Iterative Solution of Linear Systems of Equations

Direct and Iterative methods for $Ax = b$

- Direct methods like Gaussian elimination
  - returns solution after fixed but large number of operations
  - errors due to rounding and conditioning
  - based on matrix factorisations
- Iterative methods like Jacobi or Gauss-Seidel
  - returns approximation after variable number of iterations
  - each iteration evaluates matrix vector product
  - error a function of number of iterations taken
  - less affected by rounding, naturally fault tolerant

Recasting linear system $Ax = b$ as system $x = Tx - c$

- Why?
  - Solve by iteration

$$x^{(k+1)} = Tx^{(k)} + c \quad k = 0, 1, \ldots, m$$

- How to get (good) T and c?
  - find invertible M close to A such that $Mx = b$ can be solved fast
  - note that solution of $Ax = b$ is equivalent to

$$Mx = Mx + \omega(-Ax + b)$$

    where *relaxation parameter* $\omega \neq 0$
  - choose $T = I - \omega M^{-1}A$ and $c = \omega M^{-1}b$
- Simplest choice $M = I$

representing structured matrices A if only Ax is required

$$A = \begin{bmatrix} 2.1 & -1 & & & \\ -1 & 2.1 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2.1 & -1 \\ & & & -1 & 2.1 \end{bmatrix}$$

- *structure* of matrix A:
  - symmetric with constant offdiagonals and diagonal
- matrix fully determined by one number, the diagonal elements $\alpha = 2.1$
- it would be wasteful to store A using a twodimensional array with n^2 elements
- instead, use Python procedure to implement matrix vector product

Representing matrix A as Python procedure

```
def A(x): # returns A times x
    y = 2.1*x
    y[1:]  -= x[:-1]
    y[:-1] -= x[1:]
    return y

# computing the rhs
n = 200
t = np.linspace(-1.0, 1.0,n)
a = -2
xs = (1+a*t-t**2-a*t**3)*(np.exp(-8*t**2)+(t+1)**2)

b = A(xs)     # rhs is A times exact sol
```

```
pl.plot(t, xs, label="exact sol")
pl.plot(t, b, label="data");pl.legend(loc=2);
```

Doing a couple of iterations and monitoring the error

```
x0 = np.zeros(len(xs))

kmax = 100 # number of iterations
xk = x0
e = np.zeros(kmax)
om = 0.4   # relaxation factor
for i in range(kmax):
    xk -= om*(A(xk)-b)
    e[i] = nla.norm(xk - xs)
```

```
pl.semilogy(e,'.'); # plot the error
pl.title('error as function of number of itns');
```

2.1.2.1 Matrix splitting methods (Jacobi and Gauss-Seidel)

Let $A = (a_{i,j})$ be an $n \times n$ matrix and write

$$A = L + D + U$$

where

$$L = \begin{bmatrix} 0 & & & \\ a_{2,1} & 0 & & \\ \vdots & \vdots & \ddots & \\ a_{n,1} & a_{n,2} & \cdots & 0 \end{bmatrix}, \quad D = \begin{bmatrix} a_{1,1} & & & \\ & a_{2,2} & & \\ & & \ddots & \\ & & & a_{n,n} \end{bmatrix}$$

$$U = \begin{bmatrix} 0 & a_{1,2} & \cdots & a_{1,n} \\ & 0 & \cdots & a_{2,n} \\ & & \ddots & \vdots \\ & & & 0 \end{bmatrix}.$$

Jacobi method

Use the decomposition $A = L + D + U$, we can write $Ax = b$ as

$$Dx = b - (L + U)x.$$

In case none of the diagonal entries are zero we have

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

This motivates the iteration

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

- In terms of components, Jacobi method takes the form

$$x_1^{(k+1)} = \left(b_1 - \sum_{j \neq 1} a_{1,j} x_j^{(k)}\right) / a_{1,1}$$

$$\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$$

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

$$\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$$

$$x_n^{(k+1)} = \left(b_n - \sum_{j \neq n} a_{n,j} x_j^{(k)}\right) / a_{n,n}$$

  for $k = 0, 1, \cdots$.

- Note that the computation of every component $x_j^{(k+1)}$ of $x^{(k+1)}$ is independent of other components. Thus Jacobi method can be implemented in a manner.

Jacobi method – example

Consider the system $Ax = b$ where

$$A = \begin{bmatrix} 6 & -2 & 2 \\ -2 & 5 & 1 \\ 2 & 1 & 4 \end{bmatrix}, \quad b = \begin{bmatrix} -1 \\ 8 \\ 8 \end{bmatrix},$$

which has a solution $x = [-0.5, 1, 2]^T$. Jacobi method for this system is

$$\begin{aligned} x_1^{(k+1)} &= \frac{1}{6}\left(-1 + 2x_2^{(k)} - 2x_3^{(k)}\right), \\ x_2^{(k+1)} &= \frac{1}{5}\left(8 + 2x_1^{(k)} - x_3^{(k)}\right), \\ x_3^{(k+1)} &= \frac{1}{4}\left(8 - 2x_1^{(k)} - x_2^{(k)}\right). \end{aligned}$$

