Incorporating preconditioners

In practice, high dimensional matrices are usually *sparse*, that is, the number of non zero entries in each row is small. Iterative solvers require preconditioning to be fast: rather then solving

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{1}$$

the *preconditioned system*

$$\mathbf{M}^{-1}\mathbf{A}\mathbf{x} = \mathbf{M}^{-1}\mathbf{b} \tag{2}$$

is solved. Here, $\mathbf{M}$ approximates $\mathbf{A}$ in some (weak) sense and systems $\mathbf{M}\mathbf{u} = \mathbf{r}$ can efficiently be solved. *Preconditioners* $\mathbf{M}$ are often of the form $\mathbf{M} = \mathbf{L}\mathbf{U}$ with $\mathbf{L}$ and $\mathbf{U}$ sparse non-singular triangular matrices, $\mathbf{L}$ lower and $\mathbf{U}$ upper triangular. Via

$$\mathbf{L}\mathbf{u}' = \mathbf{r} \quad \text{and} \quad \mathbf{U}\mathbf{u} = \mathbf{u}',$$

the system

$$\mathbf{M}\mathbf{u} = \mathbf{r}$$

can efficiently be solved for $\mathbf{u}$.

Now suppose we have a MATLAB subroutine, say

```
function [x,hist,success] = ...
                        pcg(A,b,x0,kmax,tol);
```

that solves (1) iteratively.[1] Here, at the **input** side

- `A` is the matrix $\mathbf{A}$,
- `b` is the vector $\mathbf{b}$,
- `x0` is the initial guess $\mathbf{x}_0$ (often $\mathbf{x}_0 = \mathbf{0}$, `x0=0*b` in MATLAB),
- `kmax` is the maximum number of iteration steps that is allowed and
- `tol` is the required *reduction* of the residual norm, that is, stop the iteration if

$$\|\mathbf{r}_k\|_2/\|\mathbf{r}_0\|_2 < \texttt{tol}.$$

At the **output** site

- `x` is the solution as computed by the subroutine (i.e., `x`= $\mathbf{x}_k$ with k=`kmax` or $\|\mathbf{r}_k\|_2/\|\mathbf{r}_0\|_2$ <`tol`),
- `hist` is the convergence history, that is, the sequence $(\|\mathbf{r}_j\|_2/\|\mathbf{r}_0\|_2)_{j=0}^k$ of intermediate residual norm reductions,
- `success` is 1 if $\|\mathbf{r}_k\|_2/\|\mathbf{r}_0\|_2$ <`tol` and 0 else.

To use this routine we have to form the vector $\mathbf{M}^{-1}\mathbf{b}$, which is not a problem. But, we also either have
1) to form the matrix $\mathbf{M}^{-1}\mathbf{A}$ or
2) to adapt the code `pcg.m` to compute the vector $\mathbf{c} = \mathbf{M}^{-1}\mathbf{A}\mathbf{r}$ from $\mathbf{r}$.

[1] Three dots ... form a Matlab command: they indicate that the command continues on next line.

Discussion 1. The matrices $\mathbf{L}^{-1}$, $\mathbf{U}^{-1}$ will be full. Therefore, the matrix $\mathbf{M}^{-1}\mathbf{A}$ is full. Forming the matrix $\mathbf{M}^{-1}\mathbf{A}$ explicitly is extremely expensive (in, both computational costs as well as in memory and is often even not possible). Even if the matrix $\widetilde{\mathbf{A}} \equiv \mathbf{M}^{-1}\mathbf{A}$ would be available, the computation of $\mathbf{c} = \widetilde{\mathbf{A}}\mathbf{r}$ from $\mathbf{r}$ by multiplication by $\widetilde{\mathbf{A}}$ (at $2\,n^2$ flop) is much more expensive than performing the steps

$$\begin{aligned} &\mathbf{c}_1 = \mathbf{A}\mathbf{r}, \\ &\texttt{solve } \mathbf{L}\mathbf{c}_2 = \mathbf{c}_1 \text{ for } \mathbf{c}_2, \\ &\texttt{solve } \mathbf{U}\mathbf{c} = \mathbf{c}_2 \text{ for } \mathbf{c}, \end{aligned} \tag{3}$$

(at kn flop with k the maximum number of non-zeros in each row of $\mathbf{L}$, $\mathbf{U}$ and $\mathbf{A}$ together).

Discussion 2. Adapting the code `pcg.m` to perform the steps as in (3) is not attractive:
a) the code has to be adapted at several places (in the input list of `pcg` [we need `A, L, U` instead of `A`], and in all lines where the command `c=A*u` is used [with the danger of missing one matrix-vector multiplication]),
b) other types of preconditioning would require similar but new adaptations, and
c) if you want to try another solver (than PCG), then the code for this solver has to be adapted as well.

