Gradient methods

The equivalent optimisation problem

Proposition:
If A is symmetric and positive definite the following are equivalent

1. x^* satisfies $Ax^* = b$
2. $f(x^*) \leq f(x)$ for all vectors x and

$$f(x) = \frac{1}{2}x^T Ax - b^T x$$

Proof.

- Let x, z be arbitrary vectors and $x^* = x - z$
- Then one has

$$\begin{aligned} f(x) &= f(x^* + z) \\ &= f(x^*) + z^T(Ax^* - b) + \frac{1}{2} z^T A z. \end{aligned}$$

- If $Ax^* = b$ then $f(x)$ is minimal for $z = 0$ as A is positive definite
- If $f(x^*) \leq f(x^* + z)$ for all vectors z then $Ax^* - b = 0$ as otherwise it would be possible to find some z for which $f(x^* + z) < f(x^*)$ (choose z such that the second term dominates in size and is negative)

- optimisation methods to solve the minimisation problem can now be used to compute the solution of the linear system of equations
- in particular, use methods of the form

$$x^{(k+1)} = x^{(k)} + \alpha_k d^{(k)}$$

  where α_k is the *stepsize* and $d^{(k)}$ the *search direction*
- use optimisation method technology to find best step size and descent direction
- We will consider two particular methods, the steepest descent method an the conjugate gradient method

Gradient method and steepest descent

- For a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$, its ∇f is defined by

$$\nabla f(x) = \left[\frac{\partial f}{\partial x_1}(x) \quad \dots \quad \frac{\partial f}{\partial x_n}(x) \right]^T.$$

  It is known that, for any vector d,

$$\frac{d}{d\alpha} f(x + \alpha d) = d^T \nabla f(x + \alpha d).$$

- Suppose we have an iterative scheme in which x^{k+1} is given by , i.e. $x_{k+1} = x_k + \alpha_k d_k$. Then an application of Taylor's formula gives

$$f(x_{k+1}) = f(x_k) + \alpha_k d_k^T \nabla f(x_k) + o(\alpha_k) \text{ as } \alpha_k \to 0.$$

- Suppose that $\nabla f(x_k) \neq 0$. We can then ensure $f(x_{k+1}) < f(x_k)$ (at least for small α_k) if

$$d_k^T \nabla f(x_k) < 0.$$

- We say that d_k is a *descent direction* if this holds
- Various different methods can be developed by the choices of descent directions d_k and step sizes α_k.
- Clearly $d_k := -\nabla f(x_k)$ is a descent direction at x_k because

$$d_k^T \nabla f(x_k) = -\|\nabla f(x_k)|_2^2 < 0.$$

With such choices of d_k, it leads to the gradient method

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k), \quad k = 0, 1, \cdots$$

with suitable choices of α_k.

- If the step size α_k is chosen optimally, in the sense that

$$f(x_k + \alpha_k d_k) = \min_{\alpha > 0} f(x_k + \alpha d_k),$$

the corresponding gradient method is called . It is easy to see that α_k must then satisfy

$$\frac{d}{d\alpha} f(x_k + \alpha d_k)\Big|_{\alpha = \alpha_k} = 0.$$

This is equivalent to

$$d_k^T \nabla f(x_k + \alpha_k d_k) = 0.$$

Example

- We return to the quadratic function

$$f(x) = \frac{1}{2}x^T Ax - b^T x.$$

  Its gradient is given by $\nabla f(x) = Ax - b$.
- Thus, the corresponding gradient method for solving $Ax = b$ becomes

$$x_{k+1} = x_k + \alpha_k d_k, \quad \text{where } d_k = b - Ax_k.$$

- If α_k is chosen optimally, we have

$$\begin{aligned} 0 &= d_k^T \nabla f(x_k + \alpha_k d_k) = d_k^T \left(A(x_k + \alpha_k d_k) - b\right) \\ &= d_k^T (Ax_k - b) + \alpha_k d_k^T A d_k \\ &= -d_k^T d_k + \alpha_k d_k^T A d_k \end{aligned}$$

  which gives

$$\alpha_k = \frac{d_k^T d_k}{d_k A d_k}$$

Summarizing the above, we obtain the following algorithm for solving $Ax = b$ with symmetric positive definite matrix A.