Starting with $x^{(0)} = 0$, we obtain

k	$x_1^{(k)}$	$x_2^{(k)}$	$x_3^{(k)}$
0	0	0	0
1	-0.166667	1.6	2.0
2	-0.3	1.133333	1.683333
3	-0.35	1.143333	1.866667
4	-0.407778	1.086667	1.889167
5	-0.434167	1.059056	1.932222
10	-0.491339	1.008028	1.990504

Algorithm (Jacobi method)

```
input A and b
take an initial guess x_0
k = 0   (iteration number)
while a stopping criterion is not satisfied do
   for i = 1:n
      z = 0
      for j = 1:n
         if j != i do
            z = z + A(i,j)*x_0(j)
      x(i) = (b(i)-z)/A(i,i)
   k = k+1
   x_0 = x
output x and k
```

Python implementation of Jacobi method

Lab questions for Jacobi method: (to be thought about at your own leisure)

1. view the documentation of Jacobi with ?Jacobi, extend it, understand the code
2. can you recover the example above?
3. print out the number of iterations and plot iterations against error and residual norm, discuss convergence
4. do a couple of other (larger) examples for point 2
5. consider and try other stoping criteria
6. time your routine and compare to other (library) implementations
7. make routine more robust by checking against wrong input, possibly correcting ...

```
def Jacobi(A,b,tol=0.001,x0=0):
    '''solving Ax=b by the Jacobi method'''
    n = len(b)
    xk = x0*np.ones((n,))
    rk = np.dot(A,xk) - b
    dinv = 1.0/np.diag(A)    # diagonal matrix D^{-1} stored
    while (nla.norm(rk,2) > tol*nla.norm(xk,2)+tol):
        print(".",end="") # monitor progress
        xk = xk - dinv*rk
        rk = np.dot(A,xk) - b
    print("\n")
    return xk
```

```
A = np.array([[2.0, 1.0],[1.0, 2.0]])
xex = np.array([3.0,-1.0])
b = np.dot(A,xex)
print("b = ", b, ",    xex = ", xex)

xnum = Jacobi(A,b,tol=1e-5)
print("xnum =", xnum)
```

Gauss-Seidel Method

Use $A = L + D + U$ to write $Ax = b$ in the form $(L + D)x = b - Ux$, or equivalently

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

This leads to the iterative method

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

or

$$(L + D)x^{(k+1)} = b - Ux^{(k)}, \quad k = 0, 1, ...$$

Consequently one gets a method which can implemented very similarly as the Jacobi method with the iteration

$$x^{(k+1)} = D^{-1}\left(b - Lx^{(k+1)} - Ux^{(k)}\right)$$

. This is the Gauss-Seidel method.

- In terms of components, Gauss-Seidel method takes the form

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

  for $k = 0, 1, \cdots$.
- Gauss-Seidel method is closely related to Jacobi method, the only difference being that the improved values $x^{(k+1)}$ are used as soon as they are available.
- Gauss-Seidel method is not a parallel method.

Gauss-Seidel method – Example

Consider the system in the previous example, Gauss-Seidel method is

$$
\begin{aligned}
x_1^{(k+1)} &= (-1 + 2x_2^{(k)} - 2x_3^{(k)})/6, \\
x_2^{(k+1)} &= (8 + 2x_1^{(k+1)} - x_3^{(k)})/5, \\
x_3^{(k+1)} &= (8 - 2x_1^{(k+1)} - x_2^{(k+1)})/4.
\end{aligned}
$$

Starting with $x^{(0)} = 0$, we obtain

k	$x_1^{(k)}$	$x_2^{(k)}$	$x_3^{(k)}$
0	0	0	0
1	-0.166667	1.533333	1.7
2	-0.222222	1.171111	1.818333
3	-0.382407	1.083370	1.920361
4	-0.445664	1.037662	1.963416
5	-0.475251	1.017216	1.983322
10	-0.499510	1.000341	1.999670

Algorithm (Gauss-Seidel method)

While this looks very similar to the Jacobi method, note that the x(i) are changed during the outer loop so that there is less vectorisation possible . . .

```
input A and b
take an initial guess x
k = 0   (iteration number)
while a stopping criterion is not satisfied do
   for i = 1:n
      z = 0
      for j = 1:n
         if j != i do
            z = z + A(i,j)*x(j)
      x(i) = (b(i)-z)/A(i,i)
   k = k+1
output x and k
```

Python implementation of Gauss-Seidel

Lab Questions for Gauss-Seidel (GS) method:

- check the Gauss-Seidel code and compare the performance with Jacobi
- be careful when implementing the method, you might have some non-vectorisable parts ...

```
def GS(A,b,tol=0.001,x0=0):
    '''solving Ax=b by the Gauss-Seidel method'''
    n = len(b)
    xk = x0*np.ones((n,))
    dinv = 1.0/np.diag(A)    # diagonal matrix D^{-1} stored
    rk = -b.copy()
    while (nla.norm(rk,2) > tol*nla.norm(xk,2)+tol):
        print('.', end="")
        for i in range(n):
            rk[i] = np.dot(A[i,:],xk) - b[i]
            xk[i] -= dinv[i]*rk[i]
    print('\n')
    return xk

A = np.array([[2.0, 1.0],[1.0, 2.0]])
xex = np.array([3.0,-1.0])
b = np.dot(A,xex)
print("b = ", b, ",    xex = ", xex)
```

