LECTURE II: Solving linear Equations

LU-Decomposition

We start with the linear system:

$$\mathbf{Ax} = \mathbf{b} \tag{1}$$

Possibly the first method that one learns for solving such a linear system of equations is the Gaussian elimination. In computer algebra a slightly modified version of this method is the LU-decomposion, in which one tries to decompose the matrix $\mathbf{A}$ according to $\mathbf{A} = \mathbf{L} \cdot \mathbf{U}$. Knowing the LU decomposition for a matrix A allows us to solve the linear system very easily:

$$\begin{aligned} \mathbf{Ax} &= \mathbf{b} \\ \mathbf{LUx} &= \mathbf{b} \\ \mathbf{Ux} &= \mathbf{L^{-1}b} \\ \mathbf{x} &= \mathbf{U^{-1}}(\mathbf{L^{-1}b}), \end{aligned}$$

where $\mathbf{L^{-1}b}$ uses forward substitution and $\mathbf{U^{-1}}(\mathbf{L^{-1}b})$ backward substitution. Note that sometimes an additional step 'Pivoting', is needed in which either only rows (partial pivoting) or rows and columns (full pivoting) is required, e.g., if you would get zeros on the diagonal.

The question arises of how to obtain the LU decomposition? One way uses the recursive leading-row column LU algorithm.

$$\begin{pmatrix} a_{11} & \mathbf{a_{12}} \\ \mathbf{a_{21}} & \mathbf{A_{22}} \end{pmatrix} = \begin{pmatrix} 1 & \mathbf{0} \\ \mathbf{l_{21}} & \mathbf{L_{22}} \end{pmatrix} \begin{pmatrix} u_{11} & \mathbf{u_{12}} \\ \mathbf{0} & \mathbf{U_{22}} \end{pmatrix}$$

Here it is worth pointing out, if $\mathbf{A}$ is an $n \times n$-matrix, then $\mathbf{A_{22}}$ is a $(n-1) \times (n-1)$-matrix and $\mathbf{a_{12}}$, ... are vectors of length $(n-1)$. The matrix equation can also be rewritten as:

$$\begin{aligned} a_{11} &= u_{11} \\ \mathbf{u_{12}} &= \mathbf{a_{12}} \\ \mathbf{l_{21}} &= \frac{1}{u_{11}} \mathbf{a_{21}} \\ \mathbf{L_{22}U_{22}} &= \mathbf{A_{22}} - \mathbf{a_{21}}/(a_{11})^{-1}\mathbf{a_{12}} \end{aligned}$$

The $(n-1) \times (n-1)$ matrix $\mathbf{A_{22}} - \mathbf{a_{21}}/(a_{11})^{-1}\mathbf{a_{12}}$ is the Schur complement and defines a new system of size $(n-1) \times (n-1)$ to solve.

Overall, the LU-decomposition has costs proportional to n^3. Therefore, this method can only be used for 'small' matrices.

Iterative Solvers

This subsection is based on the book 'Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods' by Berrett et al.
Iterative methods use successive approximations to obtain more accurate solutions to a

linear system

$$x^{(k)} = Bx^{(k-1)} + c$$

at each step. Overall, one can distinguish 'Stationary methods' (e.g., Jacobi method, Gauss-Seidel method or Successive Overrelaxation) in which neither B nor c depend on the iteration k and 'Nonstationary methods' (e.g.,Conjugate Gradient, Minimal Residual, Generalized Minimal Residual, Biconjugate Gradient Stabilized).
The rate at which an iterative method converges depends greatly on the spectrum of the coefficient matrix. Therefore, '**preconditioner**' are employed to transforms the coefficient matrix into one with a more favorable spectrum. A good preconditioner improves the convergence of the iterative method, but sometimes is even required to ensure that the iterative method does not fail to converge. Example:

$$AP^{-1}(Px) = b \tag{2}$$

so that you solve $AP^{-}1y = b$ and $Px = y$ (right preconditioned system or alternatively:

$$P^{-1}(Ax - b) = 0. \tag{3}$$

One could even use:

$$P_1 A P_2^{-1}(P_2 x) = P_1 b. \tag{4}$$

The Jacobi Method

Considering a linear system consisting of n equations

$$Ax = b,$$

with the i-th equation given by

$$x_i = (b_i - \sum_{j \neq i} a_{i,j} x_j)/a_{i,i}$$

Then the simplest way to solve for x_i and keeping all other equations fixed, i.e., we consider all equations as being independent from each other:

$$x_i^{(k)} = (b_i - \sum_{j \neq i} a_{i,j} x_j^{(k-1)}/a_{i,i}.$$

In matrix terms this can be rewritten as:

