Exploiting sparsity; An explanation of the code for Lecture 5 and 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 then solving

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{5.1}$$

the *preconditioned system*

$$\mathbf{M}^{-1}\mathbf{A}\mathbf{x} = \mathbf{M}^{-1}\mathbf{b} \tag{5.2}$$

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 $\mathbf{L}\mathbf{u}' = \mathbf{r}$ and $\mathbf{U}\mathbf{u} = \mathbf{u}'$, the system $\mathbf{M}\mathbf{u} = \mathbf{r}$ can efficiently be solved for $\mathbf{u}$.

Now suppose we have a (file with the) MATLAB subroutine, say

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

that solves (5.1) iteratively. 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 $\|\mathbf{r}_k\|_2/\|\mathbf{r}_0\|_2 <$`tol`.

At the **output** site

- `x` is the solution as computed by the subroutine at termination
  (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,
- `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 have
1) to form the matrix $\mathbf{M}^{-1}\mathbf{A}$ or
2) to adapt the code `gcr.m` to compute the vector $\mathbf{c} = \mathbf{M}^{-1}\mathbf{A}\mathbf{r}$ from $\mathbf{r}$.

Discussion.
1) To focus the discussion, assume that $\mathbf{M} = \mathbf{L}\mathbf{D}^{-1}\mathbf{U}$ with $\mathbf{D}$ diagonal, and $\mathbf{L}$ and $\mathbf{U}$ sparse triangular matrices that are explicitly available. Generally, the matrices $\mathbf{L}^{-1}$, $\mathbf{U}^{-1}$ will be full (check that $\mathbf{L}^{-1}$ is a full matrix if, for instance, $\mathbf{L}$ has all ones on the main diagonal and on the first lower co-diagonal and zeros elsewhere). Therefore, the matrix $\mathbf{M}^{-1}\mathbf{A}$ will generally be full as well. 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). And 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 obtaining $\mathbf{c}$ by performing the following four steps

$$\mathbf{c}' = \mathbf{A}\mathbf{r}, \quad \texttt{solve } \mathbf{L}\mathbf{c}'' = \mathbf{c}' \text{ for } \mathbf{c}'', \quad \mathbf{c}''' = \mathbf{D}\mathbf{c}'', \quad \texttt{solve } \mathbf{U}\mathbf{c} = \mathbf{c}''' \text{ for } \mathbf{c}, \tag{5.3}$$

(at $2\,kn$ flop with k the sum of the maximum number of non-zeros in each row of $\mathbf{L}$, $\mathbf{U}$ and $\mathbf{A}$ plus 1. If the matrices are sparse k will be $\ll n$).

2) Adapting the code `gcr.m` to perform the steps in (5.3) is not attractive for several reasons:
a) the code has to be adapted at several places (to be explicit: (i) in the input list of `gcr`, '`A`' has to be extended to '`A,L,D,U`', and (ii) whenever the command `c=A*u` is used in `gcr`, it has to be replaced by a coding of the steps in (5.3)),
b) other types of preconditioning would require similar but new adaptations, and
c) if you want to exploit other solvers (as GMRES), the code for these solvers has to be adapted as well.
Of course, each adaptation caries the danger of introducing errors in the code.

Defining the action of a matrix by means of a function

An efficient way out is to call a function subroutine, say `MyPreMV`, in the `gcr` code whenever a matrix-vector multiplication is required. For instance,

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

where now `MV` is a handle of this function, as `MV=@MyPreMV`,[1] and, at each occurrence, in the `gcr` code the command `c=A*u;` is to be replaced by `c=feval(MV,u);`. For instance, `MyPreMV.m` could be (a file of) the function subroutine

```
function c=MyPreMV(u)                                    (5.4)
        c1=A*u; c2=L\c1; c3=D*c2; c=U\c3;
return
```

We can make the routine `MyPreMV` as efficient as possible without fiddling with the routine `gcr` (for instance, we can allow old 'c-values' to be replaced by new ones as

```
function c=MyPreMV(u)                                    (5.5)
        c=A*u; c=L\c; c=D*c; c=U\c;
return
```

thus saving memory). Now, we can call `gcr` in the command window, by first executing the command `MV=@MyPreMV;`, followed by

```
x=gcr(MV,b,0*b,500,1.e-6);                               (5.6)
```

If another type of preconditioning is required, we can make another function subroutine, say `MyPreMV2`, to handle this new matrix vector product. Then, redefining `MV` to `MV=@MyPreMV2;` allows us to use the command (5.6) again.

