Lab class (large-scale linear systems)

1 Testing methods for sparse linear systems

Katz centrality

Let $A \in \mathbb{R}^{n\times n}$ be the adjacency matrix of an undirected graph, i.e.,

$$A_{ij} = \begin{cases} 1 & \text{if there is an edge between } i \text{ and } j, \\ 0 & \text{otherwise,} \end{cases}$$

and $e \in \mathbb{R}^n$ be the column vector of all ones, $e_i = 1$ for $i = 1, 2, \ldots, n$.

Let moreover $\alpha \in \mathbb{R}$ be such that $\alpha > 0$ and $|\lambda|\alpha < 1$ for each eigenvalue λ of A, and set $x = (I - \alpha A)^{-1}e$. We have argued during the lectures that x_i provides a measure of centrality of the node i of the graph; it is higher the more node i is 'well-connected'.

We are interested in comparing the efficiency of various numerical methods of computing this vector of centrality indices x. Since the matrix $M = (I - \alpha A)$ is symmetric and positive definite with our choice of α, the more natural competitors are:

- The dense Cholesky factorization $A = LL^*$, with L lower triangular.
- Its analogue for sparse matrices, using heuristics to increase the sparsity of the factor L.
- The conjugate gradient method (CG), which is a Krylov subspace method.

In the next sections you will just need to follow along using Matlab's predefined library functions, not code the algorithms yourself.

If something confuses you about the Matlab syntax or you don't know how to do something, feel free to ask me.

Dense Cholesky

1. Download the text file `karate_ascii.mat` from the course web page. This is a 34×34 matrix coming from a famous paper in network theory from the 1970s. Load it in Matlab with `A = load('karate_ascii.mat', '-ascii')`. You may inspect the resulting matrix by typing `A`, or with `spy(A)`.

   If your Matlab version is recent enough, you can also display the graph with `plot(graph(A))`,

2. Check using the `eig(A)` command that $\alpha = 0.1$ satisfies the condition that we required above on α.

3. Compute the Cholesky factor with the command `R = chol(M)`. This function returns the upper triangular matrix $R = L^*$ for which $M = R^*R$. Check that this equality holds approximately by computing in Matlab $\|M - R^*R\|$.

4. Look at the sparsity pattern of R with `spy(R)`; then and use the command `nnz(R)` to count how many nonzero entries it has. Are they more or less than the nonzero entries of M?

5. Solve the system $Mx = e$ using the Cholesky factor that we have computed (how?). Recall that Matlab's operator `Matrix \ vector` can be used to solve linear systems, and that it requires time $O(n^2)$ if the matrix is triangular.

6. Check the residual $\|M\tilde{x} - e\|$ for the solution $\tilde{x}$ returned by Matlab. It should be of the order of 10^{-15}; you can check that the condition number `cond(M)` is very small (about 4.4).

Sparse Cholesky

1. Now we shall convert A to a sparse matrix object with the command `A = sparse(A)`. Note that Matlab outputs sparse matrices by default with a different notation,

   ```
   (2,1) 1
   (3,1) 1
   (4,1) 1
   ...
   ```

   We also need to recreate M as a sparse matrix object, with

   ```
   M = speye(size(A)) - 0.1*A;
   ```

   You can check that the commands that you gave earlier, such as `chol(M)`, still work with sparse matrix objects, and return essentially the same matrices but as sparse matrix objects. Since the matrix is small, the difference in time between the various algorithms is not so relevant.

2. The function `p = symrcm(M)` returns a permutation of the vertices that has the aim to reduce the number of zeros generated by Cholesky factorization — it is the order returned by a sort of breadth-first visit of the graph (reverse Cuthill-McKee order). Check that `Rp = chol(M(p, p))` has fewer nonzeros than `R`, and that the system `M(p,p)*y=e(p)` has solution `y=x(p)`.

   (Note that permuting the entries of e with `e(p)` would not be necessary here, because they are all ones anyway.)

Conjugate gradient

1. Solve the system $Mx = e$ using conjugate gradient, stopping when the residual goes below 10^{-6}, with the command `x = pcg(M, e, 1e-6)`. Check that the relative residual $\frac{\|M\tilde{x}-e\|}{\|e\|}$ of the solution $\tilde{x}$ is indeed below 10^{-6}.

2. Calling the function with more return values, one can get a vector `resvec` containing the residuals obtained at each iteration of CG:

   ```
   [x, useless1, useless2, useless3, resvec] = pcg(M, e);
   ```

   We can plot this residual vector using a logarithmic scale on the y axis using `semilogy(resvec)`. The method converges quite fast; indeed, this is a very favorable matrix for Krylov subspace methods, because the eigenvalues of M contain 1 with multiplicity 10, and the other eigenvalues are not too far from 1. (Check this with `eig(M)`).

Scaling up

Now it's time to try examples with larger n. Let us generate a random symmetric sparse matrix of size $n = 4000$ with density 0.001 (i.e., only one out of 1000 entries is nonzero, or about 4 per row).

```
A = sprandsym(4000, 0.001); M = speye(length(A)) - 0.1*A;
```

This command already returns A in the form of a sparse matrix object. If you want the full array, use `full(A)`.

