Scientific Computing

A practical Companion

8th Notebook

Author:

Walter Hoffmann (Korteweg-de Vries Institute for Mathematics, UvA)

February - March, 2008

Solving systems of linear equations IV

Iterative solution methods by minimization methods

▽ Using minimization techniques

A totally different approach follows from creating a function $Q(x)$ that takes its minimum at the solution of the linear system under consideration and then using the techniques that are available for finding a minimum of a function.

A simple choice for such a function is

$$Q(x) = (b - A\,x)^T\,(b - A\,x).$$

In the situation that the matrix A of the linear system to be solved is *symmetric* and *positive definite,* an interesting choice for such a function is:

$$Q(x) = x^T\,A\,x - 2\,x^T\,b.$$

<u>Theorem</u>

The quadratic function $Q(x)$ as defined above has a unique minimum for $x = A^{-1}\,b$.

<u>Proof</u>

Consider the vector $(x + p)$ for $p \neq 0$ then we have:

$$\begin{aligned} Q(x+p) &= (x+p)^T\,A\,(x+p) - 2\,(x+p)^T\,b \\ &= Q(x) + x^T\,A\,p + p^T\,A\,x + p^T\,A\,p - 2\,p^T\,b \\ &= Q(x) + 2\,p^T\,A\,x + p^T\,A\,p - 2\,p^T\,b \\ &= Q(x) + p^T\,A\,p \end{aligned}$$

From the fact that A is nonsingular we conclude $p^T\,A\,p \neq 0$ and from the fact that A is a positive definite matrix it follows that for any $p \neq 0$ $p^T\,A\,p > 0$ so that $Q\,(x + p) > Q\,(x)$.

▽ Steepest descent

The steepest descent algorithm for minimizing the function

$$Q(x) = x^T\,A\,x - 2\,x^T\,b.$$

is as follows.

Assume that x is the $i-$th approximant; (we leave out the iteration index for simplicity).

The steepest descent direction for any function $Q(x)$ is the direction of the *negative gradient*: which in this case gives:

$$-\nabla Q = -2\,A\,x + 2\,b\,.$$

We are only interested in the direction p of the next step, so we leave out the factor 2 and set $p = b - A\,x$.

Then we have to decide the length of the step in that direction. This is a matter of finding α that minimizes $Q(x + \alpha\,p)$, seen as a function of α.

$$\begin{aligned} Q(x + \alpha\,p) &= (x + \alpha\,p)^T A\,(x + \alpha\,p) - 2\,(x + \alpha\,p)^T\,b \\ &= \alpha^2\,p^T A\,p + \alpha\left(p^T A\,x + x^T A\,p - 2\,p^T\,b\right) + \text{'rest'} \\ &= \alpha^2\,p^T A\,p + 2\,\alpha\left(p^T A\,x - 2\,p^T\,b\right) + \text{'rest'} \\ &= \alpha^2\,p^T A\,p - 2\,\alpha\,p^T\,p + \text{'rest'}. \end{aligned}$$

This takes its minimum for $\frac{dQ}{d\alpha} = 0$, which is for $\alpha = \frac{p^T\,p}{p^T\,A\,p}$

This defines the 'backbone' of the Steepest descent algorithm:

1. Find direction: $p = b - A\,x$
2. Find step: $\alpha = \frac{p^T\,p}{p^T\,A\,p}$
3. Perform update $x := x + \alpha\,p$

▿ Implementation of Steepest descent :

As an example and for comparison reasons we give an implementation of the Steepest Descent algorithm to solve the well known 20 × 20 problem; A, its diagonal DD and b are given here.

Matrix A:

In[731]:=

```
A = {{4, −1, 0, 0, −1, −1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {−1, 4, −1, 0, 0, 0, −1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, −1, 4, −1, 0, 0, 0, −1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, −1, 4, −1, 0, 0, 0, −1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {−1, 0, 0, −1, 4, 0, 0, 0, 0, −1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {−1, 0, 0, 0, 0, 4, −1, 0, 0, −1, −1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, −1, 0, 0, 0, −1, 4, −1, 0, 0, 0, −1, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, −1, 0, 0, 0, −1, 4, −1, 0, 0, 0, −1, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, −1, 0, 0, 0, −1, 4, −1, 0, 0, 0, −1, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, −1, −1, 0, 0, −1, 4, 0, 0, 0, 0, −1, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, −1, 0, 0, 0, 0, 4, −1, 0, 0, −1, −1, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, −1, 0, 0, 0, −1, 4, −1, 0, 0, 0, −1, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, −1, 0, 0, 0, −1, 4, −1, 0, 0, 0, −1, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, −1, 0, 0, 0, −1, 4, −1, 0, 0, 0, −1, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, −1, −1, 0, 0, −1, 4, 0, 0, 0, 0, −1},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, −1, 0, 0, 0, 0, 4, −1, 0, 0, −1},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, −1, 0, 0, 0, −1, 4, −1, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, −1, 0, 0, 0, −1, 4, −1, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, −1, 0, 0, 0, −1, 4, −1},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, −1, −1, 0, 0, −1, 4}};
```