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 $\mathbf{Y}(0) = \mathbf{u}$: in this case, $\mathbf{c}$ depends linearly on $\mathbf{u}$. Then a subroutine that solves the differential equation defines the action of the matrix ($\mathbf{A} = \exp(T\mathbf{H})$) while the matrix itself is not available.

[1] For information, search MATLAB's documentation for 'function handle'. MATLAB allows calling a function inside another function by handle as well as by name. To be more specific, we could also take `MV='MyPreMV'`. Here, we focus on function handles for passing functions, rather than on strings of function names. The approach with function handles is more elegant in MATLAB. The approach with function names will require the use of global variables as we will explain in the subsection below on global variables.

Incorporating the action of the matrix in a function

Note that, in order to be able to use the subroutine `MyPreMV` of (5.4), we have to get the matrices `A`, `L` and `U` 'known' to this subroutine. We do not want to do that by explicitly passing these quantities, that is, via an input argument of the form

```
function c=MyPreMV(u,A,L,D,U).
```
(5.7)

Because, we then have to include these quantities also in the input list of `gcr`, which we were trying to avoid. The alternative of defining these quantities inside the subroutine `MyPreMV` 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` is called (by `c=feval(MV,u);`, here `MV=@MyPreMV`). This is where function handles can be conveniently exploited.[2] In a function, say, `DefineMatrixAction`, we can define the quantities as $\mathbf{L}$, $\mathbf{D}$ and $\mathbf{U}$, depending on a given matrix $\mathbf{A}$, as well as the function `MyPreMV`. By means of a function handle we can make this function `MyPreMV` available outside `DefineMatrixAction`. In the following, as an example, we take $\mathbf{L}$ to be the lower triangular part of $\mathbf{A}$, $\mathbf{U}$ the upper triangular part, both including the diagonal of $\mathbf{A}$, and $\mathbf{D}$ the diagonal of $\mathbf{A}$.

```
function [MV,bt]=DefineMatrixAction(A,b)
    L=tril(A); U=triu(A); D=diag(diag(A));

    function c=MyPreMV(u)
       c=A*u; c=L\c; c=D*c; c=U\c;
    end

    MV=@MyPreMV;
    bt=L\b; bt=D*bt; bt=U\bt;
end
```
(5.8)

This function `DefineMatrixAction` defines the preconditioned matrix-vector multiplication `MyPreMV`. Via the list of ouput arguments, the function handle in `MV=@MyPreMV;`, allows to make the preconditioned matrix-vector multiplication `MyPreMV` available outside `DefineMatrixAction`. Note that `DefineMatrixAction` also computes the preconditioned right-hand side vector $\widetilde{\mathbf{b}} \equiv \mathbf{U}^{-1}\mathbf{D}\mathbf{L}^{-1}\mathbf{b}$.

Since the function `MyPreMV` is defined inside the function `DefineMatrixAction`, the quantities as `L` and `U` that are 'known' inside `DefineMatrixAction` are also 'known' to `MyPreMV`. In particular, the undesirable explicit passing of quantities as `L` to `MyPreMV` (as in (5.7)) is not required now.[3]

Before executing the command (5.6) in MATLAB's command window, execute a command as `DefineMatrixAction`:

```
[MV,bt]=DefineMatrixAction(A,b);
x=gcr(MV,bt,0*b,500,1.e-6);
```

[2] For an alternative approach, using global variables, see the next subsection.

[3] If you are familiar to C++ programming, then the observation that the function handle `@MyPreMV` actually is a *pointer* to the function `MyPreMV` might be illuminating.

Matrix actions using global variables

In this subsection, we suggest the use of *global variables* as an alternative to the use of function handles. If you are comfortable with function handles, you can skip this subsection.

If the matrix vector multiplications $\mathbf{c} = \mathbf{A}\mathbf{u}$ have been defined in `gcr` as `c=feval(MV,u)`, then, as an alternative to function handles, as `MV=@MyPreMV`, the name of a function can be used: with `MyPreMV` as in (5.4), command (5.6) with `MV=@MyPreMV` or with `MV='MyPreMV'` leads to the same result.

However, calling a function by name rather than by handle, makes it more tricky to get the quantities `A`, `L`, ... 'known' to `MyPreMV` without explicitly passing them (see the discussion in the preceding subsection).

The problem is that, if `MyPreMV` has been defined as in (5.8), then `MyPreMV` is a local function and the string `'MyPreMV'` is recognised as the name of a function only inside the function `DefineMatrixAction`. Defining `MV='MyPreMV'` inside `DefineMatrixAction` (instead of `MV=@MyPreMV` as in (5.8)) with `MV` in the output list turns `MV` outside `DefineMatrixAction` into the string `'MyPreMV'` but the string has now further meaning (it will not be recognised as the name of a function). In summary, in contrast to function handles, *strings* as `'MyPreMV'` are recognised as the name of a function only if the function `MyPreMV` is not defined inside another function (i.e., the file `MyPreMV.m` should only consist of the function `MyPreMV`): `MyPreMV` should be a *global* function.

