, k] / A[k, k])
    //Fill lower triangular matrix with zeros:
    A[i, k]  := 0
```

LU decomposition

GE is equivalent to performing a factorization of the matrix A into the product of two matrices, A=LU, with $U=A^{(n)}$.

- L and U do not depend on **b** and can therefore be used to solve the linear system for different **b.**
  This means a reduction of computation time to $O(n^2)$
- Let us go back to the Hilbert matrix example to see how the matrix L is constructed:

define:

$$\mathrm{M}_1 = \begin{bmatrix} 1 & 0 & 0 \\ -\frac{1}{2} & 1 & 0 \\ -\frac{1}{3} & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -m_{21} & 1 & 0 \\ -m_{31} & 0 & 1 \end{bmatrix}$$

indeed:

$$\mathrm{M}_1\mathrm{A} = \mathrm{M}_1\mathrm{A}^{(1)} = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ 0 & \frac{1}{12} & \frac{1}{12} \\ 0 & \frac{1}{12} & \frac{4}{45} \end{bmatrix} = \mathrm{A}^{(2)}$$

...

and

$$\mathrm{M}_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -m_{32} & 1 \end{bmatrix}$$

therefore

$$\mathrm{M}_2\mathrm{M}_1\mathrm{A} = \mathrm{A}^{(3)} = \mathrm{U}$$

$$\mathrm{A} = (\mathrm{M}_2\mathrm{M}_1)^{-1}\mathrm{U} = \mathrm{LU}$$

In general

$$\mathbf{m}_k = (0, \ldots, 0, m_{k+1,k}, \ldots, m_{n,k})^T \in \mathbb{R}^n$$

$$\mathrm{M}_k = \begin{bmatrix} 1 & \ldots & 0 & 0 & \ldots & 0 \\ \vdots & \ddots & \vdots & \vdots & & \vdots \\ 0 & & 1 & 0 & & 0 \\ 0 & & -m_{k+1,k} & 1 & & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \ldots & -m_{n,k} & 0 & \ldots & 1 \end{bmatrix} = \mathrm{I}_n - \mathbf{m}_k\mathbf{e}_k^T$$

$$\mathrm{A}^{(k+1)} = \mathrm{M}_k\mathrm{A}^{(k)}$$

...

The complete elimination process is therefore: $\mathrm{M}_{n-1}\mathrm{M}_{n-2}\ldots\mathrm{M}_1\mathrm{A}=\mathrm{U}$

with $\mathrm{M}_k^{-1}=2\mathrm{I}_n-\mathrm{M}_k=\mathrm{I}_n+\mathbf{m}_k\mathbf{e}_k^T$

we get L from:

$$\begin{aligned}
\mathrm{A} &= \mathrm{M}_1^{-1}\mathrm{M}_2^{-1}\ldots\mathrm{M}_{n-1}^{-1}\mathrm{U}\\
&= (\mathrm{I}_n+\mathbf{m}_1\mathbf{e}_1^T)(\mathrm{I}_n+\mathbf{m}_2\mathbf{e}_2^T)\ldots(\mathrm{I}_n+\mathbf{m}_{n-1}\mathbf{e}_{n-1}^T)\mathrm{U}\\
&= \left(\mathrm{I}_n+\sum_{i=1}^{n-1}\mathbf{m}_i\mathbf{e}_i^T\right)\mathrm{U}\\
&= \begin{bmatrix} 1 & 0 & \ldots & \ldots & 0\\ m_{21} & 1 & & & \vdots\\ \vdots & m_{32} & \ddots & & \vdots\\ \vdots & \vdots & & \ddots & 0\\ m_{n1} & m_{n2} & \ldots & m_{n,n-1} & 1 \end{bmatrix}\mathrm{U}.