Convergence Analysis

We turn to the convergence analysis of the *general iterative method*

$$x^{(k+1)} = M^{-1}b - M^{-1}(A - M)x^{(k)}.$$

Let x^* be the solution, i.e. $Ax^* = b$. Then

$$\begin{aligned} x^{(k+1)} - x^* &= M^{-1}Ax^* + (I - M^{-1}A)x^{(k)} - x^* \\ &= (I - M^{-1}A)(x^{(k)} - x^*). \end{aligned}$$

Let $e^{(k)} = x^{(k)} - x^*$ denote the error vector and let $E = I - M^{-1}A$. Then

$$e^{(k+1)} = Ee^{(k)}.$$

By induction we obtain

$$e^{(k)} = E^k e^{(0)}, \qquad k = 0, 1, \cdots .$$

Therefore

$$\|e^{(k)}\| = \|E^k e^{(0)}\| \leq \|E^k\| \|e^{(0)}\| \leq \|E\|^k \|e^{(0)}\|.$$

where $\|E\|$ can be any induced norm on E. This shows the following result.
If $\|E\| = \rho < 1$, then for any initial guess $x^{(0)}$ the sequence $\{x^{(k)}\}$ defined above converges to the solution x^* of $Ax = b$ and

$$\|x^{(k)} - x^*\| \leq \rho^k \|x^{(0)} - x^*\|.$$

Convergence using Spectral Radius

In terms of spectral radius, the above theorem can be strengthened to the following version.
$E^k e^{(0)} \to 0$ for every $e^{(0)}$ as $k \to \infty$ if and only if $\rho(E) < 1$.
Therefore, the Jacobi method with any initial guess $x^{(0)}$ is convergent if and only if $\rho(E) < 1$.

Application to Jacobi and Gauss-Seidel Methods

- Jacobi and Gauss-Seidel methods take the form of the general iterative method with

$$M_J = D \qquad \text{and} \qquad M_{GS} = D + L$$

  respectively.

- Thus, the error matrices associated with them are

$$E_J = I - D^{-1}A = -D^{-1}(L + U),$$
$$E_{GS} = I - (D + L)^{-1}A = -(D + L)^{-1}U.$$

- If A is singular, then there is $x \neq 0$ with $Ax = 0$. Thus

$$E_J x = x \quad \text{and} \quad E_{GS} x = x.$$

  i.e. E_J and E_{GS} have an eigenvalue equal to 1. So $\rho(E_J) \geq 1$ and $\rho(E_{GS}) \geq 1$. Thus, for some initial guess, Jacobi and Gauss-Seidel methods may not converge in case A is singular.

- Conditions on A should be imposed to guarantee that $\rho(E_J) < 1$ and $\rho(E_{GS}) < 1$.
- Consider $\rho(E_J)$ first. Recall that $\rho(E_J) \leq \|E_J\|$ for any induced matrix norm. In particular

$$\rho(E_J) \leq \|E\|_\infty = \|D^{-1}(L+U)\|_\infty = \max_{1\leq i\leq n} \frac{1}{|a_{i,i}|} \sum_{\substack{j=1 \\ j\neq i}}^{n} |a_{i,j}|.$$

- Recall also that similarity transform does not change the eigenvalues and hence the spectral radius. Thus

$$\begin{aligned}\rho(E_J) = \rho(DE_JD^{-1}) = \rho((L+U)D^{-1}) &\leq \|(L+U)D^{-1}\|_1 \\ &= \max_{1\leq j\leq n} \frac{1}{|a_{j,j}|} \sum_{\substack{i=1 \\ i\neq j}}^{n} |a_{i,j}|.\end{aligned}$$

The above discussion motivates the following definition.

Diagonal Dominance

An $n \times n$ matrix A is diagonally dominant if either

$$\sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{i,j}| < |a_{i,i}|, \qquad i = 1, ..., n$$

or

$$\sum_{\substack{i=1 \\ i \neq j}}^{n} |a_{i,j}| < |a_{j,j}|, \qquad j = 1, ..., n.$$

The above argument has showed that if A is diagonally dominant then $\rho(E_J) < 1$. Therefore, we have

Convergence of Jacobi method If A is diagonally dominant, then Jacobi method converges for any initial guess.

Next we consider $\rho(E_{GS})$ for Gauss-Seidel method. The analysis is harder. However, we can show that if A is diagonally dominated, then

$$\rho(E_{GS}) < 1.$$

Thus we can conclude the following result.

Convergence of Gauss-Seidel method If A is diagonally dominated, then Gauss-Seidel method converges for any initial guess.

Questions: explore spectral radius, norms and convergence speed for various examples, make your own matrices with given eigenvalues maybe?