1. Pick a starting point $x_0 \in \mathbb{R}^n$ and set $k := 0$;
2. For $k = 0, 1, \cdots$ do
   - 2.1 $d_k = b - Ax_k$;
   - 2.2 $\alpha_k = \dfrac{d_k^T d_k}{d_k^T A d_k}$;
   - 2.3 $x_{k+1} = x_k + \alpha_k d_k$;
   - 2.4 $k = k + 1$;

```
import numpy as np
from numpy import linalg as nla

def A1(x):    # matrix vector product
    y = np.zeros(3)
    y[0] =  6*x[0] -2*x[1] + 2*x[2]
    y[1] = -2*x[0] +5*x[1] +   x[2]
    y[2] =  2*x[0] +  x[1] + 4*x[2]
    return y
x1 = np.array((-.5,1,2))
b1  = A1(x1)
n = 11
xk = np.zeros(3)
```

```
for k in range(n):
    print("k={} xk={} error={:3.2g}".format(k,xk, nla.norm(
    dk = b1 - A1(xk)
    alphak = np.dot(dk,dk)/np.dot(dk,A1(dk))
    xk = xk + alphak*dk
```

```
k=0 xk=[ 0.  0.  0.] error=2.3
k=1 xk=[-0.18169014  1.45352113  1.45352113] error=0.78
k=2 xk=[-0.1581733   1.17058415  1.73939771] error=0.46
k=3 xk=[-0.36842545  1.18674061  1.77268385] error=0.32
k=4 xk=[-0.35833855  1.0711127   1.89252145] error=0.19
k=5 xk=[-0.44550883  1.07734604  1.90587306] error=0.13
k=6 xk=[-0.44133517  1.02944466  1.95548535] error=0.079
k=7 xk=[-0.47743373  1.03203101  1.96101931] error=0.055
k=8 xk=[-0.47570526  1.0121939   1.9815653 ] error=0.033
k=9 xk=[-0.49065468  1.01326493  1.98385702] error=0.023
k=10 xk=[-0.48993888  1.00504983  1.99236568] error=0.014
```

The method of steepest descent

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 $x = [-0.5, 1, 2]^T$. The table gives the computational result by the method of steepest descent with $x_0 = 0$:

k	x_k		
0	0	0	0
1	-0.181690	1.453521	1.453521
2	-0.158173	1.170584	1.739398
3	-0.368425	1.186741	1.772684
4	-0.358339	1.071113	1.892521
5	-0.445509	1.077346	1.905873
10	-0.489939	1.005050	1.992366

- The above example shows that the method of steepest descent, like the Jacobi method converges slowly
- By further investigation, one can show that

$$(x_{k+1} - x_k)^T (x_k - x_{k-1}) = 0 \quad \text{for } k = 1, 2, \cdots$$

for the sequence $\{x_k\}$ defined by the method of steepest descent. This means that $\{x_k\}$ moves to the exact solution in a zig-zag way which makes the method slow.

An important question is how fast does this method converge. For a positive definite matrix A, the condition number (with respect to the norm) is given by

$$\kappa(A) = \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)}$$

where $\lambda_{max}(A)$ and $\lambda_{min}(A)$ are the maximum and minimum eigenvalues of A. It is worth noting that these eigenvalues satisfy

$$\lambda_{max}(A) = \max_{y \in \mathbb{R}^m} \frac{y \cdot Ay}{\|y\|_2} \qquad \lambda_{min}(A) = \min_{y \in \mathbb{R}^m} \frac{y \cdot Ay}{\|y\|_2}$$

