Lecture 3 The Preconditioned Lanczos Method for Solving Fractional Diffusion-Reaction Equations

Dr Qianqian Yang

AMSI Winter School 2019

In Lecture 2, we developed a numerical scheme for solving fractional diffusion-reaction equations. Recall the scheme:

$$\mathbf{u}^n = (\mathbf{I} + D_\alpha \tau \mathbf{A}^{\alpha/2})^{-1}(\mathbf{u}^{n-1} + \tau g(\mathbf{u}^{n-1})).$$

Remember that $\mathbf{A}$ is the matrix representation of the standard Laplacian and can be obtained from finite difference, finite element or finite volume methods under some boundary conditions. Although $\mathbf{A}$ is sparse, $\mathbf{A}^{\alpha/2}$ will be dense. When $\mathbf{A}$ is large, the direct computation of $\mathbf{A}^{\alpha/2}$ using diagonalisation is very expensive! Furthermore, what we want is not $\mathbf{A}^{\alpha/2}$, what we really want is the action of this dense matrix applied to a vector. This motivates us to write our numerical scheme in terms of **matrix-function-vector product** at each time step

$$\mathbf{u}^n = f(\mathbf{A})\mathbf{b}^{n-1}, \text{ where } f(\mathbf{A}) = (\mathbf{I} + D_\alpha \tau \mathbf{A}^{\alpha/2})^{-1} \text{ and } \mathbf{b}^{n-1} = \mathbf{u}^{n-1} + \tau g(\mathbf{u}^{n-1}).$$

Lanczos approximation

The prevailing method in the literature for approximating the matrix-function-vector product $f(\mathbf{A})\mathbf{b}$ for *symmetric* $\mathbf{A}$ is the Lanczos approximation

$$f(\mathbf{A})\mathbf{b} \approx \|\mathbf{b}\|\mathbf{V}_m f(\mathbf{T}_m)\mathbf{e}_1, \quad \mathbf{b} = \|\mathbf{b}\|\mathbf{V}_m \mathbf{e}_1,$$

where $\mathbf{A}\mathbf{V}_m = \mathbf{V}_m\mathbf{T}_m + \beta_{m+1}\mathbf{v}_{m+1}\mathbf{e}_m^T$ is the Lanczos decomposition, $\mathbf{T}_m$ is symmetric and tridiagonal, and the columns of $\mathbf{V}_m$ form an orthonormal basis for the Krylov subspace $\mathscr{K}_m(\mathbf{A}, \mathbf{b}) = \text{span}\{\mathbf{b}, \mathbf{A}\mathbf{b}, \ldots, \mathbf{A}^{m-1}\mathbf{b}\}$.

Don't panic! Let's see what this means and how to form such a decomposition.

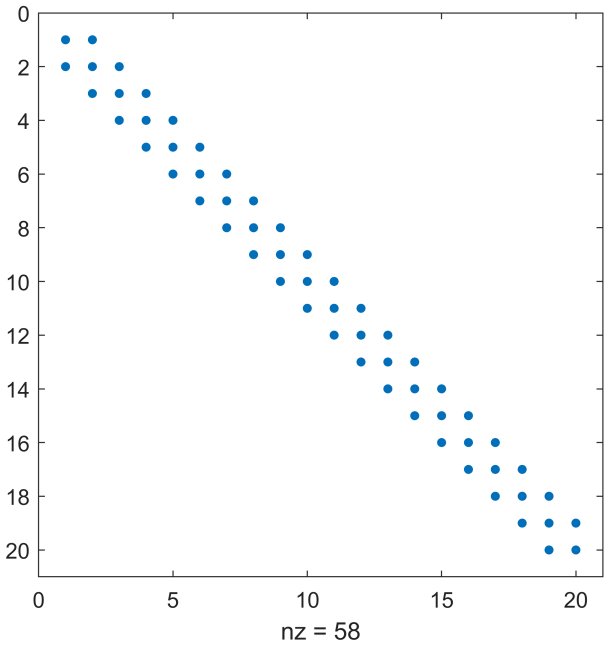
The clever idea behind this is that we want to project our large problem onto the Krylov subspace and find the approximate solution of our problem in that subspace. If the approximate solution is not good enough, we increase the dimension of the Krylov subspace by 1 and try again. Here, the dimension of a Krylov subspace *m* is much smaller than our original problem (i.e. $m \ll N$). When $\mathbf{A}$ is large, we will have problems to diagonalise $\mathbf{A}$, but we won't have any problem to diagonalise a much smaller symmetric and tridiagonal matrix $\mathbf{T}_m$, which is the projection of $\mathbf{A}$ onto the Krylov subspace.