Right-hand side:

In[732]:=

```
b = {0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 1., 1.};
```

▽ Check on the solution by direct method:

In[733]:=

```
u = LinearSolve[A, b]
```

Out[733]=

```
{0.2, 0.2, 0.2, 0.2, 0.2, 0.4, 0.4, 0.4, 0.4, 0.4,
 0.6, 0.6, 0.6, 0.6, 0.6, 0.8, 0.8, 0.8, 0.8, 0.8}
```

▽ Diagonal of A:

In[734]:=

```
DD = DiagonalMatrix[
        {4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
          4, 4, 4, 4, 4, 4, 4, 4, 4, 4}];
```

```
In[735]:=
DD // MatrixForm

Out[735]//MatrixForm=
4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4

In[736]:=
x = {0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.};

i = 0;
```

```
In[810]:=
p = b − A.x;
rho = p.p
q = A.p;
alf = rho/(p.q);
x = x + alf p;
i = i + 1;

Out[811]=
0.00854045

In[816]:=
i
rho
x

Out[816]=
13

Out[817]=
0.00854045

Out[818]=
{0.174561, 0.174561, 0.174561, 0.174561, 0.174561,
 0.366699, 0.366699, 0.366699, 0.366699, 0.366699,
 0.558838, 0.558838, 0.558838, 0.558838, 0.558838,
 0.779419, 0.779419, 0.779419, 0.779419, 0.779419}
```

A specific draw back with Steepest Descent (St.D.) will be illustrated by a simple problem.
Consider St.D. for minimizing $Q(x_1, x_2) = x_1^2 + 100\, x_2^2$ and start with $\{x_1, x_2\} = \{5, 5\}$
In the terminology of St.D. :

```
In[819]:=
B = {{1, 0}, {0, 100}};
c = {0., 0.};

In[821]:=
x = {5., 5.};
i = 0;
```

```
In[841]:=
p = c − B.x;
rho = p.p
q = B.p;
alf = rho/(p.q);
x = x + alf p;
i = i + 1;

Out[842]=
0.00230754

In[847]:=
i
rho
x

Out[847]=
4

Out[848]=
0.00230754

Out[849]=
{0.000470833, 0.000470833}
```

And now the same minimization problem with another starting value:

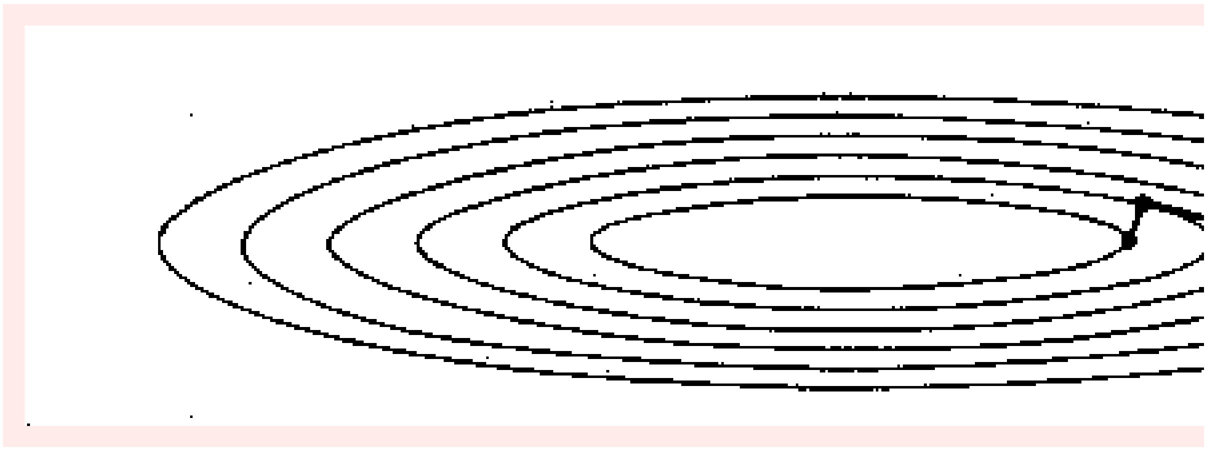
```
In[850]:=
x = {5., 0.5};
i = 0;

In[930]:=
p = c − B.x;
rho = p.p
q = B.p;
alf = rho/(p.q);
x = x + alf p;
i = i + 1;

Out[931]=
0.00474033
```

```
In[936]:=
i
x
Out[936]=
14
Out[937]=
{0.0339116, 0.00339116}
```

