Lab 1: Iterative Methods for Solving Linear Systems

January 22, 2017

Introduction

Many real world applications require the solution to very large and sparse linear systems where direct methods such as Gaussian elimination are prohibitively expensive both in terms of computational cost and in available memory. In this Lab, you will learn how to implement the Jacobi, Gauss-Seidel, Conjugate Gradient (CG) (and optionally Preconditioned Conjugate Gradient (PCG) and the General Minimal RESidual (GMRES)) algorithms to solve linear systems and investigate the associated convergence properties of these algorithms.

1 Problem 1

1.1 Problem statement

Consider the following advection-diffusion equation in 1D with the following boundary conditions.

$$-\epsilon u'' + \beta u' = 0 \qquad u(0) = 0, \quad u(1) = 1 \tag{1}$$

where ϵ is the diffusion coefficient and β is the velocity at which the quantity is flowing. Equation (1) can be discretized and written in matrix form as,

$$Ax = b$$

We have provided a function that discretizes these equations and returns the matrices for you. An example call to this function is shown below.

```
import numpy as np
N = 60
beta = 1; epsilon = 1e-2;
A,b = generate_matrices(epsilon,beta,N)
```

Note that we have included the header "import numpy as np". Numpy will be the package we will use for numerical linear algebra in Python. N is the number of intervals we use in the discretization - we are essentially dividing the interval $[0, 1]$ into N intervals each of size $h = 1/N$. The larger N, the more accurate the solution. Fix $N = 60$ for now. To solve the linear system directly using a Python function type,

```
x = np.linalg.solve(A,b)
```

The analytic solution is given by :

$$u_{ex} = \frac{\exp\left(\frac{\beta}{\epsilon}x\right) - 1}{\exp\left(\frac{\beta}{\epsilon}\right) - 1}$$

You can compare your solution to the exact solution. To do this, and to view the accuracy of the computed solution type,

```
visualize_solution(x)
```

Exercise: Try experimenting with different values of N and observe what happens. You will see that the larger N is, the smaller the discretization error and the better the match with the exact solution.

1.2 Jacobi method

We start by implementing the Jacobi method for solving linear systems. The Jacobi algorithm and the Gauss-Seidel algorithm (that you will later implement) belong to a class of iterations to solve linear systems called stationary iterations. They are named as such because the solution of the linear system is expressed as finding the stationary point of some fixed-point iteration. The matrix A,

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

can be split into $A = D + R$ where D is its diagonal component and R is the remaining matrix after subtracting the diagonal, $R = A - D$,

$$D = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}, \quad R = \begin{pmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{pmatrix}$$

The solution to the linear system by Jacobi method is then obtained iteratively by:

$$x^{k+1} = D^{-1}(b - Rx^k) \quad \text{or} \quad x_i^{k+1} = \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij}x_j^k\right)$$

A basic implementation of the Jacobi method is given below.

```
[m,n] = np.shape(A)
x_jacobi = np.zeros(np.shape(x));
maxit = 20;
nm = np.zeros((maxit,1))

D = np.diag(np.diag(A))
R = A - D
Dinvb = np.linalg.solve(D,b)
for k in np.arange(maxit):
   x_jacobi = Dinvb - np.linalg.solve(D,np.dot(R,x_jacobi) )
   nm = np.linalg.norm(np.dot(A,x_jacobi)-b)
```

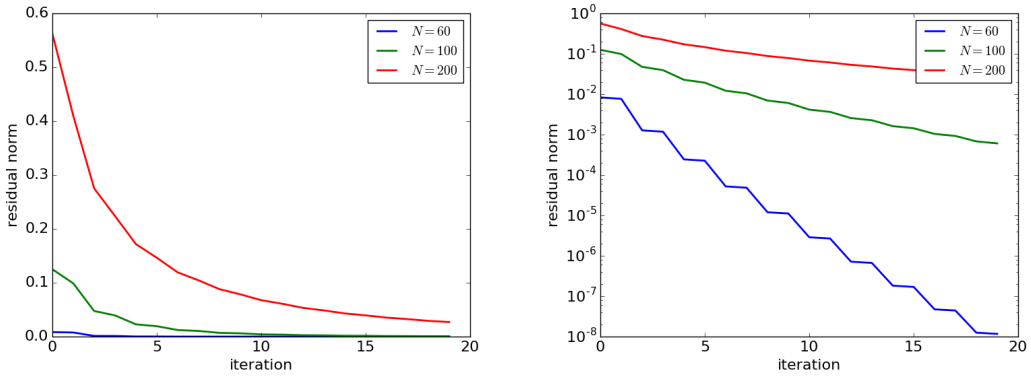

Figure 1: (left) Plot of the residual norm vs. the iteration count for several values of N. (right) Semilog plot of the residual norm vs. iteration count for several values of N.