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

with D being the diagonal, U the strictly upper, L the strictly lower part of A.

```
#include<stdio.h>
#include<math.h>

/* We are solving
   3x + 20y - z = -18
   2x - 3y + 20z = 25
   20x + y - 2z = 17
*/
/* Bring system in diagonally dominant form:
   20x + y - 2z = 17
```

```
   3x + 20y −z = −18
   2x − 3y + 20z = 25
*/
/* Equations:
   x = (17.−y+2z)/20.
   y = (−18.−3x+z)/20.
   z = (25.−2x+3y)/20.
*/
/* Defining function */
#define f1(x,y,z)  (17.−y+2.*z)/20.
#define f2(x,y,z)  (−18.−3.*x+z)/20.
#define f3(x,y,z)  (25.−2.*x+3.*y)/20.

/* Main function */
int main()
{
 float x0=0, y0=0, z0=0, x1, y1, z1, e1, e2, e3;
 int count=1;
 float e = 1e−6;

 printf("\nCount\tx\ty\tz\n");
 do
 {
  /* Calculation */
  x1 = f1(x0,y0,z0);
  y1 = f2(x0,y0,z0);
  z1 = f3(x0,y0,z0);
  printf("%d\t%0.7f\t%0.7f\t%0.7f\n",count, x1,y1,z1);

  /* Error */
  e1 = fabs(x0−x1);
  e2 = fabs(y0−y1);
  e3 = fabs(z0−z1);

  count++;

  /* Set value for next iteration */
  x0 = x1;
  y0 = y1;
  z0 = z1;
 }while(e1>e && e2>e && e3>e);

 printf("\nSolution: x=%7.6f, y=%7.6f and z = %7.6f\n",x1,y1,z1);


 return 0;
}
```

Gauss-Seidel Method

A simple extension of the Jacobi method is the Gauss-Seidel method. Now we assume that the equations are examined one at a time in sequence and we will use previously computed results. This results into

$$x_i^{(k)} = (b_i - \sum_{j<i} a_{i,j} x_j^{(k)} - \sum_{j>i} a_{i,j} x_j^{(k-1)})/a_{i,i}.$$

or in matrix form:

$$x^{(k)} = (D + L)^{-1}(-Ux^{(k-1)} + b)$$

Successive Overrelaxation (SOR)

The Successive Overrelaxation Method, or SOR, is devised by applying extrapolation to the Gauss-Seidel method. This extrapolation takes the form of a weighted average between the previous iterate and the computed Gauss-Seidel iterate successively for each component:

$$x_i^{(k)} = \omega \bar{x}_i^{(k)} + (1 - \omega)x_i^{(k-1)}$$

or

$$x^{(k)} = (D + \omega L)^{-1}(-\omega U + (1 - \omega)D)x^{(k-1)} + \omega(D + \omega L)^{-1}b$$

Usually, we have to pick $\omega \in (0, 2)$, for $\omega = 1$ we obtain the normal Gauss-Seidel method, for $\omega < 1$ we are using an underrelaxation and for $\omega > 1$ this method revers to overrelaxation.

Conjugate Gradient Method (CG)

Nonstationary methods differ from stationary methods in that the computations involve information that changes at each iteration.
The Conjugate Gradient method can be used for symmetric positive definite systems. (Note: positive definite: An $n \times n$ symmetric real matrix M is positive-definite if $x^T \cdot M \cdot x > 0$ for all non-zero x in $\mathbb{R}^n$.)
CG generates vector sequences of iterates (i.e., successive approximations to the solution), residuals corresponding to the iterates, and searches for directions used in updating the iterates and residuals.
The fundamental idea is to rewrite our system of equations as a minimization problem, i.e., $A \cdot x = b$ is rewritten as

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

The gradient of $E(x)|_{x_k}$ at position x_k is then given by $A \cdot x_k - b = -r^k$. Instead of minimizing along the residuum r_k (which would be the gradient method), one computes another direction p_k along which one minimizes $E(x)$, i.e.,

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

The corresponding residuals $r^{(k)} = b - Ax^{(k)}$ are updated following

$$r^{(k)} = r^{(k-1)} - \alpha_k A p^{(k)}.$$

The parameter α_k is given by:

$$\alpha_k = \frac{{r^{(k-1)}}^T r^{(k-1)}}{{p^{(k)}}^T A p^{(k)}}$$

and minimizes ${r^{(k)}}^T A^{(-1)} r^{(k)}$. The search direction is then updated along

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

with

$$\beta_{(k)} = \frac{{r^{(k)}}^T r^{(k)}}{{r^{(k-1)}}^T r^{(k-1)}}.$$

This choice of β ensures that $r^{(k)}$ and $r^{(k-1)}$ are orthogonal (and also orthogonal to all previous choices).