\end{aligned}$$

Once the matrices L and U have been computed, solving the linear system consists only of solving successively the two triangular systems:

$$\mathrm{L}\mathbf{y}=\mathbf{b}$$

$$\mathrm{U}\mathbf{x}=\mathbf{y}$$

LU implementation

Since L is a lower triangular matrix with diagonal entries equal to 1 and U is upper triangular, it is possible (and convenient) to store the LU factorization directly in the same memory area that is occupied by the matrix A. More precisely, U is stored in the upper triangular part of A (including the diagonal), whilst L occupies the lower triangular portion of A (the diagonal entries of L are not stored since they are implicitly assumed to be 1).

MatLab implementation

```
function [A] = lu_kji (A)
[n,n]=size(A);
for k=1:n-1
   A(k+1:n,k)=A(k+1:n,k)/A(k,k);
   for j=k+1:n, for i=k+1:n
                    A(i,j)=A(i,j)-A(i,k)*A(k,j);
   end,         end
end
```

...

One final remark on LU: If (partial) pivoting (exchange of rows) is used in GE a corresponding permutation matrix, P_i, needs to be inserted in the factorization of A, i.e.,

$$\mathrm{U} = \mathrm{A}^{(n)} = \mathrm{M}_{n-1}\mathrm{P}_{n-1} \ldots \mathrm{M}_1\mathrm{P}_1\mathrm{A}^{(1)}$$

Similarly for full pivoting, when also columns of the remaining sub-matrix are exchanged in order to move the element with largest absolute value to the pivot position.

Related factorizations:

- LDM^T factorization: L, M^T and D are lower triangular, upper triangular and diagonal matrices, respectively (L does not need to have a "1" diagonal)
- This gives for symmetric matrices: M=L, i.e., a LDL^T factorization
- Cholesky factorization for symmetric and positive definite matrices: $A=H^TH$, where H is a unique upper triangular matrix with positive diagonal elements
- QR factorization for rectangular matrices: A = QR, with $A\in\mathbb{R}^{m\times n}$ ($m\geq n$), orthogonal matrix $Q\in\mathbb{R}^{m\times m}$, and trapezoidal matrix $R\in\mathbb{R}^{m\times n}$ with zero rows n+1,…,m --- important for eigenvalue calculation of square matrices.

Special cases

In simulations the matrix A is often sparse, i.e., most elements zero.
In particular the have a band structure with finite diagonal elements and a few finite off-diagonals.

Tridiagonal matrices:
(occur e.g. when discretizing gradients and Laplacians)

$$\mathrm{A} = \begin{bmatrix} a_1 & c_1 & & 0 \\ b_2 & a_2 & \ddots & \\ & \ddots & & c_{n-1} \\ 0 & & b_n & a_n \end{bmatrix}$$

Then

$$\mathrm{L} = \begin{bmatrix} 1 & & & 0 \\ \beta_2 & 1 & & \\ & \ddots & \ddots & \\ 0 & & \beta_n & 1 \end{bmatrix} \quad \mathrm{U} = \begin{bmatrix} \alpha_1 & c_1 & & 0 \\ & \alpha_2 & \ddots & \\ & & \ddots & c_{n-1} \\ 0 & & & \alpha_n \end{bmatrix}$$

with

$$\alpha_1 = a_1, \quad \beta_i = \frac{b_i}{\alpha_{i-1}}, \quad \alpha_i = a_i - \beta_i c_{i-1}, \ i = 2, \ldots, n.$$

Thomas algorithm

O(k n) algorithm
(k number of finite off-diagonals)

Iterative methods

- Iterative methods formally yield the solution x of a linear system after an infinite number of steps.
- At each step they require the computation of the residual of the system.
- In the case of a full matrix, their computational cost is therefore of the order of n^2 operations for each iteration, to be compared with an overall cost of the order of $2/3n^3$ operations needed by direct methods.

→ Iterative methods can therefore become competitive with direct methods provided the number of iterations that are required to converge (within a prescribed tolerance) is either independent of n or scales sub-linearly with respect to n.

(Some) iterative methods can be parallelized!

Direct methods are typically sequential, and each step depends on the result of the previous one.