More interesting will be to see how Q diminishes and in what direction the successive steps are chosen:

```
In[938]:=
x = {5, 0.5};
i = 0;
fun = x.B.x
Out[940]=
50.
In[1018]:=
p = c – B.x
rho = p.p;
q = B.p;
alf = rho/(p.q);
x = x + alf p;
fun = x.B.x
i = i + 1;
Out[1018]=
{–0.139813, 0.0139813}
Out[1023]=
0.0095793
In[1025]:=
i
x
Out[1025]=
12
Out[1026]=
{0.0692073, 0.00692073}
```

A solution for the phenomenon of calculatinging repeatedly the same search direction over and over again (as shown in the next picture) is to require that successive search directions are more or less independent.



One way of solving this is presented in the Conjugate Gradients Method that is treated next.

Conjugate Gradients Method

The key idea of the Conjugate Gradients method (CG method) is that successive steps are such that they form a row of mutually orthogonal vectors, or more precise: ***A-orthogonal*** vectors; this has been observed and analyzed by Hestenes and Stiefel; allbeit that they designed the method as a direct method. This is justified by the observation that in exact arithmetic the CG method should always converge in at most *n* steps, where *n* equals the size of the linear system.

The next step will in principal be taken in the direction of the residual vector $\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A}\,\boldsymbol{x}$, like in St.D. (*x* being the current iterate). But now we will only use the component that is ***A-orthogonal*** with respect to the earlier steps.

$$\boldsymbol{p}^{(k+1)} = \boldsymbol{r}^{(k+1)} + \beta\,\boldsymbol{p}^{(k)}$$

where β is determined such that $\boldsymbol{p}^{(k+1)}$ and $\boldsymbol{p}^{(k)}$ are ***A-orthogonal***.

This gives:

$$0 = \left(\boldsymbol{p}^{(k)}, \boldsymbol{A}\,\boldsymbol{p}^{(k+1)}\right) = \left(\boldsymbol{p}^{(k)}, \boldsymbol{A}\,\boldsymbol{r}^{(k+1)}\right) + \beta\left(\boldsymbol{p}^{(k)}, \boldsymbol{A}\,\boldsymbol{p}^{(k)}\right)$$

$$\Longrightarrow \beta = -\frac{\left(\boldsymbol{p}^{(k)}, \boldsymbol{A}\,\boldsymbol{r}^{(k+1)}\right)}{\left(\boldsymbol{p}^{(k)}, \boldsymbol{A}\,\boldsymbol{p}^{(k)}\right)}$$

As a consequence of the symmetry of A we have

$$\left(\boldsymbol{p}^{(k)}, \boldsymbol{A}\,\boldsymbol{r}^{(k+1)}\right) = \left(\boldsymbol{r}^{(k+1)}, \boldsymbol{A}\,\boldsymbol{p}^{(k)}\right)$$

Moreover, the symmetry of *A* ensures that once a new direction $\boldsymbol{p}^{(k+1)}$ has been calculated ***A-orthogonal*** with respect to $\boldsymbol{p}^{(k)}$, it is automatically ***A-orthogonal*** with respect to all the steps used earlier; this sort of comes as a 'bonus'. We will not prove it here.

This defines the 'backbone' of the Conjugate Gradient algorithm:

1. Find new residual:	$r = b - A x$
2a. Find step "correction":	$\beta = -\frac{r^T A p}{p^T A p}$
2b. Determine step direction:	$p := r + \beta p$
3. Find step size:	$\alpha = \frac{p^T p}{p^T A p}$
4. Perform update	$x := x + \alpha p$

There is still another feature in the CG algorithm.

As it is formulated now, it looks as if two times a (presumed expensive) multiplication with *A* has to be performed; once for *Ax* and once for *Ap*.

