Successive Overrelaxation

Review and plan

our basic iteration methods so far:

$$x^{(k+1)} = x^{(k)} - \omega M^{-1}(Ax^{(k)} - b)$$

- interpretation: use residual to improve approximation
- if we choose $M = A$ and $\omega = 1$ we have

  $$\omega M^{-1}(Ax^k - b) = x^{(k)} - A^{-1}b$$

  improve by subtracting the error
- so far: choose $M = I$, $M = D$ (Jacobi) and $M = L + D$ (Gauss-Seidel) and $\omega = 1$
- convergence may be slow, use ω to improve convergence?

Relaxation parameter for general iteration

- Assume we are given a convergent iteration method of the form

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

- introduce relaxation parameter ω:

$$x^{(k+1)} = x^{(k)} - \omega M^{-1}(Ax^{(k)} - b)$$

  - $\omega = 1$ gives original method
  - $\omega < 1$ *relaxes* the updating to guard against instability
  - $\omega > 1$ *overcorrects* to get faster convergence (*over-relaxation*)

- We get fixed-point iteration form

$$x^{(k+1)} = T(\omega)x^{(k)} + c(\omega)$$

  where

$$T(\omega) = I - \omega M^{-1}A \quad c(\omega) = \omega M^{-1}b$$

- Note $T(\omega) = (1-\omega)I + \omega T(1) = I + \omega(T(1) - I)$

Choice of ω – simple matrices

- If $T(1) = qI$ then $T(\omega) = 0$ for $\omega = 1/(1-q)$
- If $T(1) = \begin{bmatrix} q_1 & \\ & q_2 \end{bmatrix}$ then
$$T(\omega) = \begin{bmatrix} 1+\omega(q_1-1) & \\ & 1+\omega(q_2-1)) \end{bmatrix}$$
- choose ω such that both convergence rates the same:
$$1+\omega(q_1-1) = -(1+\omega(q_2-1))$$
this choice minimises $\max_i |1+\omega(q_i-1)|$ and one gets
$$\omega = \frac{2}{2-q_1-q_2}$$
and
$$T(\omega) = \frac{1}{2-q_1-q_2}\begin{bmatrix} q_1-q_2 & \\ & q_2-q_1 \end{bmatrix}$$

larger matrices

- If $T(\omega)$ is diagonal matrix with diagonal elements $-1 < q_1 \leq \cdots \leq q_n < 1$ then $\omega = 2/(2 - q_1 - q_n)$ and
$$\rho(T(\omega)) = \frac{|q_1 - q_n|}{|2 - q_1 - q_n|}$$
- same holds if $T(\omega)$ diagonalisable with $q_i = |\lambda_i|$
- examples

q_1	q_n	ρ
$1/4$	$1/2$	$1/5$
0	$1/2$	$1/7$
0	q	$\frac{q}{2-q}$
q	q	0
q	1	1

- this indicates maximal improvements, in practice use estimates, trial and error

Gauss-Seidel

The Gauss-Seidel method

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

can now be recast as a correction method where one utilises the newest version of the components of x during the computation to determine the "residual". This leads to the iteration

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

If one relaxes this iteration formula as before one gets

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

This method is the SOR method if $\omega > 1$ and reduces to the Gauss-Seidel method if $\omega = 1$. As for the Jacobi method one sees that that the next iterate is computed using one Gauss-Seidel step

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

by

$$x^{k+1} = (1-\omega)x^k + \omega x_{GS}^{(k+1)}.$$

```python
def SOR(A,b,tolr=0.001,tola=0.001,x0=0,omega=1.0):
    '''solving Ax=b by the SOR 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) > tolr*nla.norm(xk,2)+tola):
        for i in range(n):
            rk[i] = np.dot(A[i,:],xk) - b[i]
            xk[i] -= omega*dinv[i]*rk[i]
    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)

xnum = SOR(A,b,omega=1.15, tolr=1e-5, tola=1e-5)
print("b = ", b, ",     xex = ", xex,"\nxnum =", xnum)
```

```
b =  [ 5.  1.] ,    xex =  [ 3. -1.]
xnum = [ 2.99999957 -0.99999971]
```

