

Exploiting sparsity

AN EXPLANATION OF THE CODE FOR DAY 5 AND DAY 6

In practise, 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 than solving

$$\mathbf{Ax} = \mathbf{b}, \quad (1)$$

the *preconditioned system*

$$\mathbf{M}^{-1}\mathbf{Ax} = \mathbf{M}^{-1}\mathbf{b} \quad (2)$$

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

Now suppose we have a MATLAB subroutine, say

```
function [x,hist,t]=gcr(A,b,x0,kmax,tol);
```

that solves (1) iteratively. Here,

at the **input** side

- \mathbf{A} is the matrix \mathbf{A} ,
- \mathbf{b} is the vector \mathbf{b} ,
- \mathbf{x}_0 is the initial guess \mathbf{x}_0 (often $\mathbf{x}_0 = \mathbf{0}$, $\mathbf{x}_0 = \mathbf{0} * \mathbf{b}$ in MATLAB),
- \mathbf{kmax} is the maximum number of iteration steps that is allowed and
- \mathbf{tol} is the required reduction of the residual norm, that is, stop the iteration if $\|\mathbf{r}_k\|_2 / \|\mathbf{r}_0\|_2 < \mathbf{tol}$.

At the **output** site

- \mathbf{x} is the solution as computed by the subroutine (i.e., $\mathbf{x} = \mathbf{x}_k$ with $k = \mathbf{kmax}$ or $\|\mathbf{r}_k\|_2 / \|\mathbf{r}_0\|_2 < \mathbf{tol}$),
- \mathbf{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,
- \mathbf{t} is the time the routine needed.

To use this routine we have to form the vector $\mathbf{M}^{-1}\mathbf{b}$, which is not a problem. But, we also either

1) have to form the matrix $\mathbf{M}^{-1}\mathbf{A}$ or

2) we have to adapt the code `gcr.m` to compute the vector $\mathbf{c} = \mathbf{M}^{-1}\mathbf{Ar}$ from \mathbf{r} .

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 in memory and is often even not possible). Even if the matrix $\tilde{\mathbf{A}} \equiv \mathbf{M}^{-1}\mathbf{A}$ would be available, the computation of $\mathbf{c} = \tilde{\mathbf{A}}\mathbf{r}$ from \mathbf{r} by multiplication by $\tilde{\mathbf{A}}$ (at $2n^2$ flop) is much more expensive than performing the steps

$$\mathbf{c}' = \mathbf{Ar}, \quad \text{solve } \mathbf{Lc}'' = \mathbf{c}' \text{ for } \mathbf{c}'', \quad \text{solve } \mathbf{Uc} = \mathbf{c}'' \text{ for } \mathbf{c}, \quad (3)$$

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

2) Adapting the code `gcr.m` to perform the steps in (3) is not attractive: a) the code has to be adapted at several places (in (i) the input list `A, L, U` instead of `A`, and (ii) each time 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 (as GMRES), the code for this solver has to be adapted as well.

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

```
function [x,hist,t]=gcr(MV,b,x0,kmax,tol),
```

where now `MV` is a string of the name of this subroutine, as `MV='MyPreMV'`, and in `gcr.m` the lines with `c=A*u`; are replaced by `c=feval(MV,u)`; `MyPreMV.m` could be the function subroutine

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

(4)

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

```
x=gcr('MyPreMV',b,0*b,500,1.e-6);
```

(5)

If another type of preconditioning is required, we can make another function subroutine, say `MyPreMV2.m`, to handle this new matrix vector product. This subroutine approach for incorporating the MV is also useful if, for instance, `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 and the matrix itself ($\mathbf{A} = \exp(T\mathbf{H})$) is not available.

Note that, in order to be able to use the subroutine `MyPreMV.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=MyPreMV(u,A,L,U)`. Because, we then have to include these quantities also in the input list for `gcr.m`, which we were trying to avoid. The alternative of defining these quantities inside the subroutine `MyPreMV.m` is even more undesirable, since then the quantities would be defined again and again in each iterative step of `gcr` in which the routine `MyPreMV.m` is called (by `c=feval(MV,u)`;). *Global* variables offer an efficient alternative:

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

(6)

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. Before executing the command (5), execute a command like `MakeMatrix`,

```
MakeMatrix();
x=gcr('MyPreMV',b,0*b,500,1.e-6);
```

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

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

Note. 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`.