Matrix or subroutine

An efficient way out is to call a function subroutine, say `PMV.m`, in the `pcg.m` code whenever a MV (matrix vector multiplication) is required:

```
function [x,hist,success] = ...
                    pcg(MV,b,x0,kmax,tol);
```

where now `MV` is the string of the name of this subroutine, as `MV='PMV'`, and in `pcg.m` the commands with

```
      c=A*u;
```

are replaced by

```
      c=feval(MV,u);.
```

`PMV.m` could be the function subroutine

```
function c=PMV(u)                              (4)
        c1=A*u; c2=L\c1; c=U\c2;
return
```

We can make the routine `PMV.m` as efficient as possible without fiddling with the routine `pcg.m`. Now, we can call `pcg.m` in the command window as

```
    x=pcg('PMV',b,0*b,500,1.e-6);        (5)
```

If another type of preconditioning is required, we can make another function subroutine, say `PMV2.m`, to handle this new matrix vector product.[2]

Note 1. The routines for iterative solvers of the MATLAB company (as `gmres.m` and `bicgstab.m`) take as input a matrix as well as a function subroutine that performs the MV.

Global variables

Note that, in order to be able to use the subroutine `PMV.m` of (4), we have to get the matrices `A`, `L` and `U` known to this subroutine. We do not want to do that via an input argument of the form `function c=PMV(u,A,L,U)`. Because, we then have to include these quantities also in the input list of `pcg.m`, which we were trying to avoid. The alternative of defining these quantities inside the subroutine `PMV.m` is even more undesirable, since then the quantities would be defined again and again in each iterative step of `pcg` in which the routine `PMV.m` is called (by `c=feval(MV,u);`). *Global* variables offer an efficient alternative:

```
function c=PMV(u)                                (6)
  global A L U
         c1=A*u; c2=L\c1; c=U\c2;
return
```

MATLAB (`help global`): "If several functions, all declare a particular name as GLOBAL, then they all share a single copy of that variable. Any assignment to that variable, in any function, is available to all the other functions declaring it GLOBAL". Global variables are 'known' also in other function subroutines in which they have been declared `global`, whereas *local* variables are known only inside the function subroutine in which they are used.

Note 2. To make sure that global variables do not accidentally get mixed up with local ones, global variables are usually given long complicated names, so rather `MyMATRIX` than `A`, `MyLOWERTRIANGULAR` than `L`, etc..

Note 3. If you want to have a global variables as `MyMATRIX` also available in the command window or in the workspace, then you have to declare it global in the command window (or in an M-file, not being a function subroutine, as `main.m`) as well.

Defining global variables

Of course, we have to define the global variables somewhere (that is, to give them the appropriate values). Before executing the command (5), execute a command like `PreProcess`,

```
PreProcess();
x=pcg('PMV',b,0*b,500,1.e-6);
```

where `PreProcess.m` is a function subroutine in which `A`, `L` and `U` are also declared global and where they are defined

```
function PreProcess(.....) 3                     (7)
  global A L U
         A=.....;
         L=.....;
         U=.....;
return
```

Note 4. The 'declaring' subroutine `PreProcess.m` and the 'matrix-vector subroutine' `PMV.m` go together: if you want another MV (or another preconditioner), then you have to write a new declaring subroutine, `PreProcess2.m` say, and a new MV subroutine, `PMV2.m` say. Or, is I did, use switches, in the two subroutines `PreProcess.m` and `PMV.m` to find the commands that go together. Make sure that both subroutines use the same switches.

Note 5. If you do left preconditioning, say you solve

$$\mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{A}\mathbf{x} = \widetilde{\mathbf{b}} \equiv \mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{b}$$

then you can compute $\widetilde{\mathbf{b}}$ in the 'declaring' subroutine `PreProcess.m` as well:

```
function b=PreProcess(.....,b)                   (8)
  global A L U
         A=.....;
         L=.....;
         U=.....;
         b=U\(L\b);
return
```

[2] This subroutine approach for incorporating the MV is also useful if, for instance, $\mathbf{c}$ is the solution at time T, $\mathbf{c} = \mathbf{Y}(T)$, of a high dimensional time dependent linear differential equation $\mathbf{Y}'(t) = \mathbf{H}\mathbf{Y}(t)$ with initial condition $\mathbf{Y}(0) = \mathbf{u}$. Then a subroutine that solves the differential equation defines the action of the matrix ($\mathbf{A} = \exp(T\mathbf{H})$). The matrix itself is not available but is also not needed in this approach.

[3] Here, the five dots in the input indicate that the quantities `A`, `L` and `U` may depend on parameters.