Now, we need to be careful. There is an issue with using the basis $\{\mathbf{b}, \mathbf{Ab}, \ldots, \mathbf{A}^{m-1}\mathbf{b}\}$ for the Krylov subspace, which is that those vectors will soon become almost linearly dependent according to the theory of the power method. This will make it very difficult numerically to extract the new information added to the subspace when we expand our basis by each new vector.

The way around this difficulty is to *orthogonalise* each new vector against all the previous vectors as soon as we compute it.

Let's see how it works. We will build an orthonomal basis for the Krylov subspace $\mathcal{K}_m(\mathbf{A}, \mathbf{b})$. To illustrate the idea, let's say: $\mathbf{A}$ is a 20x20 *symmetric* matrix and $\mathbf{b}$ is a random vector.

```
clear
n = 20;
A = gallery('tridiag',n);
spy(A) % exhibits the locations of the nonzero entries
```




```
rng('default'); % to ensure you produce the same random vector
b = rand(n,1)
```

```
b = 20×1
    0.8147
    0.9058
    0.1270
    0.9134
    0.6324
    0.0975
    0.2785
    0.5469
    0.9575
    0.9649
     ⋮
```

Let's normalise our first vector by dividing by its length. We'll record the result as a single column in a matrix $\mathbf{V}_m$.

```
V(:,1) = b / norm(b)
```

```
V = 20×1
    0.2537
    0.2821
    0.0395
    0.2844
    0.1969
    0.0304
    0.0867
    0.1703
    0.2982
    0.3005
      ⋮
```

Now, we'll call our next vector, for the time being, $\mathbf{w}$. We generate the next vector by multiplying the previous vector by $\mathbf{A}$.

```
w = A * V(:,1)
```

```
w = 20×1
    0.2253
    0.2709
   -0.4874
    0.3324
    0.0790
   -0.2229
   -0.0272
   -0.0443
    0.1256
    0.2537
      ⋮
```

Now though, before we include it in the matrix $\mathbf{V}$, we'll orthonormalise it against the first vector. That is, we subtract away its projection onto $\mathbf{v}_1$ and scale the result by its length.

$$\mathbf{w} \rightarrow \mathbf{w} - (\mathbf{w} \cdot \mathbf{v}_1)\mathbf{v}_1$$

$$\mathbf{v}_2 = \mathbf{w}/\|\mathbf{w}\|$$

Since we expect they will be useful, we'll record the two intermediate scalars we needed in these calculations: $(\mathbf{w} \cdot \mathbf{v}_1)$ and $\|\mathbf{w}\|$. Let's store them in an array T.

```
% orthogonalise
T(1,1) = dot(w, V(:,1)),  w = w - T(1,1) * V(:,1)
```

```
T = 0.5711
w = 20×1
    0.0804
    0.1098
```

```
   -0.5100
    0.1699
   -0.0334
   -0.2402
   -0.0768
   -0.1416
   -0.0447
    0.0821
     ⋮
```

```
% normalise
T(2,1) = norm(w), V(:,2) = w / T(2,1)
```

```
T = 2×1
    0.5711
    1.0178
V = 20×2
    0.2537    0.0790
    0.2821    0.1079
    0.0395   -0.5011
    0.2844    0.1670
    0.1969   -0.0328
    0.0304   -0.2360
    0.0867   -0.0754
    0.1703   -0.1391
    0.2982   -0.0439
    0.3005    0.0806
     ⋮
```

We can now easily confirm that we have an orthonormal basis for $\mathscr{K}_2(\mathbf{A},\mathbf{b}) = span(\mathbf{b},\mathbf{Ab})$. The columns of $\mathbf{V}$ must be mutually orthogonal, and each have unit length.

```
V'*V
```

```
ans = 2×2
    1.0000   -0.0000
   -0.0000    1.0000
```