Note that $\alpha = 0.1$ might not be small enough to guarantee $|\lambda|\alpha < 1$. It is not a good idea to use `eig(A)` to check, because it would take a while on a 4000×4000 matrix. There is a function `eigs(A)` that returns approximations to the largest (in modulus) eigenvalues of A, instead.

(We shall see later in this course which numerical methods are available to compute eigenvalues.)

1. Test the algorithms we have seen above (dense Cholesky, sparse Cholesky, sparse Cholesky with `symrcm` reordering, CG) on this new matrix. You can compare timings with the functions `tic()` (starts a timer) and `toc()` (stops it and returns the result); for instance,

   ```
   tic(); R = chol(M); x = R \ (R' \ e); time_elapsed = toc()
   ```

   Who is the winner?

2. You can try using `x = inv(M)*e` as well. Is it as fast as the other methods?

3. We have chosen a family of matrices that is particularly favorable for Krylov subspace methods. At home, you can try generating matrices with the command

```
n = 4000;
k = 5;
A = spdiags(rand(n, 2*k+1), -k:k, n, n);
A = (A+A')/2;
```

This creates a matrix with nonzero entries only in a 'band' of 5 diagonals above and below the main diagonal. On these matrices, sparse factorization methods work at their best.

4. You can also try larger matrices. Dense methods are already at their limit with $n = 4000$; sparse direct methods and Krylov subspace methods can scale to larger sizes (with their performance depending on the zero pattern and eigenvalue location of M).

2 Coding the Arnoldi iteration

We want to write a `function [Q, Hhat] = arnoldi(A, b, m)` that runs m steps of the Arnoldi process and returns two matrices $Q = Q_{m+1} \in \mathbb{C}^{n\times(m+1)}$ with orthonormal columns (i.e., $Q^*Q = I_{m+1}$) and $\hat{H} = \hat{H}_m \in \mathbb{C}^{(m+1)\times m}$ such that $AQ_m = Q\hat{H}$. Recall that we called $Q_m \in \mathbb{C}^{n\times m}$ the matrix obtained by removing the last column of Q_{m+1}, and $H_m \in \mathbb{C}^{m\times m}$ the matrix obtained by removing the last row of $\hat{H}_m$.

Recall that the pseudocode for the Arnoldi iteration is

$\hat{H} = zeros(m+1, m)$;
$q^1 = b/\|b\|$;
for $k = 1, 2, \ldots, m$ **do**
  $r = Aq^k$;
  for $j = 1, 2, \ldots, k$ **do**
    $\hat{H}(j,k) = (q^j)^* r$;
    $r = r - q^j \hat{H}(j,k)$;
  end
  $\hat{H}(k+1,k) = \|r\|$;
  $q^{k+1} = r/\hat{H}(k+1,k)$;
end
$Q = \begin{bmatrix} q^1 & q^2 & \cdots & q^{m+1} \end{bmatrix}$;

For your convenience, in this pseudocode the rows and columns of Q and H are indexed starting from 1 — which is what you need in Matlab.

1. Choose a matrix A and a right-hand side b; for instance, use as A the 34×34 matrix that we used in the previous exercise, and $b = e$.

2. Compute the first couple of iterations directly at the command prompt, just to experiment with the method: for instance, write

```
q1 = b / norm(b);
r = A*q1;
```

```
r = r - (q1'*r)*q1;
q2 = r / norm(r);
```

and then check that `q1` and `q2` are orthogonal (up to machine precision).

3. Write the full procedure, and check that the columns of `Q` are orthonormal, and that `norm(A*Q(1:end, 1:end-1) - Q*Hhat)` is of the order of machine precision.

4. Check that the matrix $\hat{H}$ has the zeros below the first subdiagonal, as it should. Actually, for the matrix A that we have suggested above, $\hat{H}$ should be tridiagonal: we have said during the lecture that this is what happens when A is symmetric. Is it the case?

5. Check that $Q^* e = \|e\| e_1$ (at least up to machine precision).

6. Now let us use the Arnoldi factorization that we have constructed to solve a linear system. Set $M = I - 0.1A$, as above, and compute Q_9 and $\hat{H}_8$ with `arnoldi(M, e, 8)`. Now compute an approximation of the solution to $Mx = e$ as
$$\tilde{x} = Q_8 H_8^{-1} (Q_8)^* e = Q_8 H_8^{-1} \|e\| e_1.$$
Is the residual $M\tilde{x} - e$ small? Is $\tilde{x}$ close to the solution that you computed earlier with the other algorithms?

3 If you still have time

Additional activities if you still have time:

- Like we did with Arnoldi, compute the first couple of iterations of conjugate gradient directly from the command prompt. Check that the choices of β_k and t_k given in the lectures produce the orthogonality properties that we have claimed, i.e., $(r^j)^* r^k = (d^j)^* A d^k = 0$ for each $j \neq k$.

- Theoretical exercise: let A be a Hessenberg matrix, and $b = e_1$. What is the basis Q produced by the Arnoldi process?