Again, if you want to visualize the solution compared to the analytic solution, just call the visualize_solution function.

```
visualize_solution(x_jacobi)
```

To see how the method is converging, plot the norm of the residual $\|Ax-b\|_2$ as a function of the iteration number. Note that we have stored the residual norm in a vector called nm.

```
import matplotlib.pyplot as plt
plt.plot(nm)
```

In Figure 1 we plot the behavior of the norm vs. the iteration count for several values of N. The plot to the right is on a semilog plot, so it is clear to see that the convergence rate is slower for N larger. This is because the spectral radius $\rho(D^{-1}R)$ gets larger as N gets larger. Recall the definition of spectral radius,

$$\rho(A) = \max\{|\lambda_1|, |\lambda_2|, \cdots, |\lambda_n|\}$$

where λ_i denotes the i-th eigenvalue of A. We have plotted in Figure 2 how the spectral radius varies with N. The spectral radius is the factor that controls convergence. For stationary methods to converge, $\rho < 1$ and the smaller ρ is, the faster the convergence rate.

Exercise: Calculate the spectral radius for the varying values of N.

```
T = np.linalg.solve(-D,R)
eigsT, _ = np.linalg.eig(T)
rho = np.max(np.abs(eigsT))
```

Another interesting thing that can be shown (and is a nice linear algebra exercise if you are interested) is that when $N \geq \beta/2\epsilon$ the matrix is diagonally dominant, a highly desirable property and a sufficient (but not necessary) condition for the Jacobi method to converge. Recall that a matrix A is diagonally dominant if

$$|a_{ii}| \geq \sum_{j \neq i} |a_{ij}| \quad \text{for all } i$$

Also recall that $h = 1/N$ so the condition for diagonal dominance can also be written in terms of the grid resolution as $h \leq 2\epsilon/\beta$.

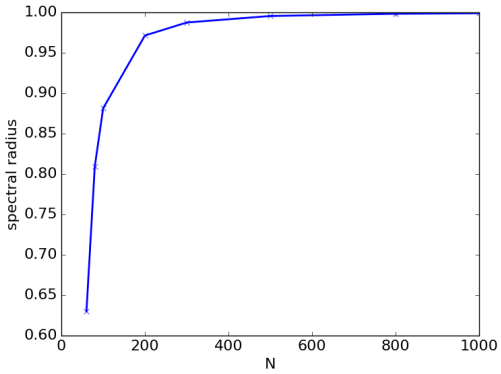

Figure 2: Spectral radius as a function of the grid size. Note that as the grid size gets larger, the spectral radius goes to 1 and convergence of Jacobi method will thus be slower.

Exercise: Run the Jacobi algorithm with an $N < \beta/2\epsilon$ and you will observe that the Jacobi method will probably not converge.

2 Problem 2

2.1 Problem statement

Let us a consider a slightly more complicated problem. We consider a 2D Poisson problem on a square $[-1, 1] \times [-1, 1]$,

$$-\nabla^2 u = f$$

with boundary conditions that vanish and with a source term,

$$f(x) = 1 \quad \text{if} \quad |x| < 0.5 \text{ and } |y| < 0.5$$

A visualization of the source field and the exact solution is shown in Figure 3. Note the difference in scales.

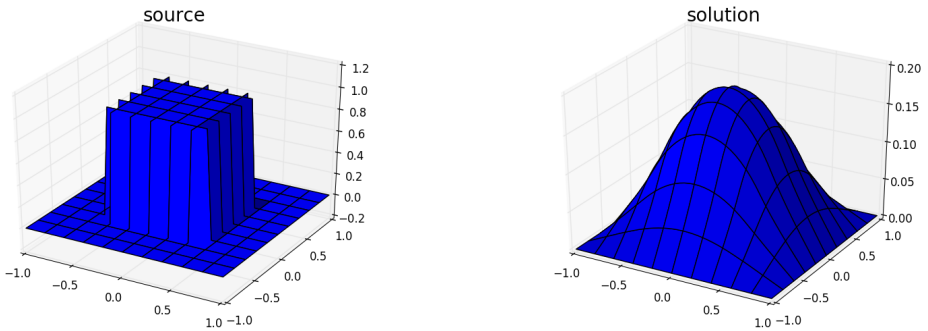

Figure 3: (left) Source term for the 2D Poisson problem. (right) The solution of the 2D Poisson problem.

```
N = 40
A, b = generate_matrices_2d(N)
```