Great! Now, just automating this process to loop m times, we have derived the **Lanczos algorithm**.

```
m = 8;
for j = 2:m
    w = A*V(:,j); % Compute next vector

    % Orthogonalise against two previous columns
    T(j-1,j) = T(j,j-1); % T is symmetric
    w = w - T(j-1,j) * V(:,j-1); % subtract the projection on V(:,j-1)
    T(j,j) = dot(w, V(:,j));
    w = w - T(j,j) * V(:,j); % subtract the projection on V(:,j)

    % Normalise
    T(j+1, j) = norm(w);

    % Record the new vector in V
```

```
    V(:, j+1) = w / T(j+1, j);

end
V, T
```

```
V = 20×9
    0.2537    0.0790   -0.4564   -0.2552   -0.1301    0.3038    0.2481    0.1496 ···
    0.2821    0.1079    0.0820    0.4733    0.0447    0.0539   -0.3968   -0.2769
    0.0395   -0.5011   -0.0345    0.2308   -0.4348   -0.0427    0.0331    0.2796
    0.2844    0.1670    0.1664    0.0389    0.0877    0.3497    0.1352    0.2437
    0.1969   -0.0328   -0.1252   -0.4543   -0.0248   -0.0274   -0.5513   -0.1344
    0.0304   -0.2360    0.2345    0.0062    0.4165    0.0670    0.0704    0.2882
    0.0867   -0.0754    0.3661   -0.1653   -0.3576   -0.4246    0.1980   -0.0620
    0.1703   -0.1391    0.0276    0.0088    0.4464    0.0101    0.2859   -0.2313
    0.2982   -0.0439   -0.2445   -0.3066    0.0082   -0.2335    0.0103   -0.1388
    0.3005    0.0806    0.2396    0.1335    0.0940   -0.1044    0.0269   -0.0181
     ⋮
T = 9×8
    0.5711    1.0178         0         0         0         0         0         0
    1.0178    2.5668    0.9003         0         0         0         0         0
         0    0.9003    1.8564    0.8570         0         0         0         0
         0         0    0.8570    1.8080    1.0080         0         0         0
         0         0         0    1.0080    2.7397    1.0166         0         0
         0         0         0         0    1.0166    1.6644    0.7269         0
         0         0         0         0         0    0.7269    2.0842    1.0363
         0         0         0         0         0         0    1.0363    1.6394
         0         0         0         0         0         0         0    0.9389
```

```
norm(V'*V - eye(m+1,m+1),'inf') % check orthogonality
```

```
ans = 1.4256e-14
```

Note here that $\mathbf{T}$ is actually an $(m+1)\times m$ tridiagonal matrix. The $(m+1,m)$ entry, called β_{m+1}, is the norm of our final vector $\mathbf{w}_m$. To make this disctinction clear, we denote the $m\times m$ matrix resulting from removing the final row as $\mathbf{T}_m$.

```
Tm = T(1:m,1:m);
```

Before we continue, let's just confirm we have correctly computed the Lanczos decomposition $\mathbf{A}\mathbf{V}_m = \mathbf{V}_m\mathbf{T}_m + \beta_{m+1}\mathbf{v}_{m+1}\mathbf{e}_m^T$.

```
e = zeros(m,1); e(m) = 1;
norm(A*V(:,1:m) - V(:,1:m)*Tm - T(m+1,m)*V(:,m+1)*e')
```

```
ans = 3.3975e-16
```