If `MyPreMV` is a global function, *global* variables can be used to get the quantities `A`, `L`, ... 'known' to `MyPreMV` without explicitly passing them:

```
function c=MyPreMV(u)                                          (5.9)
  global A L D U
        c1=A*u; c2=L\c1; c3=D*c2; c=U\c3;
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 the other type of variables, so-called *local* variables, are known only inside the function subroutine in which they are used. Before executing the command (5.6) in the commandwindowexecute a command like `MakeMatrix`,

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

where `MakeMatrix` is a function subroutine in which `A`, `L`, `D`, and `U` are defined and declared to be global. For instance,

```
function MakeMatrix(A)                                         (5.10)
  global A L D U
        L=tril(A); U=triu(A); D=diag(diag(A));
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` and the 'matrix-vector subroutine' `MyPreMV` go together: if you want another matrix-vector multiplication (MV) (or another preconditioner), then you have to write a new declaring subroutine, `MakeMatrix2` say, and a new MV subroutine, `MyPreMV2` say. In order to keep things together that go together, I placed these two related subroutines in the same file as `MyPReMV.m`. 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` be executed whenever `MyPreMV` 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)
   global MyMATRIX MyLOWER MyDIAGONAL MyUPPER
        if isempty(u), MakeMatrix(); c=size(MyMATRIX,1); return, end
        c=MyMATRIX*u; c=MyLOWER\c; c=MyDIAGONAL*c; c=MyUPPER\c;
return
function MakeMatrix()
   global MyMATRIX MyLOWER MyDIAGONAL MyUPPER
        MyMATRIX=...; MyLOWER=tril(MyMATRIX);
        MyDIAGONAL=diag(diag(MyMATRIX); MyUPPER=triu(MyMATRIX);
return
```

(5.11)

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` and the the preconditioned MV is performed.

Note. If the matrix is not available, but only its action via a subroutine as `MyPreMV`, then MATLAB's command `size` can not be applied to `A` to find the dimension. With `n=MyPreMV([]);`, tha above routine returns 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 (either known by its name or by its handle) that performes the MV.

The folder structure (Code with Lecture 6).

The folder `Problems` contains the files, as `problem0.m`, that define the matrix $\mathbf{A}$, say, and the function routine that performs the matrix-vector multiplication (MV), $\mathbf{c} = \mathbf{A}\mathbf{u}$,

(of the type `MyPreMV`). This MV is passed as first argument in the output list by a function handle as

`function [MV,b]=problem0(n).`

The second argument in the output list contains the right-hand side vector **b**. The `n` in the input of this example function `problem0` specifies the required dimension of the matrix. Some problems functions can provide a third output argument `Lambda`. This is a vector of all eigenvalues of the matrix.

`problem1b.m` relies on the use of global variables, as explained in the preceding subsection. The other routines use function handles.

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

The folder `Subroutines` contains miscellanuous 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 Lecture 5 you are asked to explore the significants of exploiting sparsity. You are asked to experiment with a diagonal matrix in several formats. You can use the routine `problem1.m` as provided in Lecture 5's MATLAB package. You can also write your own brand. For instance, you can create a file `MakeDiagonal.m` with a routine like

```
function [MV,b]=MakeDiagonal()
    n=10000;
    MyDIAGONAL=sqrt(1:n)';

    function c=MyDiagMV(u)
       c=MyDIAGONAL.*u;
    end

    MV=@MyDiagMV;
    xe=ones(n,1); b=MyDiagMV(xe);
end
```

and in the command window you can call these routines like

```
[MV,b]=MakeDiagonal();
tic, x=polynomialsolver(MV,b,0*b,500,1.e-6,(1+n)/2); toc
```

The routine `polynomialsolver.m` multiplies the residual in each step by $(\mathbf{I}-\frac{1}{\mu}\mathbf{A})$ with $\mathbf{A}$ as, in this case, defined by `MyDiagMV.m` and $\mu =$`(1+n)/2` (Note that this puts μ 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=polynomialsolver('MyDiagMV',b,0*b,500,1.e-6);` $\alpha = 1/\mu$ is computed in each step, to minimize the norm of the new residual (then `polynomialsolver` 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=polynomialsolver(...); profile viewer
```

MATLAB gives a very detailed report on timings.