The second ratio is known as the Rayleigh Quotient. Also recall that if B is a symmetric matrix with eigenvalues $\lambda_1(B), \ldots, \lambda_n(B)$ then the operator norm of B satisfies

$$\|B\|_2 = \max_j |\lambda_j(B)|.$$

To obtain an idea of how the gradient method converges, let us consider a steepest descent method with a constant step length. That is, let us consider the iterative method where $\alpha_k = \alpha$, a constant, and

$$d^{k+1} = r^k = b - Ax^k.$$

Then

$$x^{k+1} = x^k + \alpha(b - Ax^k).$$

Now the exact solution x must also satisfy

$$x = x + \alpha(b - Ax).$$

Subtracting Equations and leads to the equation

$$x^{k+1} - x = (I - \alpha A)(x^k - x).$$

If we let e^k denote the error at the kth step, then we have $e^{k+1} = Ee^k$ where the error matrix is given by $E = I - \alpha A$. For convergence we need $\|I - \alpha A\|_2 < 1$. Consequently, we need $\max_j |1 - \alpha\lambda_j(A)| < 1$. Since all the eigenvalues are positive, we conclude that this implies that

$$\alpha < \frac{2}{\lambda_{\max}(A)}.$$

Let us choose $\alpha = 1/\lambda_{\max}(A)$. It turns out that this choice is close to the best choice. Then we have

$$\|I - \alpha A\|_2 = 1 - \frac{\lambda_{\min}(A)}{\lambda_{\max}(A)} = 1 - \frac{1}{\kappa(A)}.$$

This shows that as the condition number increases, the convergence rate decreases. In fact we see that the number of iterations required to obtain a specified accuracy will be proportional to the condition number.

The following figure shows an example calculation based on the steepest descent method. Observe that all of the steps only go in one of two directions. What if we could combine the steps so we only took one big step in each direction? Then the solution would be found in two steps. The Conjugate gradient method tries to achieve that goal.

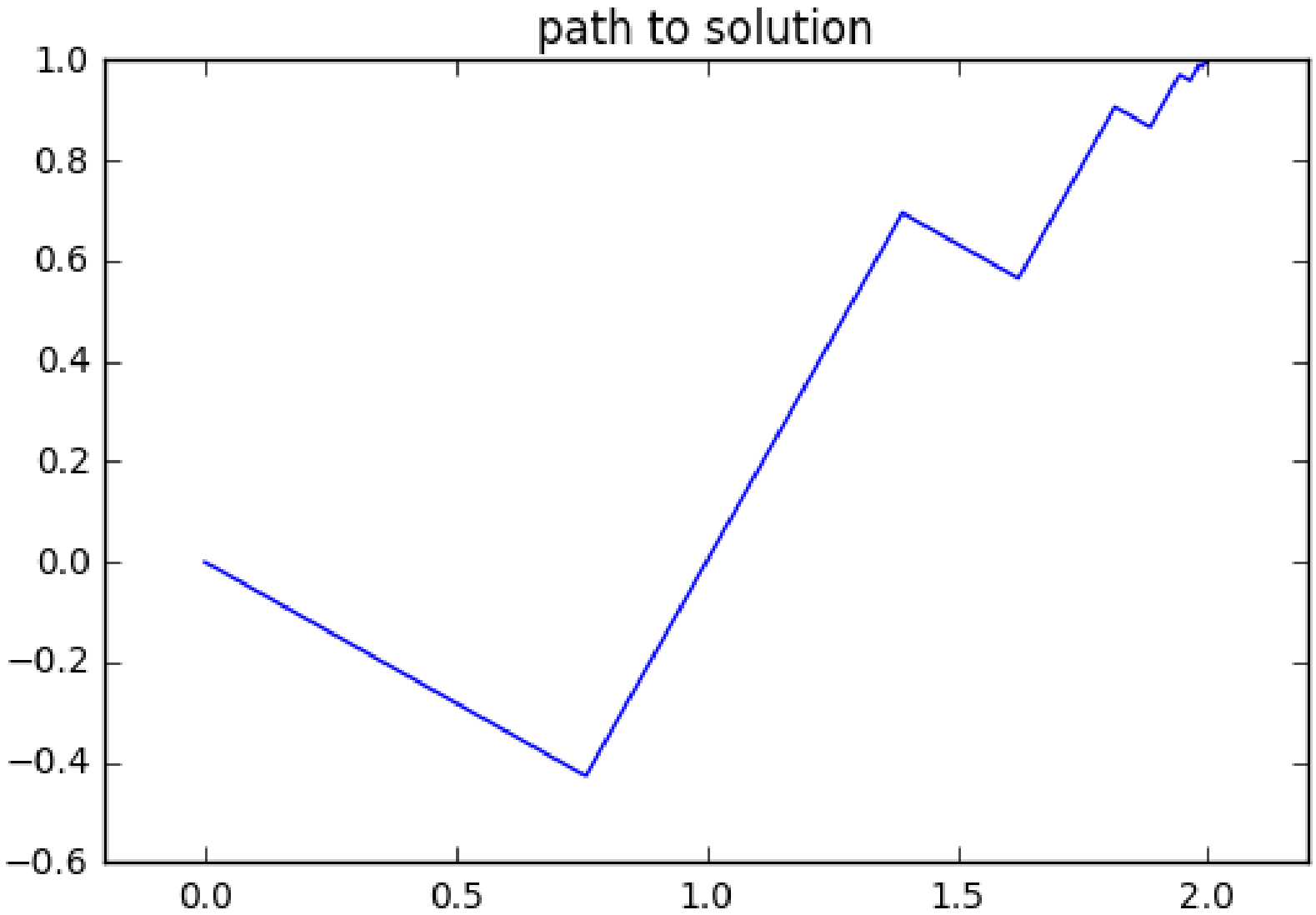
```
%matplotlib inline

def A(x):
    y1 = 1.3*x[0] -   x[1]
    y2 = -x[0] +  1.1*x[1]
    return np.array((y1,y2))
xe = np.array((2.0,1.0))
b = A(xe)

import pylab as pl

n = 51
xk = np.zeros(2)
xg = np.zeros((2,n))
for k in range(n):
    xg[:,k] = xk
    dk = b - A(xk)
    alphak = np.dot(dk,dk)/np.dot(dk,A(dk))
    xk = xk + alphak*dk
```

```
pl.title("path to solution")
pl.axis([0,2.5,-1,2]);pl.axis('equal');
pl.plot(xg[0,:],xg[1,:]);
```