In summary, the iteration procedure can be summarized by:

1. Compute x
2. Compute r
3. Compute β
4. Compute p
5. Compute α
6. Go back to (1)

In the following, we want to consider one example, to get a better understanding of the method. Consider

$$\mathbf{Ax} = \begin{pmatrix} 4 & 1 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

We start with the initial guess

$$\mathbf{x_0} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}.$$

We start by computing the residual vector:

$$\mathbf{r_0} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} - \begin{pmatrix} 4 & 1 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \end{pmatrix} = \begin{pmatrix} -8 \\ -3 \end{pmatrix} = \mathbf{p_0}$$

Now, we have to compute α:

$$\alpha_0 = \frac{{r^{(0)}}^T r^{(0)}}{{p^{(0)}}^T A p^{(0)}} = \frac{(-8,3) \cdot \begin{pmatrix} -8 \\ -3 \end{pmatrix}}{(-8,3) \begin{pmatrix} 4 & 1 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} -8 \\ -3 \end{pmatrix}} = \frac{73}{331}$$

This gives us our next solution:

$$\mathbf{x_1} = \mathbf{x_0} + \alpha_0 \mathbf{p_0} = \begin{pmatrix} 2 \\ 1 \end{pmatrix} + \frac{73}{331} \begin{pmatrix} -8 \\ -3 \end{pmatrix} = \begin{pmatrix} 0.2356 \\ 0.3384 \end{pmatrix}.$$

We now need to move to the second iteration:

$$\mathbf{r_1} = \mathbf{r_0} - \alpha_0 \mathbf{A p_0} = \begin{pmatrix} -8 \\ -3 \end{pmatrix} - \frac{73}{331} \begin{pmatrix} 4 & 1 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} -8 \\ -3 \end{pmatrix} = \begin{pmatrix} -0.2810 \\ 0.7492 \end{pmatrix}$$

Then, we get for β:

$$\beta_{(1)} = \frac{{r^{(1)}}^T r^{(1)}}{{r^{(0)}}^T r^{(0)}} = \frac{(-0.2810, 0.7492) \cdot (-0.2810, 0.7492)^T}{(-8,-3) \cdot (-8,-3)^T} = 0.0088.$$

With β we can compute

$$\mathbf{p_1} = \mathbf{r_1} + \beta_1 \mathbf{p_0} = \begin{pmatrix} -0.3511 \\ 0.7229 \end{pmatrix},$$

plugging this into the equation for α leads to

$$\alpha_1 = \frac{{r^{(1)}}^T r^{(1)}}{{p^{(1)}}^T A p^{(1)}} = 0.4122$$

This leads to the final solution:

$$\mathbf{x_2} = \mathbf{x_1} + \alpha_1 \mathbf{p_1} = \begin{pmatrix} 0.0909 \\ 0.6346 \end{pmatrix}$$

This solution is up to round up errors exact. In fact, it is possible to show that for exact arithmetic, the method converges to the correct solution within m steps where m determines the size of the $m \times m$ matrix A.

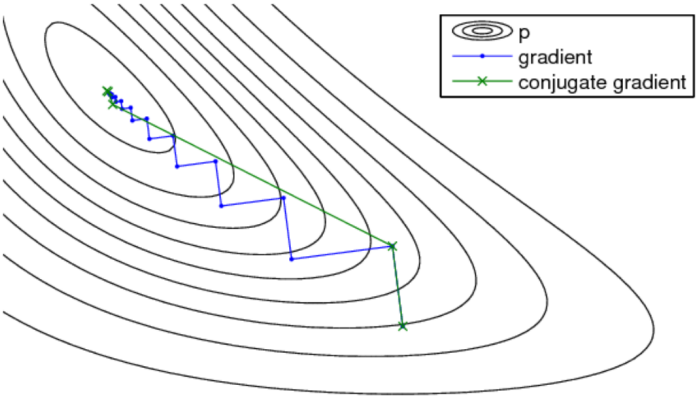

Figure 1: Sketch of the conjugate gradient method (plot taken from 'A gradient-based algorithm competitive with variational Bayesian EM for mixture of Gaussians' by Kuusela et al.)