The following relations hold:

$$\begin{aligned} r^{(k+1)} &= b - A x^{(k+1)} \\ &= b - A\left(x^{(k)} + \alpha p^{(k)}\right) \\ &= \left(b - A x^{(k)}\right) - \alpha A p^{(k)} \\ &= r^{(k)} - \alpha A p^{(k)} \end{aligned}$$

and $A p^{(k)}$ is the other multiplication by *A* that is needed.

Moreover, several other identities and choices are used in modern formulations of the CG algorithm; it is one of the most heavily investigated algorithms over the last decades and variants are still studied and presented.

In well known implementations, the value for β is calculated via a different formula, which is mathematically equivalent, but numerically superior.

Implementation and analysis of CG algorithm:

First we look at the performance for the simple problems where St.D. had some difficulties.

In[1027]:=

```
B = {{1, 0}, {0, 100}};
c = {0., 0.};
```

In[1029]:=

```
x = {5., 5.};
```

In[1030]:=

```
r = c − B.x;
p = r;
rhokm1 = r.r
i = 0;
```

Out[1032]=

```
250025.
```

In[1043]:=

```
q = B.p;
alf = rhokm1 / (p.q);
x = x + alf p;
r = r − alf q;
rho = r.r
bet = rho / rhokm1;
p = r + bet p;
rhokm1 = rho;
i = i + 1;
```

Out[1047]=

4.97318×10^{-23}

In[1052]:=

```
i
x
```

Out[1052]=

```
2
```

Out[1053]=

$\{-8.88178 \times 10^{-16}, -7.10537 \times 10^{-14}\}$

Another (very unfavorite) choice for the starting vector:

In[1054]:=

```
x = {20., 1.};
```

```
In[1055]:=
r = c − B.x;
p = r;
rhokm1 = r.r
i = 0;
```

```
Out[1057]=
10400.
```

```
In[1068]:=
q = B.p;
alf = rhokm1 / (p.q);
x = x + alf p;
r = r − alf q;
rho = r.r
bet = rho / rhokm1;
p = r + bet p;
rhokm1 = rho;
i = i + 1;
```

Out[1072]=

6.43002×10^{-26}

```
In[1077]:=
i
x
```

```
Out[1077]=
2
```

Out[1078]=

$\{0., -2.57433 \times 10^{-15}\}$

An important and well known property of the CG method is that a purely quadratic function will be minimized in at most two steps, independent of the choice of starting vector.

Next we go back to the original 20 × 20 problem; matrix *A* and right-hand side vector *b* have still their original value.

```
In[1079]:=
x = {0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.};

i = 0;
```

```
In[1081]:=
r = b − A.x;
p = r;
rhokm1 = r.r
i = 0;
```

```
Out[1083]=
5.
```

```
In[1112]:=
q = A.p;
alf = rhokm1 / (p.q);
x = x + alf p;
r  =  r − alf q;
rho = r.r
bet = rho / rhokm1;
p = r + bet p;
rhokm1 = rho;
i = i + 1;
```

Out[1116]=

7.69139×10^{-32}

In[1121]:=

```
i
rho
x
```

Out[1121]=

```
4
```

Out[1122]=

7.69139×10^{-32}

Out[1123]=

```
{0.2, 0.2, 0.2, 0.2, 0.2, 0.4, 0.4, 0.4, 0.4, 0.4,
 0.6, 0.6, 0.6, 0.6, 0.6, 0.8, 0.8, 0.8, 0.8, 0.8}
```

It can be shown that the convergence of CG is ruled by:

$$\frac{\| \overline{x} - x_k \|_2}{\| \overline{x} - x_0 \|_2} \leq 2 \gamma^k \sqrt{\kappa_2 (A)} ;$$

where

$$\gamma = \frac{\sqrt{\kappa_2 (A)} - 1}{\sqrt{\kappa_2 (A)} + 1} \quad \text{and} \quad \kappa_2 (A) = \frac{\lambda_{\text{max}}}{\lambda_{\text{min}}} ;$$

▽ For the case that CG shows very slow convergence or is even diverging during a couple of steps, the accuracy of the finite-precision solution may be influenced in a negative way by the fact that we calculate the residual vector in a recursive way.

This can be analyzed as follows. Let us track the computation of ***x, q*** and ***r*** in successive steps; we need not to consider the actual calculations of ***p*** and α .