Conjugate Gradient Method

- The method of steepest descent has a tendency to zig-zag into the solution. This can make the method convergent slowly.
- We next consider the which is a fast algorithm; it was proposed by Hestens and Stiefel in 1950.
- We start with the description of a general class of iterative methods — .

A set of nonzero vectors $\{d_0, d_1, \cdots, d_{n-1}\}$ in $\mathbb{R}^n$ is called (or) if

$$d_i^T A d_j = 0 \quad \text{for all } i \neq j.$$

One can check that any nonzero A-conjugate vectors $\{d_0, \cdots, d_{n-1}\}$ are linearly independent and hence form a basis of $\mathbb{R}^n$.

- Given a set of nonzero A-conjugate vectors $\{d_0, \cdots, d_{n-1}\}$ in $\mathbb{R}^n$. A is to find the minimizer of

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

in n steps by successively minimizing $f(x)$ along the individual directions d_i.

- To be more precise, let $x_0 \in \mathbb{R}^n$ be a starting point, it generates a sequence $\{x_k\}$ by setting

$$x_{k+1} = x_k + \alpha_k d_k,$$

where α_k is chosen by

$$f(x_k + \alpha_k d_k) = \min_{\alpha \in \mathbb{R}} f(x_k + \alpha d_k).$$

- α_k can be determined by solving the equation

$$\frac{d}{d\alpha} f(x_k + \alpha d_k) = 0.$$

- Note that

$$f(x_k + \alpha d_k) = \frac{1}{2}(x_k + \alpha d_k)^T A(x_k + \alpha d_k) - b^T(x_k + \alpha d_k)$$
$$= \frac{1}{2}\alpha^2 d_k^T A d_k + \alpha d_k^T A x_k + \frac{1}{2} x_k^T A x_k - b^T x_k - \alpha d_k^T b$$

  Therefore

$$\frac{d}{d\alpha} f(x_k + \alpha d_k) = \alpha d_k^T A d_k + d_k^T A x_k - d_k^T b$$
$$= \alpha d_k^T A d_k + d_k^T (A x_k - b).$$