Main concept

The basic idea of iterative methods is to construct a sequence of vectors $\mathbf{x}^{(k)}$ that enjoy the property of convergence:

$$\mathbf{x} = \lim_{k \to \infty} \mathbf{x}^{(k)}$$

where **x** is the solution of A**x**=**b**

The iteration processes is stopped when $\|\mathbf{x}^{(n)} - \mathbf{x}\| < \varepsilon$ with a prescribed tolerance ϵ.

Problem with this conditions: Impractical, since we do not know **x**.

General scheme:

$$\mathbf{x}^{(0)} = \mathbf{f}_0(\mathrm{A}, \mathbf{b}),$$

$$\mathbf{x}^{(n+1)} = \mathbf{f}_{n+1}(\mathbf{x}^{(n)}, \mathbf{x}^{(n-1)}, \ldots, \mathbf{x}^{(n-m)}, \mathrm{A}, \mathbf{b}), \text{ for } n \geq m$$

Definitions

$$\mathbf{x}^{(0)} = \mathbf{f}_0(\mathrm{A}, \mathbf{b}),$$

$$\mathbf{x}^{(n+1)} = \mathbf{f}_{n+1}(\mathbf{x}^{(n)}, \mathbf{x}^{(n-1)}, \ldots, \mathbf{x}^{(n-m)}, \mathrm{A}, \mathbf{b}), \text{ for } n \geq m$$

In this general scheme f_i and $x^{(m)}$,…, $x^{(1)}$ are given functions and vectors, respectively.

- The number of steps which the current iteration depends on is called the *order of the method*.
- If the functions f_i are independent of the step index i, the method is called *stationary*, otherwise it is *non-stationary*.
- Finally, if f_i depends linearly on $x^{(0)}$, …, $x^{(m)}$, the method is called *linear*, otherwise it is *nonlinear*.

Linear iterative methods

Here we focus on stationary, linear iterative methods of order one.

- general technique: additive splitting of matrix A of form A=P−N
- P and N are two suitable matrices and P is nonsingular
- P is called preconditioning matrix or preconditioner

Here we consider an iteration of the form

$$\mathbf{x}^{(0)} \text{ given}, \quad \mathbf{x}^{(k+1)} = \mathrm{B}\mathbf{x}^{(k)} + \mathbf{f}, \quad k \geq 0$$

- B is an n x n square matrix called the iteration matrix
- **f** is a vector obtained from the right-hand side **b**
- Consistent with A**x**=**b** if **f**=(I-B)A^{-1}**b**

Using the above splitting of A, we calculate $\mathbf{x}^{(k)}$ for k>0, solving

$$\mathrm{P}\mathbf{x}^{(k+1)} = \mathrm{N}\mathbf{x}^{(k)} + \mathbf{b}, \quad k \geq 0$$

i.e., B=P^{-1}N and **f**=P^{-1}**b**

...

This scheme can be written as $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathrm{P}^{-1}\mathbf{r}^{(k)}$

with the ***residual*** $\mathbf{r}^{(k)} = \mathbf{b} - \mathrm{A}\mathbf{x}^{(k)}$

We note that:

1. P should be chosen such that it can be easily inverted
2. If P=A and N=0, the iteration would converge in one step
3. The residual is a measure of how good $\mathbf{x}^{(k)}$ approximates the real solution **x**

Jacobi iteration

If the diagonal entries of A are nonzero, we can single out in each equation the corresponding unknown on the diagonal and write:

$$x_i = \frac{1}{a_{ii}} \left[b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j \right], \qquad i = 1, \ldots, n$$

In the Jacobi method $\mathbf{x}^{(k+1)}$ is computed by [$\mathbf{x}^{(0)}$ can be an arbitrary initial guess]

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left[b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k)} \right], \quad i = 1, \ldots, n$$

This corresponds to a splitting: P=D, N=D-A=E+F,

- D is a diagonal matrix having the diagonal elements of A
- E is the lower triangular matrix with elements: $e_{ij}=-a_{ij}$ for i>j, 0 else
- F the upper triangular matrix: $f_{ij}=-a_{ij}$ for i<j, 0 else

A generalization is the over-relaxation method (JOR): $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \omega \mathrm{D}^{-1} \mathbf{r}^{(k)}$

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}} \left[b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k)} \right] + (1-\omega) x_i^{(k)}, \qquad i = 1, \ldots, n$$

where ω is a relaxation parameter $0<\omega \leq 1$

...