The algorithm starts with:

$$\begin{aligned} &x_0 \text{ arbitrary } ; \\ &r_0 = b - A\, x_0 ; \\ &p_0 = r_0 ; \end{aligned}$$

and further succesively 'until convergence':

$$\begin{array}{lll} x_1 = x_0 + \alpha_1 p_0 ; & q_1 = A p_0 ; & r_1 = r_0 - \alpha_1 q_1 ; \\ x_2 = x_1 + \alpha_2 p_1 ; & q_2 = A p_1 ; & r_2 = r_1 - \alpha_2 q_2 ; \\ \vdots & \vdots & \vdots \\ x_k = x_{k-1} + \alpha_k p_{k-1} ; & q_k = A p_{k-1} ; & r_k = r_{k-1} - \alpha_k q_k ; \\ \vdots & \vdots & \vdots \end{array}$$

What equation does x_k satisfy?

$$\begin{aligned} x_k &= x_{k-1} + \alpha_k p_{k-1} \\ &= x_{k-2} + \alpha_{k-1} p_{k-2} + \alpha_k p_{k-1} \\ &\vdots \\ &= x_0 + \alpha_1 p_0 + \alpha_2 p_1 + \alpha_3 p_2 + \cdots + \alpha_{k-1} p_{k-2} + \alpha_k p_{k-1} \end{aligned}$$

and as a consequence:

$$\begin{aligned} \mathrm{A}x_k &= A x_0 + \alpha_1 A p_0 + \alpha_2 A p_1 + \cdots + \alpha_{k-1} A p_{k-2} + \alpha_k A p_{k-1} \\ &= A x_0 + \alpha_1 q_1 + \alpha_2 q_2 + \cdots + \alpha_{k-1} q_{k-1} + \alpha_k q_k \\ &= (b - r_0) + (r_0 - r_1) + \cdots + (r_{k-2} - r_{k-1}) + (r_{k-1} - r_k) \\ &= b - r_k \end{aligned}$$

but only so in exact arithmetic! Whenever an iterative algorithm is applied, it may be the case that in each step original information is 'fed into the process'. For instance, if in each step the residual vector was actually calculated from its definition **($r_k = b - A x_k$),** then each time the original vector ***b*** and matrix ***A*** were used, and therefore 'fed into the process'. In the CG algorithm, the residual vector is calculated recursively, and as long as the norm of this residual vector is decreasing fast enough (say at least by a factor 2 in each step) then the result is still accurate enough. If, however, the norm of the residual vector is increasing during some steps, then this will have a negative influence on the final accuracy. This is analyzed as follows. The floating-point calculation of r_k causes the result to show a rounding error. This can be modeled as follows:

$$\begin{aligned} &\hat{r}_0 = b - A x_0 + s_0 ; \\ &\hat{r}_p = \hat{r}_{p-1} - \hat{\alpha}_p \hat{q}_p + s_p ; \; p = 1, 2, \cdots \end{aligned}$$

where variables with a 'hat' denote calculated results and s_0 and s_p denote vectors that represent the rounding error in that calculation. For the sizes we may assume

$$\| s_0 \| \le \eta \, \| b \|$$

and

$$\| s_p \| \le \eta \, \| \hat{r}_{p-1} \| ; \; p = 1, 2, \cdots$$

where η denotes the machine-precision.

If we now analyze the equation that is solved by the calculated x_k then we see:

$$\begin{aligned} fl(Ax_k) &= fl(A x_0 + \alpha_1 q_1 + \alpha_2 q_2 + \cdots + \alpha_{k-1} q_{k-1} + \alpha_k q_k) \\ &= (b - \hat{r}_0 + s_0) + (\hat{r}_0 - \hat{r}_1 + s_1) + \cdots \\ &\qquad + (\hat{r}_{k-2} - \hat{r}_{k-1} + s_{k-1}) + (\hat{r}_{k-1} - \hat{r}_k + s_k) \\ &= b + s_0 + s_1 + \cdots + s_{k-1} + s_k - \hat{r}_k \end{aligned}$$

and therefore

$$\begin{aligned} \| b - fl(Ax_k) \| &\le \| s_0 + s_1 + \cdots + s_{k-1} + s_k - \hat{r}_k \| \\ &\le \eta (\| b \| + \| \hat{r}_0 \| + \cdots + \| \hat{r}_{k-1} \|) + \| \hat{r}_k \| \end{aligned}$$

If the succesive residual vectors are decreasing in norm by a factor ρ then this reduces to

$$\begin{aligned} &\| b - fl(Ax_k) \| \\ &\quad \le \eta \left(\| b \| + \frac{1 - \rho^k}{1 - \rho} \| \hat{r}_0 \| \right) + \| \hat{r}_k \| \end{aligned}$$

Even for moderately slow convergence, we still can judge by the size of $\hat{\mathbf{r}}_k$ how well the currant iterand x_k does satisfy the original equation. Depending on the condition of the matrix, the actual error may of course be large.