Below you also find a python code, that actually uses the CG and also Gradient method to solve our problem from above:

```
 import numpy as np
import matplotlib.pyplot as plt

##setup plot
fig,ax=plt.subplots(1,1,figsize=(12,12))

##define the problem
A = np.array([[4.,1.],[1.,3.]])
b = np.array([1.,2.])

#create grid for contour plot
xp = np.arange(-2.5, 2.5, 0.01)
yp = np.arange(-2.5, 2.5, 0.01)
X, Y = np.meshgrid(xp, yp)
##Setup quadratic E for minimization
R = 0.5*(A[0][0]*X+A[0][1]*Y)*X+ 0.5*(A[1][0]*X+A[1][1]*Y)*Y - b[0]*X - b
    [1]*Y
plt.contourf(X,Y,R,50)
```

```
18
19
20 #initial guess
21 x    = np.array([2.,1.])
22 beta = 0
23 p    = np.array([0, 0])
24 r    = np.array([1, 1])
25
26 #plot initial guess
27 plt.scatter(x[0],x[1],color='red')
28
29
30 ### CG
31 #perform 2 steps, since we know that we are then at the final location
32 for i in range(1,3):
33   r0   = r
34   r    = b - np.dot(A,x)
35   if i>1:
36       beta = np.dot(r.T,r)/np.dot(r0.T,r0)
37   p    = r + beta*p
38   a    = np.dot(r.T,r)/np.dot(np.dot(p.T,A),p)
39   #storing it shortly in xnew for plotting
40   xnew = x + a*p
41   plt.scatter(xnew[0],xnew[1],color='red')
42   plt.arrow(x[0], x[1], xnew[0]-x[0], xnew[1]-x[1], color='red')
43   x  = xnew
44
45 ### Gradient method
46 #reset initial guess
47 x    = np.array([2.,1.])
48
49 for i in range(1,10):
50   r    = b - np.dot(A,x)
51   a    = np.dot(r.T,r)/np.dot(np.dot(r.T,A),r)
52   xnew = x + a*r
53   plt.scatter(xnew[0],xnew[1],color='blue',alpha=0.5)
54   plt.arrow(x[0], x[1], xnew[0]-x[0], xnew[1]-x[1], color='blue',alpha=0.5)
55   x  = xnew
56
57 plt.show()
```

Generalized Minimal Residuals

In the following, we will only sketch the idea of the Generalized Minimal Residual (GMRES) method. This method is important since it allows to solve nonsymmetric linear systems. The method is used to generate a sequence of orthogonal vectors. As in the Conjugate Gradient method, the residuals form an orthogonal basis for the space

$$r^{(0)}, Ar^{(0)}, A^2r^{(0)}, ...$$

This basis might be linearly dependent, so that a new set of orthonormal vectors $q_1, q_2, ...$ are constructed. The basis construction is based on the Arnoldi method (a modified Gram-Schmidt orthogonalization procedure). The vectors q_i are stored in an $m \times n$ matrix, Q_n.

The final GMRES iterates are then constructed as

$$x^{(i)} = x^{(0)} + y_1 q^{(1)} + ...y_i q^{(i)} = x_0 + \mathbf{Q_n} \cdot \mathbf{y}.$$

The coefficients y_k are chosen to minimize the residual norm $||b - Ax^{(i)}||$. It is worth pointing out that for the computation of the residual it is not necessary to compute x at every iteration step:

$$||r_n|| = ||b - Ax_n|| = ||b - A(x_0 + Q_n y_n)|| = ||r_0 - AQ_n y_n|| =$$
$$||\beta q_1 - Q_{n+1}\tilde{H}_n y_n|| = ||Q_{n+1}(\beta e_1 - \tilde{H} y_n)|| = ||\beta e_1 - \tilde{H}_n y_n||,$$

with e_1 being the first vector in the standard basis of $\mathbb{R}^{n+1}$ and $\beta = ||r_0||$. In the equation above, we have also introduced the upper Hessenberg matrix $\tilde{H}_n$ (which is an $(n+1) \times n$ matrix) given by $AQ_n = Q_{n+1}\tilde{H}_n$.
The can be summarized according to:

- calculate q_n using the Arnoldi method
- find q_n that minimizes $||r_n||$
- repeat until residual is small enough
- compute x_n