Remarks:

- In the Jacobi method P=D can be easily inverted
- Each iteration step required therefore only one matrix multiplication, i.e., $A\mathbf{x}^{(k)}$
- Therefore it can be easily parallelized
- The method converges when A is strictly diagonally dominant, i.e., $|a_{ii}|$ is larger than the sum of all other absolute values of the elements in the row
- Standard convergence criterion: $\rho(D^{-1}N)<1$ (ρ is the spectral radius, i.e., the largest absolute value of this eigenvalues)
- Jacobi is convergent if A and (2D-A) are symmetric and positive definite
- The above convergence criterions are not always necessary for convergence…

Jacobi algorithm

Choose an initial guess $x^{(0)}$ to the solution
k = 0
check if convergence is reached, e.g., $||\mathbf{r}(k)||_\infty < \epsilon$
while convergence not reached do
  for i := 1 step until n do
    $\sigma = 0$
    for j := 1 step until n do
      if $j \neq i$ then
        $\sigma = \sigma + a_{ij}\, x_j^{(k)}$
      end if
    end (j-loop)
    $x_i^{(k+1)} = (b_i - \sigma)/ a_{ii}$
  end (i-loop)
  check if convergence is reached
  k = k + 1
loop (while convergence condition not reached)

Gauss-Seidel iteration

The Gauss-Seidel method differs from the Jacobi method in the fact that at the (k+1)-th step the available values of $x_i^{(k+1)}$ are being used to update the solution

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left[b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right], \quad i = 1, \ldots, n$$

i.e., P=D-E , N=F

The related over-relaxation iteration (SOR) is

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}} \left[b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right] + (1-\omega) x_i^{(k)}$$

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \left(\frac{1}{\omega} \mathrm{D} - \mathrm{E} \right)^{-1} \mathbf{r}^{(k)}$$

Remarks:

- GS is monotonically convergent if A is symmetric and positive definite
- GS converges also for the same criteria as Jacobi
- GS is not parallelizable
- GS has less memory requirements than Jacobi, since the current iteration can overwrite elements of the previous approximation

Preconditioners

The spectral radius of the iteration matrix B is important for the convergence of the iterative solver. Using the expressions above, the original problem is (obviously) equivalent to solving

$$\mathrm{P}^{-1}\mathrm{A}\mathbf{x} = \mathrm{P}^{-1}\mathbf{b}$$

This is called a preconditioned system, where P is the preconditioning matrix or left preconditioner. Right and centered preconditioners can be introduced as well:

$$\mathrm{A}\mathrm{P}^{-1}\mathbf{y} = \mathbf{b}, \quad \mathbf{y} = \mathrm{P}\mathbf{x}$$

$$\mathrm{P}_L^{-1}\mathrm{A}\mathrm{P}_R^{-1}\mathbf{y} = \mathrm{P}_L^{-1}\mathbf{b}, \quad \mathbf{y} = \mathrm{P}_R\mathbf{x}$$

Since the preconditioner acts on the spectral radius of the iteration matrix, it would be useful to pick up, for a given linear system, an optimal preconditioner, i.e., a preconditioner which is able to make the number of iterations required for convergence independent of the size of the system.
Notice that the choice P=A is optimal but, trivially, “inefficient”.

