LECTURE II: Solving linear Equations

This lecture is about solving linear systems of equations,

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

where A is a $n \times n$ matrix, and x, b are n-dimensional vectors.

LU-Decomposition

Possibly the first method that one learns for solving Eq. (1) 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 A according to $A = L\,U$ with L being a lower-triangular matrix and U and upper triangular matrix. Knowing the LU decomposition for a matrix A allows us to solve the linear system very easily:

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

where $L^{-1}b$ is computed using forward substitution, and $U^{-1}(L^{-1}b)$ by backward substitution. Note that sometimes an additional 'Pivoting step', is needed in which either rows (partial pivoting) or rows and columns (full pivoting) are reordered, 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, where we separate out the first row/column of A, U, L:

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

Here it is worth pointing out, that A_{22} is a $(n-1) \times (n-1)$-matrix and a_{12}, a_{21} ... are vectors of length $(n-1)$, either in column-form (subscript 21) or row-form (subscript 12). The elements a_{11} and u_{11} are real numbers.
Multiplying out, this matrix equation can be rewritten as:

$$\begin{aligned} a_{11} &= u_{11} \\ u_{12} &= a_{12} \\ l_{21} &= \frac{1}{a_{11}} a_{21} \\ L_{22}U_{22} &= A_{22} - \frac{1}{a_{11}} a_{21} \otimes a_{12} \end{aligned}$$

The $(n-1) \times (n-1)$ matrix $A_{22} - \frac{1}{a_{11}} a_{21} \otimes a_{12}$ 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, $n \lesssim 10,000$.

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).

Preconditioning: The rate at which an iterative method converges depends greatly on the spectrum of the coefficient matrix. A '**preconditioning**' P is employed to transform 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. One multiples Eq. (1) by P,

$$PAx = Px, \tag{2}$$

to obtain a new linear system with the coefficient matrix being the product (PA). The trick is now to find a preconditioning matrix, such that (i) it is approximately the inverse of A and (ii) it is easy to calculate. There is an immense amount of knowledge and tools around preconditioning, although we don't have time to go into them.

The Jacobi Method

Considering a linear system consisting of n equations

$$Ax = b,$$

the i-th equation can be written as

$$x_i = \frac{1}{a_{ii}}\Big(b_i - \sum_{j \neq i} a_{i,j} x_j\Big).$$

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:

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

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>

```

```c
/* 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 in

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

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 $x^{(k)}$ and the Gauss-Seidel iterate $\tilde{x}_i^{(k-1)}$, successively for each component:

$$x_i^{(k)} = \omega\tilde{x}_i^{(k)} + (1-\omega)x_i^{(k-1)}, \qquad i = 1, \ldots, n.$$

In matrix-form, this gives

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

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 gives 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 $x^T Mx > 0$ 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., Eq. (1) is rewritten as

$$E(x) := \frac{1}{2}x^T Ax - b^T x \quad \rightarrow \quad \text{min}$$

The gradient of $E(x)$ is grad$E = Ax - b$, so that the minimum of E corresponds to the solution of Eq. (1).
We denote the residual of the k-th step as $r^{(k)} := b - Ax^{(k)}$, and in CG, one determines the next step along a search direction $p^{(k)}$, i.e.

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

with α_k chosen to yield the minimum of $E(x^{(k)})$ along this line, which is given by

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

This α_k also minimizes ${r^{(k)}}^T A^{(-1)} r^{(k)}$.
The trick in Conjugate Gradients is the choice of $p^{(k)}$. Just going as steeply downhill as possible (i.e. $p^{(k)} = -gradE(x^{(k-1)}) = r^{(k-1)}$) is not optimal. This "gradient method" is prone to going 'zig-zag', needing many steps.

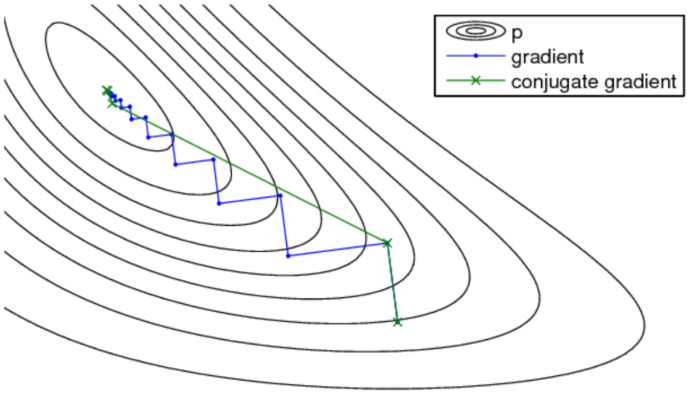

Figure 1: *Sketch of the gradient and conjugate gradient methods (plot from 'A gradient-based algorithm competitive with variational Bayesian EM for mixture of Gaussians' by Kuusela et al.)*

Instead one chooses the search direction by

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

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). This orthogonality, in turn prevents the 'zig-zag' of the gradient method, and leads to faster convergence.