If however the convergence is not monotone, then we can no longer bound the factor in front of $\| \hat{\mathbf{r}}_0 \|$ and it may be possible that the actual residual vector for the iterand x_k is very large.

▽ An improvement of the CG algorithm is given by considering an appropriate *preconditioning*. This is the same idea as was used in the fixed-point iteration case: look for a matrix that is in some sense close to *A* and for which a linear system is easy to solve.
Instead of solving $Ax = b$, one is going to solve $F^{-1} A x = F^{-1} b$.

For the preconditioning to work properly, it must be required that *F* is also symmetric, positive definite.

For good result, *F* should prefereably be chosen such that

$$\kappa_2 \left(F^{-1} A \right) << \kappa_2 (A).$$

The CG algorithm can be reformulated such that the operations are not applied on $F^{-1} A$ (which is itself probably no longer symmetric !) but on a similar and symmetric pos.def. matrix.

This is as follows :

$$F = U \Lambda U^T \implies F^{1/2} \equiv U \Lambda^{1/2} U^T$$

Consequently:

$$F^{-1} A x = F^{-1} b$$

is equivalent to

$$\left(F^{-1/2}\,A\,F^{-1/2}\right)\left(F^{1/2}\,x\right) = F^{-1/2}\,b$$

for a coefficientmatrix that is again symmetric and positive definite; furthermore:

$$F^{-1/2}\left(F^{-1/2}\,A\,F^{-1/2}\right)F^{1/2} = F^{-1}\,A$$

From which we conclude that the symmetric positive definite matrix ($F^{-1/2}\,A\,F^{-1/2}$) and ($F^{-1}\,A$) are *similar*, (which means that they have the same eigenvalues). Therefore, a supposedly better conditioned matrix $F^{-1}\,A$ forms "the key" to constructing an also better conditioned matrix $F^{-1/2}\,A\,F^{-1/2}$ which is moreover *symmetric and positive definite*.

The CG algorithm can be reformulated in such a way that we need not to form $F^{-1/2}\,A\,F^{-1/2}$ explicitly, but instead of that may work with $F^{-1}\,A$ (be aware that F^{-1} is never calculated explicitly; if a vector $z = F^{-1}\,Au$ is needed, then z is computed by solving $Fz = Au$ for z).

The reformulated algorithm is given hereafter; the formulation presented here is the same as found in K&K.

First we take as the preconditioning matrix the identity matrix, so that the results should be equal to those we found before.

```
In[1124]:=
Id = IdentityMatrix[20];
```

```
In[1125]:=
x = {0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.};
```

```
In[1126]:=
r = b;
res = r.r;
z = LinearSolve[Id, r];
p = z;
rhokm1 = z.r
i = 0;

Out[1130]=
5.
```

In[1132]:=

```
While[res > 10^(-20) && i < 20,
  q = A.p;
  alf = rhokm1 / (p.q);
  x = x + alf p;
  r  =  r  −  alf q;
  res = r.r;
  i = i + 1;
  z = LinearSolve[Id, r];
  rho = z.r;
  bet = rho / rhokm1;
  p = z + bet p;
  rhokm1 = rho;
  Print["i =  ", i, "  res=  ", res]
]
```

```
i  =   1   res=   1.25
i  =   2   res=   0.555556
i  =   3   res=   0.3125
i  =   4   res=   7.69139 × 10^-32
```

In[1133]:=

```
x
```

Out[1133]=

```
{0.2, 0.2, 0.2, 0.2, 0.2, 0.4, 0.4, 0.4, 0.4, 0.4,
 0.6, 0.6, 0.6, 0.6, 0.6, 0.8, 0.8, 0.8, 0.8, 0.8}
```

For a description of a parallel implementation using MPI see:

George Em Karniadakis and Robert M. Kirby II; *Parallel Scientific Computing in C++ and MPI*;

(pp. 510 and further); Cambridge University Press 2003;

Close all sections