SOR performance

Consider again 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 the solution $x = [-0.5, 1, 2]^T$. The SOR method is

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

With $\omega = 1.15$ the results of an SOR calculation are:

k	$x_1^{(k)}$	$x_2^{(k)}$	$x_3^{(k)}$
0	0	0	0
1	-0.191667	1.751833	1.906446
2	-0.222227	1.036493	1.843806
3	-0.467803	1.045262	1.991903
4	-0.484375	1.002260	1.915800
5	-0.498250	1.002404	1.999566
10	-0.499998	1.000000	1.999999

- We next derive the error matrix of the SOR method.
- We have

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

Therefore, by using $U + D = A - L$,

$$\begin{aligned} x^{(k+1)} &= (\omega L + D)^{-1} \left(\omega b + Dx^{(k)} - \omega(D + U)x^{(k)}\right) \\ &= (\omega L + D)^{-1} \left(\omega b - \omega Ax^{(k)} + (\omega L + D)x^{(k)}\right) \\ &= x^{(k)} - \omega(\omega L + D)^{-1}(Ax^{(k)} - b) \end{aligned}$$

- Let x^* be the solution of $Ax = b$. Then

$$x^{(k+1)} - x^* = \left(I - \omega(\omega L + D)^{-1}A\right)(x^{(k)} - x^*)$$

- Hence the error matrix of the SOR method is

$$\begin{aligned} E_{GS}(\omega) &= I - \omega(\omega L + D)^{-1}A = (\omega L + D)^{-1}(\omega L + D - \omega A) \\ &= (\omega L + D)^{-1}((1-\omega)D - \omega U). \end{aligned}$$

- Thus, the SOR method is convergent if and only if $\rho(E_{GS}(\omega)) < 1$.
- In the error matrix

$$E_{GS}(\omega) = (\omega L + D)^{-1}((1-\omega)D - \omega U),$$

the first factor is lower triangular with $1/a_{1,1}, \cdots, 1/a_{n,n}$ on the main diagonal, and the second factor is upper triangular with $(1-\omega)a_{1,1}, \cdots, (1-\omega)a_{n,n}$ on the main diagonal. Thus

$$\det(E_{GS}(\omega)) = \left(\prod_{i=1}^{n} \frac{1}{a_{i,i}}\right)\left(\prod_{i=1}^{n}(1-\omega)a_{i,i}\right) = (1-\omega)^n.$$

- Let $\lambda_1, \cdots, \lambda_n$ be eigenvalues of $E_{GS}(\omega)$. Then

$$\lambda_1 \cdots \lambda_n = \det(E_{GS}(\omega)) = (1-\omega)^n.$$

- Thus

$$\rho(E_{GS}(\omega))^n \geq |\lambda_1| \cdots |\lambda_n| = |1-\omega|^n.$$

  which implies

$$\rho(E_{GS}(\omega)) \geq |\omega - 1|.$$

- The convergence of SOR method forces $\rho(E_{GS}(\omega)) < 1$ and hence forces $|\omega - 1| < 1$.

In order for the SOR method to be convergent, the relaxation parameter ω must satisfy $|\omega - 1| < 1$, i.e. $0 < \omega < 2$.
When A has special structure, the converse of above result also holds.

If A is symmetric and positive definite, then the SOR method converges for any initial guess if and only if $0 < \omega < 2$.

- A right choice of the relaxation parameter ω can improve the speed of convergence considerably.
- However, the calculation of the optimal relaxation parameter, i.e the parameter minimising the spectral radius, is difficult except in some simple case.
- Usually it is obtained only approximately by trial and error, based on trying several values of ω and observing the effect on the speed of convergence.