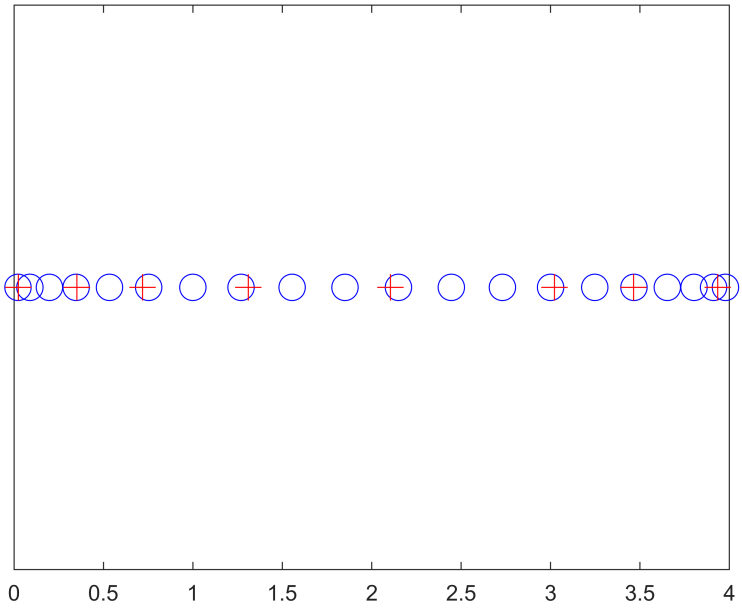
Great! Let's save the algorithm above as a MATLAB function lanczos.m.

The significance of this matrix $\mathbf{T}_m$ is that it is the projection of the matrix $\mathbf{A}$ onto the Krylov subspace $\mathscr{K}_m\{\mathbf{A},\mathbf{b}\}$. So, it's our "best approximation" of $\mathbf{A}$ in this space.

For example, we can find the eigenvalues of $\mathbf{T}_m$ and they should approximate those of $\mathbf{A}$, in some sense. Our main focus will be computing matrix functions, but functions of matrices are closely connected to their

eigenvalues. And actually the Lanczos method is a popular method for calculating eigenvalues anyway, so let's see how it performs.

```
lamA = eig(A); lamT = eig(Tm);
figure;
plot(lamA,0,'bo','MarkerSize',12); % blue circles: e.vals of A
hold on
plot(lamT,0,'r+','MarkerSize',12) % red crosses: e.vals of Tm
set(gca,'ytick',[]);
```



We see that the eigenvalues of $\mathbf{T}_m$ did a reasonable job of representing the spectrum of $\mathbf{A}$, especially those eigenvalues that are near the edge of the spectrum. There is more theoretical analysis on which eigenvalues the Lanczos method converges to, but it is beyond the scope of this lecture.

Matrix-function-vector product approximation

Now we are ready to use this Lanczos decomposition to approximate $f(\mathbf{A})\mathbf{b}$. Remember we "project" this problem onto the Krylov subspace. The m-dimensional projection of $\mathbf{A}$ is $\mathbf{T}_m$ as we already mentioned, and the m-dimensional projection of $\mathbf{b}$ is just $V_m{}^T\mathbf{b} = \|\mathbf{b}\|\mathbf{e}_1$. So our best approximation using the Krylov subspace would be to calculate $f(\mathbf{T}_m)(\|\mathbf{b}\|\mathbf{e}_1)$ and then promote the result back to $\mathbf{R}^n$: $f(\mathbf{A})\mathbf{b} \approx \mathbf{V}_m f(\mathbf{T}_m)\|\mathbf{b}\|\mathbf{e}_1$.

Now let's approximate a matrix-function-vector product $f(\mathbf{A})\mathbf{b} = \exp(\mathbf{A}^{\alpha/2})\mathbf{b}$ using the Lanczos method. We will take the tridiagonal matrix $\mathbf{A}$ with 1000 elements representing the discrete Laplacian in 1D. Let's try the standard case, $\alpha = 2$ first.