Note: A diagonal preconditioner is generally effective if A is symmetric positive definite

Stationary vs. non-stationary methods

Define the iteration matrix $\mathrm{R_P} = \mathrm{I} - \mathrm{P}^{-1}\mathrm{A}$

With a relaxation (or acceleration) parameter α we get the following *stationary Richardson method*

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha \mathrm{P}^{-1}\mathbf{r}^{(k)}, \qquad k \geq 0$$

If we allow α to be dependent on the iteration index, we get the *nonstationary Richardson method* or *semi-iterative method*

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathrm{P}^{-1}\mathbf{r}^{(k)}, \qquad k \geq 0$$

With iteration matrix at step k: $\mathrm{R}(\alpha_k) = \mathrm{I} - \alpha_k \mathrm{P}^{-1}\mathrm{A}$

Jacobi and GS are stationary Richardson methods with α=1

For practical applications this is rewritten: $\mathbf{z}^{(k)} = \mathrm{P}^{-1}\mathbf{r}^{(k)}$ (the so-called preconditioned residual) → $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{z}^{(k)}$ and $\mathbf{r}^{(k+1)} = \mathbf{b} - \mathrm{A}\mathbf{x}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k \mathrm{A}\mathbf{z}^{(k)}$.

A nonstationary Richardson method requires the following operations:

- solve the linear system $\mathrm{P}\mathbf{z}^{(k)} = \mathbf{r}^{(k)}$
- compute the acceleration parameter α_k
- update the solution $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{z}^{(k)}$
- update the residual $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k \mathrm{A}\mathbf{z}^{(k)}$

Gradient Method

For symmetric positive definite matrices, any optimal acceleration parameter can be dynamically computed at each step k.
First, note that solving system A**x**=**b** is equivalent to finding the minimizer $x \in \mathbb{R}^n$ of the quadratic form

$$\Phi(\mathbf{y}) = \frac{1}{2}\mathbf{y}^T \mathrm{A}\mathbf{y} - \mathbf{y}^T\mathbf{b}$$

This is also called the *energy of system* A**x**=**b**, calculating the gradient gives:

$$\nabla\Phi(\mathbf{y}) = \frac{1}{2}(\mathrm{A}^T + \mathrm{A})\mathbf{y} - \mathbf{b} = \mathrm{A}\mathbf{y} - \mathbf{b} \quad \text{i.e.} \quad \nabla\Phi(\mathbf{x}) = \mathbf{0}$$

Problem:

- determine the minimizer **x** of Φ starting from a point $\mathbf{x}^{(0)} \in \mathbb{R}^n$ and,
- select suitable directions $\mathbf{d}^{(k)}$ along which gets us as close as possible to the solution **x**.

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{d}^{(k)}$$

Where α_k is the length along the step $\mathbf{d}^{(k)}$.
The most natural idea: take the descent direction of maximum slope $\nabla\Phi(x^{(k)})$, which yields the *gradient method* or *steepest descent method*:

$$\nabla\Phi(\mathbf{x}^{(k)}) = \mathrm{A}\mathbf{x}^{(k)} - \mathbf{b} = -\mathbf{r}^{(k)} \quad \rightarrow \mathbf{d}^{(k)} = \mathbf{r}^{(k)}$$

...

