CS 6220 – HW5/Problem2

by Ivo Boyadzhiev (iib2)

I picked **5 tall and thin** (rectangular) matrices from the UF data set. Then, I selected only the **first 100 columns**, which happened to be enough for these concrete matrices to have a full-column basis (in Matlab's working precision).

1. Using **svds** in the mode of searching for the smallest singular value did not work, because of the intermediate representation that this algorithms uses internally. It actually creates the same matrix as in case 3: $C = [0\ A;\ A^T\ 0]$, which always has 0 as an eigenvalue, because of the fact that A is rectangular ($n > m$), which makes C rank deficient ($n + m > 2 * m$).

2. Searching for the smallest eigenvalue of $B = A^TA$ worked much better, as this matrix (B) has full rank (this comes from the fact that A has full column rank).

   Running **eigs**(B) is very efficient, as it boils down to a bunch of matrix-vector multiplications of the kind (A' * (A * x)), which is very fast operation when A is sparse. We can run **eigs**(B) in the mode of searching for the "Smallest Algebraic" eigenvalue, because B, as defined above, is symmetric positive definite matrix. The implication of this is that all its eigenvalues are positive, so the smallest in magnitude will be the same as the smallest algebraic.

3. As **svds** uses this representation internally, we have the same situation as in 1. I want to notice here that we can use this form explicitly to gain efficiency if we have a structure in A that we

can exploit by providing our own routine for solving linear equations of the form (A – sigma * I) * x = b.

As case 1 and 3 are basically the same (in our case we don't exploit structure in A, as they are randomly selected from the UF data set), I will explain my idea for only one of them.

We know that **eigs** can go after internal eigenvalues, if we provide an initial guess (through the **sigma** parameter). Therefore, my idea is to step through the possible values starting from the largest one, which we can easily find (applying the power method for example). As we are interested in the smallest positive eigenvalue, within a factor of 10 of the true one, we can efficiently search the space by exponentially decreasing the region of interest (until we reach sufficiently close to 0 value).

The result of my experiments are summarized in the following table (all matrices are clamped to 100 columns and time is given in seconds):

		Matrix (903)	Matrix (1881)	Matrix (2031)	Matrix (2186)	Matrix (2218)
		Rows (71952)	Rows (1977885)	Rows (11760)	Rows (12360060)	Rows (1748122)
Case1	Time	1.34	22.43	0.10	220.89	9.67
	Result	0.009	0.12	6.33	0.24	6.50
Case2	**Time**	**0.48**	**0.3**	**0.009**	**6.87**	**0.17**
	Result	**0.009**	**0.12**	**4.90**	**0.24**	**2.94**
Case3	Time	0.96	18.14	0.08	179.19	8.09
	Result	0.009	0.12	6.33	0.24	6.50

Conclusions:

We can see that case 2 has the fastest timing, and all other produced results that are within a factor of 10 from the true value (case 2), but with higher timing. Though case 1 and 3 produced the same results, **svds** seems slower than mine custom implementation of the same idea, so they might be doing more work than needed there. I want to notice that if we have a good initial estimation of the smallest eigenvalue, case 1 and 3 can be as efficient (we don't have to iteratively search for the shift). Though in case 2 we only invoke matrix-vector multiplications (regarding A), where on the other hand for cases 1 and 3 we need to solve linear systems involving A – sigma * I.

There are situations where case 1 (or 3) might be preferable, and those are the cases where the smallest and the second smallest eigenvalue of A^TA are very close to each other, which will make the convergence rate of case 2 very slow. Another problem of going through A^TA is that it can "destroy" info for the small eigenvalues, but I did not observe that in my tests.

The code of the script follows:

```
function [] = ShowSigmaMinMethods()
numSamples = 5;

% Define the searching criteria
minNumRows = 3000; % We want big sparse matrices
rowsColsRatio = 0.1; % We want tall and thin matrices
maxNumCols = 100;
ratioTol = 10; % The allowed tollerance

index = UFget; % get index of the UF Sparse Matrix Collection

% Find all SPD matrices, that meet the defined criterias
```

```
allIds = find (index.nrows > minNumRows & ...
    index.ncols  <= index.nrows * rowsColsRatio);

% Select the 1st numSamples of them
spdIndices = allIds(1:numSamples);

for i=spdIndices
    Problem = UFget(i);

    fprintf('**************** New Matrix (%d) ****************', i);

    % Get the matrix, which will be in sparse format..
    A = Problem.A;

    % Get the first few columns of A, so that we will increase the
    % chance of having full-column rank matrix
    A = A(:, 1:maxNumCols);

    [n, m] = size(A)

    %fprintf('Size of A(%d) = [%d %d], nnz(A) = %d\n', i, n, m, nnz(A));

    % 1st: Compute smallest eval of the SPD matrix A'*A
    EIGS_OPTS.issym = true;
    EIGS_OPTS.isreal = true;
    EIGS_OPTS.tol = 1e-10;

    tic
    s1 = eigs(@(x) (A' * (A * x)), m, 1, 'SA', EIGS_OPTS);
    s1 = sqrt(s1)
    toc

     % 2nd: Compute smallest singular value, using SVDS

    % We will use the largest eval (which is an easier problem),
    % as an initial starting value, during the iteration search
    s_max = svds(A, 1);

    tic
    s2 = EstimateSmallestEvalUsingSvds(A, s_max, ratioTol)
    toc

    abs(s2 - s1) / s1

    % 3rd: Compute the smallest positive eval of [0 A; A' 0]

    B = [sparse(n, n), A; A', sparse(m, m)];

    tic
    s3 = EstimateSmallestEvalUsingB(B, s_max, ratioTol)
    toc

    abs(s3 - s1) / s1
end
end
```

```
function [sigma] = EstimateSmallestEvalUsingSvds(A, b, ratio_tol)
    SVDS_OPTS.tol = 1e-10;

    % With step factor of ratio_tol, search for the smallest positive
    % singular value of A
    curr_sigma = b;
    while curr_sigma > SVDS_OPTS.tol
        sigma = curr_sigma;
        curr_sigma = svds(A, 1, b, SVDS_OPTS);

        b = b / ratio_tol;
    end
end

% Estimates the smallest positive eval using SVDS + shifts and exponentially
% decreasing search on the shift parameter.
function [sigma] = EstimateSmallestEvalUsingB(B, b, ratio_tol)
    EIGS_OPTS.issym = true;
    EIGS_OPTS.isreal = true;
    EIGS_OPTS.tol = 1e-10;

    % With step factor of ratio_tol, search for the smallest positive
    % singular value of A
    curr_sigma = b;
    while curr_sigma > EIGS_OPTS.tol
        sigma = curr_sigma;
        curr_sigma = eigs(B, 1, b, EIGS_OPTS);

        b = b / ratio_tol;
    end
end
```