In summary, the iteration procedure can be summarized by:

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

In infinite precision, CG will arrive at the exact solution in at most n steps. In practical application, CG typically needs $N_{\text{its}} \ll n$. The number of iterations needed to reduce the

residual by a certain factor depends on the conditioning number of the matrix A.

$$N_{\text{its}} \propto \sqrt{\frac{\lambda_{\max}}{\lambda_{\min}}}, \tag{3}$$

where $\lambda_{\max/\min}$ is the largest and smallest eigenvalue of A. (Preconditioning precisely aims to bring the preconditioned eigenvalues close to unity)
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.
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,(ax1,ax2)=plt.subplots(1,2,figsize=(10,5))

##define the problem
A = np.array([[4.,1.],[1.,3.]]) # change 3->23 for bad 'gradient'
    convergence
b = np.array([1.,2.])

#create grid for contour plot
xp = np.arange(-.6, 2.4, 0.01)
yp = np.arange(-1, 2, 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
ax1.contour(X,Y,R,25)


#initial guess
x    = np.array([2.,1.])
beta = 0
p    = np.array([0, 0])
r    = np.array([1, 1])

#plot initial guess
ax1.scatter(x[0],x[1],color='red')


### CG
#perform 2 steps, since we know that we are then at the final location
CGresiduals=[]
for i in range(1,3):
  r0   = r
  r    = b - np.dot(A,x)
  if i>1:
      beta = np.dot(r.T,r)/np.dot(r0.T,r0)
  p    = r + beta*p
  a    = np.dot(r.T,r)/np.dot(np.dot(p.T,A),p)
  #storing it shortly in xnew for plotting
  xnew = x + a*p
  ax1.scatter(xnew[0],xnew[1],color='red')
  ax1.arrow(x[0], x[1], xnew[0]-x[0], xnew[1]-x[1], color='red')
  x  = xnew
```

```
  CGresiduals.append(np.dot(r.T,r)**0.5)

# append last residual
r     = b - np.dot(A,x)
CGresiduals.append(np.dot(r.T,r)**0.5)

### Gradient method
#reset initial guess
x     = np.array([2.,1.])

GradientResiduals=[]
for i in range(1,20):
  r     = b - np.dot(A,x)
  a     = np.dot(r.T,r)/np.dot(np.dot(r.T,A),r)
  xnew = x + a*r  # step in direction of 'r'

  ax1.scatter(xnew[0],xnew[1],color='blue',alpha=0.5)
  ax1.arrow(x[0], x[1], xnew[0]-x[0], xnew[1]-x[1], color='blue',alpha=0.5)
  x  = xnew
  GradientResiduals.append(np.dot(r.T,r)**0.5)

ax2.plot(CGresiduals, '-o', label='CG')
ax2.plot(GradientResiduals, '-+', label='gradient')
ax2.set_yscale('log');
ax2.legend();
```

Generalized Minimal Residuals (GMRES)

GMRES is important, because this iterative method also works for non-symmetric A and/or for A without positivity properties. This generality comes at a higher cost (both computations and storage) than for conjugate gradients. In the following, we will only sketch the ideas behind this method.
At iteration K, define the Krylov-subspace

$$K_k := \text{span}\left(r^{(0)}, Ar^{(0)}, A^2r^{(0)}, \dots A^{k-1}r^{(0)}\right) \tag{4}$$

At each iteration k, GMRES will determine $x^{(k)} = x^{(0)} + c^{(k)}$, with $c^{(k)} \in K_k$, such that the residal $||b - Ax^{(k)}||$ is minimized over all possible $c^{(k)}$.
To do this, GMRES needs to build and store a orthonormal basis $\{v^{(k)}\}$ of K_k, and needs to evaluate $x^{(k)} = x^{(0)} + y_1v^{(1)} + y_2v^{(2)} + \dots y_kv^{(k)}$ for suitable coefficients $y_1, \dots y_k$.
It turns out the residual r^k can be computed from known data without forming $x^{(k)}$ explicitly, and therefore, the method can be summarized as follows:

- calculate the new orthonormal basis-vector v_k using the Arnoldi method
- find y_k that minimizes $||r^{(k)}||$
- Calculate residual $r^{(k)}$
- repeat until residual is small enough
- compute final $x^{(k)}$

The storage space for the orthonormal basis and the sums over the orthogonal basis, entails a cost that grows linearly with k. To mitigate this growing cost, it is typical to perform 'm' iterations, and then to re-start the entire algorithm with the then best solution.

This is called 'GMRES(m)'.

GMRES and Conjugate Gradients are actually simular in that they construct their approximate solutions using the Krylov subspace K_k, and in that residuals obey orthogonality conditions.
Conjugate Gradients takes advantage of the extra knowledge that A must be symmetric and positive-definite to express the scheme without the need to retain information from all previous iterations.