Note. The ‘declaring’ subroutine `MakeMatrix.m` and the ‘matrix-vector subroutine’ `MyPreMV.m` go together: if you want another MV (or another preconditioner), then you have to write a new declaring subroutine, `MakeMatrix2.m` say, and a new MV subroutine, `MyPreMV2.m` say. In order to keep things together that go together, I placed these two related subroutines in the same file. However, a second routine in a file can only be called from routines in the file and not from routines outside this file nor from the command window. I solved this obstacle by letting `MakeMatrix.m` be executed whenever `MyPreMV.m` was called by an empty argument:

```
MyPreMV([]);
x=gcr('MyPreMV',b,0*b,500,1.e-6);
```

where the file `MyPreMV.m` contains two function subroutines and looks like

```
function c=MyPreMV(u) (8)
    global MyMATRIX MyLOWER MyUPPER
    if isempty(u), MakeMatrix(); return, end
    c1=MyMATRIX*u; c2=MyLOWER\c1; c=MyUPPER\c2;
    return
function MakeMatrix()
    global MyMATRIX MyLOWER MyUPPER
    MyMATRIX=...;
    MyLOWER=...;
    MyUPPER=...;
    return
```

If `u` is empty (`u=[]`), then the global variables `MyMATRIX`, ..., are created. If `u` is a non-empty vector (as it will be when called by `gcr.m`), then the ‘`MakeMatrix` line’ is skipped in `MyPreMV.m` and the the preconditioned MV is performed.

Note. If the matrix is not available, but only its action via a subroutine as `MyPreMV.m`, then MATLAB's `size` can not be applied to `A` to find the dimension (it can be applied to `b`: `n=size(b,1);`). With `n=MakeMatrix();` and `n=MyPreMV([]);`, my routines return the dimension.

Note. If you want to have a global variables as `MyMATRIX` also available in the command window or 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.

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

The folder structure (Day 6).

The folder `Problems` contains the files that define the matrix (with a routine of the type `MakeMatrix` as discussed above) and the matrix-vector routines (of the type `MyPreMV`. The routines `MakeMatrix` and `MyPreMV` will be together in one file `MyPreMV.m`). A call as `[n,Lambda]=MyPreMV([]);` will also return the dimension `n` of the matrix and a sequence (row vector) of eigenvalues of `A` (if defined in `MyPreMV.m`). Some routines return the right hand side vector `b`: `[n,b]=MyPreMV([])` (consult the code). This may be a convenient way to precondition the vector `b`, if `b` has to be preconditioned to $M^{-1}b$ before passing it to a solver as `gcr`.

The folder `Solvers` contains the files with codes of iterative solver as LMR, Richardson (`polynomial solver.m`), GCR (`gcr.m`), etc.

The folder `Subroutines` contains miscellaneous subroutines (for orthogonalisation — Gram-Schmidt variants—, etc.).

Before running `main`, run `install`. This subroutine sets the path for MATLAB (tells MATLAB also to 'look' in the folders `Problems`, `Solvers`, `Subroutines` for subroutines).

Example. In the first assignment of Day 5 you are asked to explore the significant of exploiting sparsity. You are asked to experiment with a diagonal matrix in several formats. You can use the routine `matrix1.m` as provided in Day 5's matlab package. You can also write your own brand. For instance, you can create files `MakeDiagonal.m` and `MyDiagMV.m` like

```
function n=MakeDiagonal()
    global MyDIAGONAL
        n=10000;
        MyDIAGONAL=sqrt(1:n)';
return
```

and

```
function c=MyDiagMV(u)
    global MyDIAGONAL
        c=MyDIAGONAL.*u;
return
```

and in the command window you can call these routines like

```
n=MakeDiagonal();  
xe=ones(n,1); b=MyDiagMV(xe);  
tic, x=polynomialssolver('MyDiagMV',b,0*b,500,1.e-6,2/(1+n)); toc
```

The routine `polynomialssolver.m` multiplies the residual in each step by $(\mathbf{I} - \alpha \mathbf{A})$ with \mathbf{A} as, in this case, defined by `MyDiagMV.m` and $\alpha = 2/(1+n)$ (Note that this puts $1/\alpha$ in the center of the spectrum). It stops the iteration when the residual is reduced by $1.e-6$ or when more than 500 steps were required. `x` is the approximate solution at termination.

The choice `b=MyDiagMV(xe)` makes sure that the exact solution is known, which may be convenient in an experimental stage, since it gives access to the error (`norm(x-xe,2)`). With `x=polynomialssolver('MyDiagMV',b,0*b,500,1.e-6)`; α is computed in each step, to minimize the norm of the new residual (then `polynomialssolver` is LMR). The `tic toc` commands give the time that MATLAB spent in between the tic-toc. `clock` and `etime` can do this as well. With

```
profile on, x=polynomialssolver(...); profile viewer
```

MATLAB gives a very detailed report on timings.