The code above will generate a system matrix A of size $(N+1) \times (N+1)$, corresponding to the number of nodes in the grid.

2.2 Jacobi method

Exercise: Run the Jacobi algorithm you implemented in Question 1 on this problem. Set maxit= 20. You will observe that even after 20 iterations, the method is far from convergence. Again, pay close attention to the scales. Note that the convergence rate is very slow. This is because the spectral radius in this case is 0.997. To visualize the solution after 20 iterations as shown in Figure 4,call the visualize_solution_2d function.

```
visualize_solution_2d(x_jacobi)
```

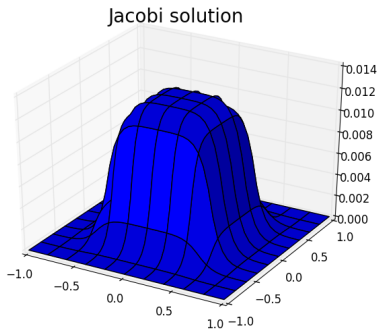

Figure 4: Solution of the 2D Poisson problem after 20 steps of the Jacobi method.

2.3 Gauss-Seidel method

The next algorithm we will implement is Gauss-Seidel. Gauss-Seidel is another example of a stationary iteration. The idea is similar to Jacobi but here, we consider a different splitting of the matrix A. The matrix A,

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

can be split into $A = L + U$ where L is the lower triangular part of A and U is the strictly upper triangular part of A (without the diagonal),

$$L = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ a_{12} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{pmatrix}, \quad U = \begin{pmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}$$

The solution to the linear system by the Gauss-Seidel algorithm is then obtained iteratively by:

$$x^{k+1} = L^{-1}(b - Ux^k) \quad \text{or} \quad x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij} x_j^k \right)$$

Note that $\rho_{gs}(L^{-1}U) < \rho_{\text{jacobi}}(D^{-1}R)$ for the class of matrices that converge with these algorithms. See the convergence plot in Figure 6.

Exercise: Program the Gauss-Seidel method and run it for 20 iterations. Hint: To extract the lower triangular part of the matrix, use np.tril(A), and visualize the solution. You should get something that matches the figure shown in Figure 5.
Exercise: Gauss-Seidel has favorable convergence properties over Jacobi, but it also has the additional advantage that it requires less storage. Why does Gauss-Seidel require less storage than Jacobi?

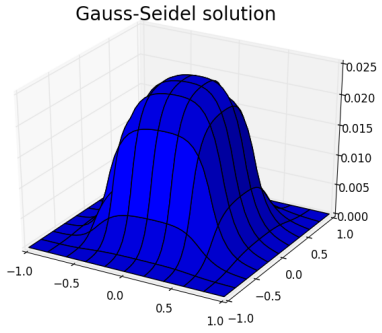

Figure 5: Solution of the 2D Poisson problem after 20 steps of the Gauss-Seidel method.

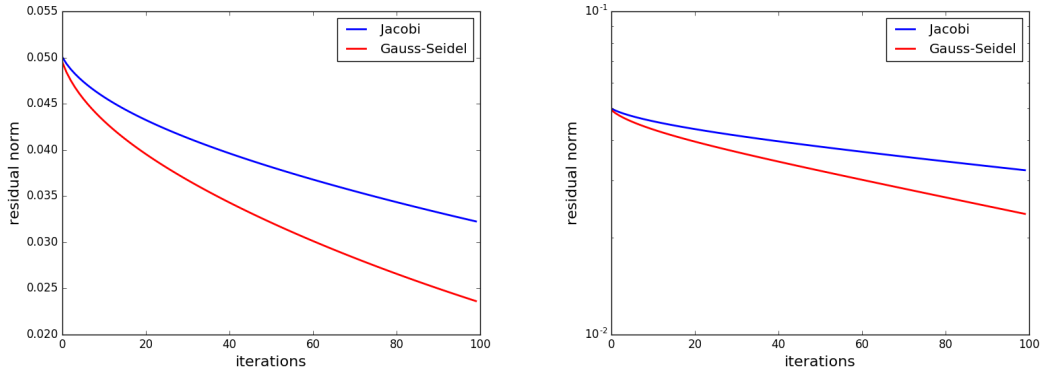

Figure 6: (left) Plot of the residual norm vs. the iteration count. (right) Semilog plot of the residual norm vs. iteration count.

