Project Notes

Hugh Delaney

July 2, 2021

1 Introduction

We want to compute $f(A)x$ for $f(A) = e^A$, where $A \subseteq \mathbb{R}^{n\times n}$ and n is large. We want to approximate this by using the Krylov subspace $\mathcal{K}_n(A, x) = \text{span}\{x, Ax, \ldots, A^{m-1}x\}$ which has rank $m \ll n$.

We are using an impicit Schur decomposition by Lanczos algorithm. This is used since A is Hermitian (as G is an undirected graph), so the reduction to Hessenberg form (by Arnoldi/Lanczos) yields a hermitian upper Hessenberg matrix, which is tridiagonal.

Note that we do not explicitly form the matrix $f(A)$ but rather apply it to some starting vector, which is normalized to become the first vector in our Krylov subspace.

In abstract terms:

$A = QTQ*$	Schur Decomposition of A using Lanczos, where $q_1 = x/\|x\|$
$A = Q(V\Lambda V^*)Q^*$	Eigendecomposition of T
$f(A) = QVf(\Lambda)V^*Q^*$	Applying f
$f(A)x = QVf(\Lambda)V^*Q^*x$	Allowing $f(A)$ to act on x
$f(A)x = QVf(\Lambda)V^*\|x\|e_1$	$Q^*v = \|v\|e_1$ (since x is normalized to give q_1 and Q is unitary)

Here $Q \subseteq \mathbb{R}^{n\times m}$ is the orthonormal basis for our Krylov subspace $\mathcal{K}_n(A, v)$. $V \subseteq \mathbb{R}^{m\times m}$ is the eigenvector matrix, which is easily calculated since T is tridiagonal and small relative to A. Note that since A is symmetric, $V^* = V^{-1}$, as long as the eigenvectors have been normalized.

Which leads us to our approximation:

$$f(A)x \approx \|x\|QVf(\Lambda)V^*e_1 \tag{1}$$

2 Serial Implementation

In order to implement (1), we must.

- Read in/generate graph. The graph is understood to be a sparse graph, and so we must read in/generate the graph into a sparse matrix A.
- Compute Lanczos with respect to a starting vector x and some dimension n for $\mathcal{K}_n$. Store Q as a full matrix (not in sparse format).
- Perform an eigendecomposition of the tridiagonal T. This can be done using LAPACK's SSTEVD routine.
- Apply f to the eigenvalues of T (also known as the Ritz values).
- Perform some matrix multiplication to compute $\|x\|QVf(\Lambda)V^*e_1$. Note that $V^*e_1 = v_{1*}$, the first row of V. Since $f(\Lambda)$ is diagonal we can treat it like a vector, so the computation becomes $\|x\|QV(f(\Lambda).*v_{1*})$, where $.*$ denotes elementwise multiplication of vectors. So we need to multiply tall, skinny Q by small, square V, and then multiply tall, skinny QV by the vector $(f(\Lambda).*v_{1*})$. Then scale the result by $\|x\|$.

3 Data Structures

3.1 Adjacency Matrix

By virtue of the fact that a_{ij} can only take on values 0 or 1, we can store the adjacency matrix as two arrays: `unsigned * row, * col`, both of which have dimension `edge_count`. Where there is an edge between nodes `row[i]` and `col[i]` for all `i`. We do not need to store the value at index `(row[i], col[i])` since all of the nonzero entries in our adjacency matrix have value 1. Since we are dealing with an undirected graph we do not need to store the bottom triangular part of the matrix since it is the same as the top (`row[i] < col[i]` for all `i`).

```
struct adjMatrix {
      unsigned* row_idx; // dimension edge_count
      unsigned* col_idx; // dimension edge_count
      const unsigned edge_count;
      const unsigned n;
};
```

3.2 Lanczos Decomp

It might be nice to be able to keep `alpha, beta` and `Q` together just for neatness.

```
struct lanczosDecomp {
      double* alpha;   // dimension krylov_dim    -- diagonal of T
      double* beta;    // dimension krylov_dim-1  -- subdiagonal of T
      double* Q;       // dimension krylov_dim*n  -- orthonormal basis for Krylov subspace
      double* x;       // dimension n             -- starting vector for Krylov subspace
      const unsigned krylov_dim;
      const unsigned n;
};
```

3.3 Eigendecomposition

Potentially do the same for eigendecomposition:

```
struct eigenDecomp {
      double* V;        // dimension krylov_dim*krylov_dim -- the eigenvectors of T
      double* lambdas;  // dimension krylov_dim            -- the eigenvalues of T
};
```

Note that we don't need V^{-1} since $V^{-1} = V^*$, moreover our algorithm only requires the first column of V^{-1}, which is the first row of V.

4 Functions needed

4.1 Read Sparse Matrix from File

```
void readGraph(adjMatrix A, FILE * file);
```

4.2 Lanczos

```
void Lanczos(adjMatrix A, lanczosDecomp D);
```

To populate `D`'s arrays. Note that `Q` must be initialized before calling `Lanczos` with the input vector x normalized as the first column of `Q`.

Note that the algorithm proceeds as follows:

```
function [alpha, beta,Q] = Lanczos(A, v, k)

Q(:, 1) = v/norm(v);
for j=1:k
    v = A*Q(:,j);
    alpha(j) = Q(:,j)'*v;
    v = v-alpha(j)*Q(:,j);
    if j > 1
        v = v-beta(j-1)*Q(:,j-1);
```

```
    end
    if j < k
        beta(j) = norm(v);
        Q(:,j+1) = v/beta(j);
    end
end
```

A few subprocesses/functions needed:

- A function to mutiply a sparse matrix A by a non-sparse column v. Can be parallelized?
- A dense vector dot product function.
- A dense vector norm function.

4.3 Eigendecomposition

```
void eigenDecomp(lanczosDecomp D, eigenDecomp E);
```

This can be done with LAPACK's SSTEVD routine.

4.4 Apply function

```
template <typename F>
void applyFunction(eigenDecomp E, F func);
```

This will apply the function to the `E.lambdas`. The function will be applied in place. Explicit instantiation for templates will be needed. `F func` is a functor or function pointer.

4.5 Multiply out

```
void multOut(lanczosDecomp D, eigenDecomp E);
```

A few things involved:

- Tall, skinny matrix-matrix multiplication to get QV.
- Vector-vector element-wise multiplication to get $f(\Lambda)v_{1*}$
- Tall, skinny matrix-vector multiplication to get $QVf(\Lambda)v_{1*}$
- Scaling vector by $\|x\|$ to get $\|x\|QVf(\Lambda)v_{1*}$

The approximation to $f(A)x$ will be stored in `D.x`.

Can potentially be grouped with previous step (applying f) and parallelized.