Iterative Refinement

Iterative Improvement and Gaussian Elimination

- When solving a linear system $Ax = b$ by a direct method such as Gaussian elimination, due to the presence of rounding errors, the computed solution may sometimes deviate from the exact solution.
- Iterative refinement is an iterative method to improve the accuracy of numerical solutions to linear systems.
- Assume the basic solution method is Gaussian elimination. It provides a factorisation LU that is the exact factorisation of a matrix close to A.

- The refinement method takes the following form:
  1. Take $x^{(0)}$ to be the solution obtained by Gaussian elimination.
  2. Given an approximation $x^{(k)}$, compute $b - Ax^{(k)}$ using *double precision* and round to single precision to obtain the residual $r^{(k)}$.
  3. Find $e^{(k)}$ of $LUe^{(k)} = r^{(k)}$ using back and forward substitutions.
  4. Let $x^{(k+1)} = x^{(k)} + e^{(k)}$.
  5. Continue until the difference between $x^{(k+1)}$ and $x^{(k)}$ is within a given tolerance.

Consider the system

$$\begin{bmatrix} 0.20000 & 0.16667 & 0.14286 \\ 0.16667 & 0.14286 & 0.12500 \\ 0.14286 & 0.12500 & 0.11111 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0.50953 \\ 0.43453 \\ 0.37897 \end{bmatrix}.$$

The exact solution is $x = (1, 1, 1)^T$. If floating point arithmetic with 5 digits is used then Gaussian elimination will give the computed triangular factors

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0.83335 & 1 & 0 \\ 0.71430 & 1.49874 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 0.20000 & 0.16667 & 0.14286 \\ 0 & 0.00397 & 0.00595 \\ 0 & 0 & 0.00015 \end{bmatrix}$$

and computed solution

$$x^{(0)} = (1.0345, 0.89673, 1.06667)^T.$$

The first step of iterative refinement involves calculating the residual $r^{(0)} = b - Ax^{(0)}$ in double precision arithmetic (in this case 10 digits).

$$Ax^{(0)} = \begin{bmatrix} 0.5095324653 \\ 0.4345190593 \\ 0.3789619207 \end{bmatrix} \quad \text{and so} \quad r^{(0)} = 10^{-5} \begin{bmatrix} -0.24653 \\ 1.09407 \\ 0.80793 \end{bmatrix}.$$

Then we must solve for $e^{(0)}$ using backward and forward substitution. We get

$$e^{(0)} = \begin{bmatrix} -0.03709 \\ 0.09955 \\ -0.06424 \end{bmatrix}, \qquad x^{(1)} = x^{(0)} + e^{(0)} = \begin{bmatrix} 1.00136 \\ 0.99628 \\ 1.00243 \end{bmatrix}.$$

Note that the error in the corrected solution $x^{(1)}$ is approximately 30 times smaller than those in $x^{(0)}$.

If we continue the iterations then the approximate solutions $x^{(k)}$ converge rapidly to the exact solution.

k	$x_1^{(k)}$	$x_2^{(k)}$	$x_3^{(k)}$
0	1.03845	0.89673	1.06667
1	1.00136	0.99628	1.00243
2	1.00005	0.99986	1.00009
3	1.00000	1.00000	1.00000

Iterative refinement with SOR

- Instead of a lower accurate LU decomposition we use an SOR solver as a starting point for the iterative refinement, see *next slide*

```python
# iterative refinement using SOR
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)
kiter = 5
xk = 0*b
print("xk =", xk)
for k in range(kiter):
    rk = np.dot(A,xk) - b
    ek = SOR(A,rk, tolr=1e-1, tola=1e-10)
    xk = xk - ek
    print("xk =", xk)
```

```
b =  [ 5.  1.] ,     xex =  [ 3. -1.]
xk = [ 0.  0.]
xk = [ 2.96875  -0.984375]
xk = [ 2.99951172 -0.99975586]
xk = [ 2.99999237 -0.99999619]
xk = [ 2.99999988 -0.99999994]
```