Exercise: The Jacobi algorithm has recently gained popularity, even with its slow convergence. This is because it is very easy to parallelize. Why?

2.4 Conjugate Gradient

Another class of iterative algorithms that are used to solve linear systems are Krylov algorithms. Krylov subspace methods are considered as one of the ten most important classes of numerical methods (To read about its history and about the other 9 algorithms, refer to https://www.siam.org/pdf/news/637.pdf). Krylov subspace methods are based on successive multiplications of the matrix A with the right-hand-side vector b, and in fact, this allows A to be given as an operator. This eliminates the need to store every entry in the matrix. Additionally, since A is sparse, these multiplications of A with b are generally cheap.
Conjugate gradient (CG) is one of the most widely used methods for solving sparse linear systems, with A being a symmetric positive definite matrix.

```
x_cg = np.zeros(np.shape(x));
rk = b-np.dot(A,x_cg)
p = rk
for k in np.arange(maxit):
   alpha = np.dot(rk,rk)/np.dot(p,np.dot(A,p))
   x_cg = x_cg + alpha*p
   rkp1 = rk - alpha*np.dot(A,p)
   beta = np.dot(rkp1,rkp1)/np.dot(rk,rk)
   p = rkp1 + beta*p
   rk = rkp1
```

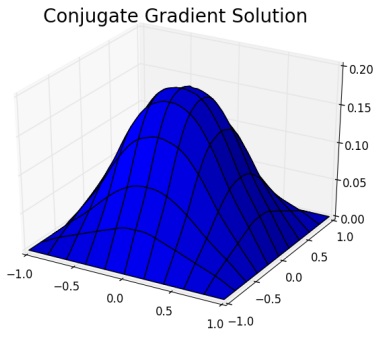

Figure 7: Solution of the 2D Poisson problem after 20 steps of the Conjugate Gradient algorithm.

In Figure 7 we can see that even with 20 iterations, the computed solution very well matches the exact solution. In fact, the convergence plot in Figure 8 shows how the residual norm behaves with conjugate gradient.
Exercise: Note that the residual norm does not monotically decrease for CG. Why? What is the norm that Conjugate Gradient is minimizing?

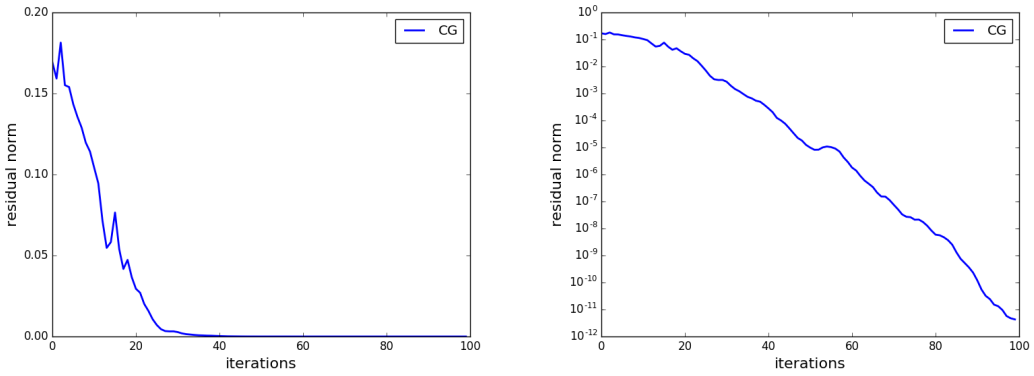

Figure 8: (left) Plot of the residual norm vs. the iteration count. (right) Semilog plot of the residual norm vs. iteration count.