To compute the parameter α_k let us write explicitly $\Phi(x^{(k+1)})$ as a function of parameter α:

$$\Phi(\mathbf{x}^{(k+1)}) = \frac{1}{2}(\mathbf{x}^{(k)} + \alpha\mathbf{r}^{(k)})^T \mathrm{A}(\mathbf{x}^{(k)} + \alpha\mathbf{r}^{(k)}) - (\mathbf{x}^{(k)} + \alpha\mathbf{r}^{(k)})^T\mathbf{b}$$

Differentiating with respect to α and setting it equal to zero, yields

$$\alpha_k = \frac{{\mathbf{r}^{(k)}}^T\mathbf{r}^{(k)}}{{\mathbf{r}^{(k)}}^T\mathrm{A}\mathbf{r}^{(k)}}$$

This non-stationary Richardson method is called *gradient method with dynamic parameter* or just *gradient method*:

$$\mathbf{r}^{(k)} = \mathbf{b} - \mathrm{A}\mathbf{x}^{(k)}$$

$$\alpha_k = \frac{{\mathbf{r}^{(k)}}^T\mathbf{r}^{(k)}}{{\mathbf{r}^{(k)}}^T\mathrm{A}\mathbf{r}^{(k)}}$$

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k\mathbf{r}^{(k)}$$

For a symmetric, positive definite matrix the gradient method is convergent for any choice of $\mathbf{x}^{(0)}$

Conjugate gradient method

We can calculate the local minimum for Φ along any direction $\mathbf{p}^{(k)}$ and find α_k as the value minimizing $\Phi(\mathbf{x}^{(k)}+\alpha\,\mathbf{p}^{(k)})$, yielding

$$\alpha_k = \frac{\mathbf{p}^{(k)^T}\mathbf{r}^{(k)}}{\mathbf{p}^{(k)^T}\mathrm{A}\mathbf{p}^{(k)}}$$

Instead of just using the gradient of Φ as direction (i.e. the residual), we now use the definition of an optimal direction $\mathbf{x}^{(k)}$ with respect to a direction $\mathbf{p}\neq 0$

$$\Phi(\mathbf{x}^{(k)}) \leq \Phi(\mathbf{x}^{(k)} + \lambda\mathbf{p}), \qquad \forall\lambda \in \mathbb{R}$$

From this it follows that $\mathbf{p}$ must be orthogonal to $\mathbf{r}^{(k)}$, since $\frac{\partial\Phi}{\partial\lambda}(\mathbf{x}^{(k)})_{|\lambda=0} = 0 \quad \text{iff} \quad \mathbf{p}^T(\mathbf{r}^{(k)}) = 0$

For an iteration $\mathbf{x}^{(k+1)}=\mathbf{x}^{(k)}+\mathbf{q}$ to preserve this optimality we need also $\mathbf{p}^T\mathrm{A}\mathbf{q} = 0$

Which means the descent directions must be mutually *A-orthogonal* or *A-conjugate*

...

These conjugate directions can be constructed, yielding finally the iteration:

$$\alpha_k = \frac{\mathbf{p}^{(k)^T}\mathbf{r}^{(k)}}{\mathbf{p}^{(k)^T}\mathrm{A}\mathbf{p}^{(k)}}$$

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)}$$

$$\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k \mathrm{A}\mathbf{p}^{(k)}$$

$$\beta_k = \frac{(\mathrm{A}\mathbf{p}^{(k)})^T\mathbf{r}^{(k+1)}}{(\mathrm{A}\mathbf{p}^{(k)})^T\mathbf{p}^{(k)}}$$

$$\mathbf{p}^{(k+1)} = \mathbf{r}^{(k+1)} - \beta_k \mathbf{p}^{(k)}$$

This is the CG method, with $\mathbf{r}^{(0)} = \mathbf{b} - \mathrm{A}\mathbf{x}^{(0)}$ and $\mathbf{p}^{(0)} = \mathbf{r}^{(0)}$

For a symmetric and positive definite matrix, any method which employs conjugate directions to solve A**x**=**b** terminates after at most n steps, yielding the exact solution.

Krylov Subspace Iterations

No covered here, but sometimes useful.

These methods requires saving the vectors of the Krylov subspace of order m:

$$K_m(\mathrm{A};\mathbf{v}) = \mathrm{span}\left\{\mathbf{v}, \mathrm{A}\mathbf{v}, \ldots, \mathrm{A}^{m-1}\mathbf{v}\right\}$$

And solving an iteration

$$\mathbf{x}^{(k)} = \mathbf{x}^{(0)} + q_{k-1}(\mathrm{A})\mathbf{r}^{(0)}$$

And q_{k-1} being a appropriately chosen polynomial.

Next lecture:

- Numerical integration
- Root finding