- Thus α_k satisfies the equation

$$\alpha_k d_k^T A d_k + d_k^T (A x_k - b) = 0.$$

  Let $r_k = A x_k - b$. Then

$$\alpha_k = -\frac{d_k^T r_k}{d_k^T A d_k}.$$

- Consequently, a conjugate direction method can be formed by repeating the steps

$$
\begin{aligned}
r_k &= Ax_k - b, \\
\alpha_k &= -\frac{d_k^T r_k}{d_k^T A d_k}, \\
x_{k+1} &= x_k + \alpha_k d_k
\end{aligned}
$$

  from an initial guess x_0, once a set of nonzero A-conjugate vectors $\{d_0, \cdots, d_{n-1}\}$ are known.
- How to find a set of nonzero A-conjugate vectors $\{d_0, \cdots, d_{n-1}\}$ in a computationally efficient way?
- is a conjugate direction method with the conjugate vectors constructed successively during computation.

- It starts with

$$r_0 = Ax_0 - b \quad \text{and} \quad d_0 = -r_0$$

and construct each new d_{k+1} as the linear combination of r_{k+1} and d_k of the form

$$d_{k+1} = -r_{k+1} + \beta_{k+1} d_k,$$

where β_{k+1} is chosen so that d_{k+1} and d_k are A-conjugate, i.e.

$$0 = d_{k+1}^T A d_k = -r_{k+1}^T A d_k + \beta_{k+1} d_k^T A d_k$$

and hence

$$\beta_{k+1} = \frac{r_{k+1}^T A d_k}{d_k^T A d_k}.$$

- Combining this construction of d_{k+1} with ([3.24.2]), it leads to the following conjugate gradient method.

1. Pick a starting point $x_0 \in \mathbb{R}^n$;
2. Set $r_0 := Ax_0 - b$, $d_0 := -r_0$, and $k := 0$;
3. While $r_k \neq 0$ do
   - 3.1 $\alpha_k = -\dfrac{r_k^T d_k}{d_k^T A d_k}$;
   - 3.2 $x_{k+1} = x_k + \alpha_k d_k$;
   - 3.3 $r_{k+1} = Ax_{k+1} - b$;
   - 3.4 $\beta_{k+1} := \dfrac{r_{k+1}^T A d_k}{d_k^T A d_k}$;
   - 3.5 $d_{k+1} := -r_{k+1} + \beta_{k+1} d_k$;
   - 3.6 $k = k + 1$;

- One can show that the vectors $\{d_0, d_1, \cdots, d_{n-1}\}$ constructed by the algorithm are A-conjugate.
- The sequence $\{x_k\}$ generated by the conjugate gradient method converges to the solution x^* of $Ax = b$ in at most n steps.

Consider the linear system $Ax = b$ of size $n = 600$, where

$$A = \begin{bmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 4 & -1 \\ & & & -1 & 4 \end{bmatrix}, \qquad b = \begin{bmatrix} 1 \\ 2 \\ 3 \\ \vdots \\ n \end{bmatrix}.$$

```
# some other data
def A2(x): # the matrix vector product
    y = 4*x
    y[1:]  -= x[:-1]
    y[:-1] -= x[1:]
    return y
x2 = (np.linspace(0,1,102)*(1-np.linspace(0,1,102)))[1:-1]*
b2 = A2(x2)

A = A1
b = b1
xex = x1
```

```
xk = np.zeros(len(b))
rk = A(xk)-b
dk = -rk
for k in range(5):
    print("k={} xk={} error={:3.2g}".format(k,xk, nla.norm(
    Adk = A(dk)
    dkAdk = np.dot(dk,A(dk))
    alphak = -np.dot(rk,dk)/dkAdk
    xk = xk + alphak*dk
    rk = A(xk) - b
    betak = np.dot(rk,A(dk))/dkAdk
    dk = -rk + betak*dk

k=0 xk=[ 0.  0.  0.] error=2.3
k=1 xk=[-0.18169014  1.45352113  1.45352113] error=0.78
k=2 xk=[-0.16248694  1.20250784  1.78683386] error=0.45
k=3 xk=[-0.5  1.   2. ] error=3.1e-16
k=4 xk=[-0.5  1.   2. ] error=3.1e-16
```