2.5 Preconditioned Conjugate Gradient (PCG) (Optional)

In many cases, the linear systems are very ill-conditioned and CG may be slow to converge, or even not converge at all. In fact, the number of iterations required for the conjugate

gradient algorithm to converge is proportional to the square root of the condition number of A. Preconditioning is often necessary to ensure convergence of CG. A standard method for preconditioning is to multiply the matrix equation by a matrix M such that

$$M^{-1}Ax = M^{-1}b$$

Note that this system is equivalent to the original system. To improve convergence of CG, the preconditioner M needs to be chosen so that $M^{-1}A$ is better conditioned than A. In the case of conjugate gradient, M needs to be positive definite.
In the code below, we show the conjugate gradient code with Jacobi preconditioning ($M = \text{diag}(A)$).

```
D = np.diag(np.diag(A))
x_pcg = np.zeros(np.shape(x));
rk = b-np.dot(A,x_pcg)
zk = np.linalg.solve(D,rk)
p = zk
for k in np.arange(maxit):
   alpha = np.dot(rk,zk)/np.dot(p,np.dot(A,p))
   x_pcg = x_pcg + alpha*p
   rkp1 = rk - alpha*np.dot(A,p)
   zkp1 = np.linalg.solve(D,rkp1)
   beta = np.dot(zkp1,rkp1)/np.dot(zk,rk)
   p = zkp1 + beta*p
   rk = rkp1
   zk = zkp1
```

Consider the example where A is a matrix with entries $a_{ii} = 0.2 + \sqrt{i}$ along the diagonals, and 1 on the subdiagonal, superdiagonal, and along the 200th subdiagonal and 200th superdiagonal (i.e. $a_{ij} = 1$ when$|i - j| = 1$ and $|i - j| = 200$). We have provided a function to return this matrix for you.

```
[A,b] = pcg_example()
```

Observe the behaviour of the residual norm for CG and PCG for this matrix.

2.6 GMRES (Optional)

For non-symmetric matrices, one of the most popular iterative methods is the Generalized Minimal RESidual (or, GMRES). Similar to CG, it is a Krylov solver. Recall the definition of a Krylov subspace $\mathcal{K}_m(A, b)$,

$$\mathcal{K}_k(A, b) = \text{span}\{b, Ab, A^2, \cdots, A^{k-1}\}$$

The idea behind GMRES is that at every iteration, GMRES computes the solution estimate x_k that minimizes the residual Euclidean norm over a Krylov subspace of dimension k. To generate this subspace, the Arnoldi method is used. The Arnoldi iteration uses a modified Gram-Schmidt method to get the orthonormal vectors that span the Krylov subspace. The Arnoldi algorithm is summarized in Algorithm 1.
Note that after k steps of the Arnoldi algorithm, we have generated an upper Hessenberg matrix H and a matrix Q whose columns are the orthonormal vectors spanning the Krylov subspace. These matrices satisfy the following relation,

$$AQ_k = Q_{k+1}\bar{H}_k$$

Algorithm 1 Arnoldi using Modified Gram-Schmidt

1: Given A, b the right hand side.
2: Compute $q_1 = b/\beta$ and $\beta \stackrel{\text{def}}{=} \|b\|_2$
3: **for all** $j = 1, \ldots, k$ **do**
4: Compute $v = Aq_j$
5: **for all** $i = 1, \ldots, j$ **do**
6: $h_{ij} := q_i^* v$
7: Compute $v := v - h_{ij} q_i$
8: **end for**
9: $h_{j+1,j} := \|v\|_2$. If $h_{j+1,j} = 0$ stop
10: $v_{j+1} = v/h_{i+1,i}$
11: **end for**

Algorithm 2 GMRES

1: Given A, b the right hand side.
2: Compute $q_1 = b/\beta$ and $\beta \stackrel{\text{def}}{=} \|b\|_2$
3: **for all** $j = 1, \ldots, k$ **do**
4: Perform j steps of the Arnoldi algorithm as described in Algorithm 1
5: Find y that minimizes $\|\bar{H}_j - \beta e_1\|_2$ where e_1 is the first vector of the standard basis $\mathcal{R}^{j+1}$ (i.e. $e_1 = (1,\ 0\ , \cdots, 0)^T$) Set $x_j = Q_j y$
6: **end for**

The GMRES algorithm is then described in Algorithm 2. In step 5, you can either solve the least squares problem using QR or use the numpy function, numpy.linalg.lstsq($\bar{H}_j, \beta e_1$).
Exercise: Where in the code are the Krylov vectors computed?