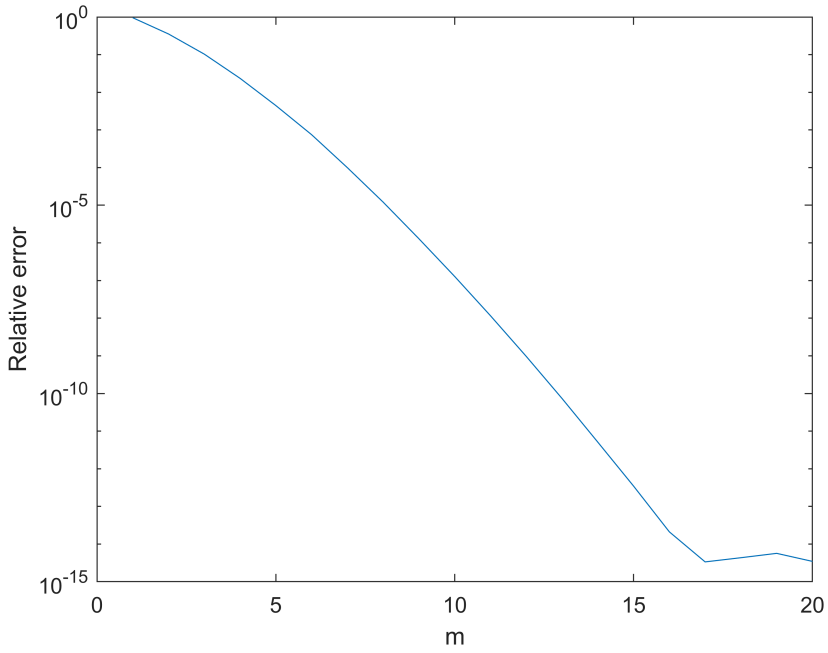
```
clear;
n = 1000; alpha = 2;
A = gallery('tridiag',n); b = rand(n,1);
```

```
m = 20; % let's just try if m=20 is resonalble for this matrix
[V, T, beta] = lanczos(A,b,m); % beta is norm(b)
fAb_exact = expm(full(A)^(alpha/2))*b; % using MATLAB's function to compute matrix exponential
e1 = [1;zeros(m-1,1)];
fAb_lanczos = beta * V(:,1:m) * expm(T(1:m,1:m)^(alpha/2)) * e1;
rel_err = norm(fAb_exact - fAb_lanczos) / norm(fAb_exact)
```

```
rel_err = 3.4482e-15
```

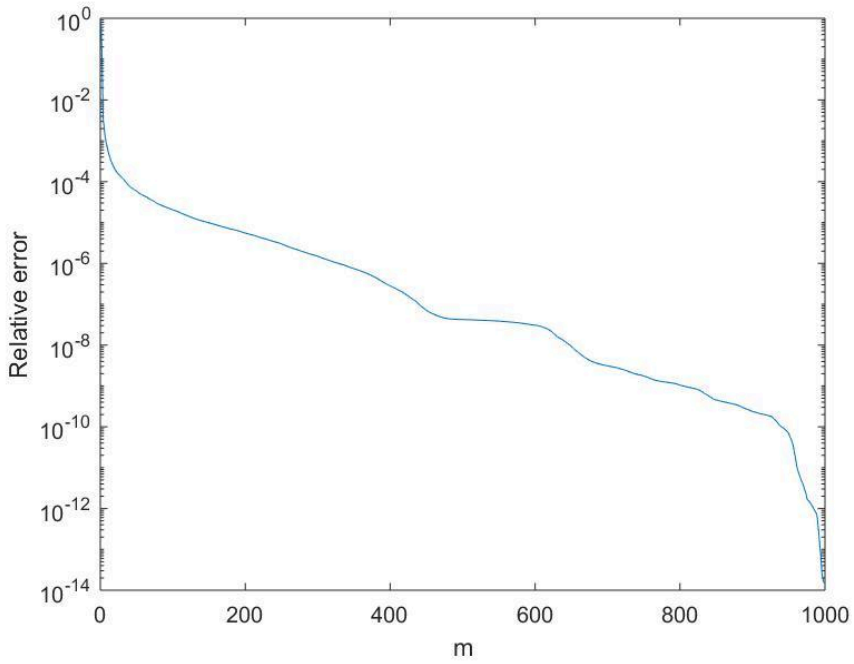
Wow, we get machine precision for $m = 20$, which is much less than our full dimension $n = 1000$. Let's take a closer look at how the error reduces with m for this example.

```
for k = 1:m
    err(k) = norm(fAb_exact - V(:,1:k) * expm(T(1:k,1:k)^(alpha/2)) * [beta;zeros(k-1,1)]) / no
end
figure; semilogy(err)
xlabel('m'); ylabel('Relative error')
```



Great! We see that the Lanczos method converges to machine precision in under 20 iterations when computing $f(\mathbf{A})\mathbf{b} = \exp(\mathbf{A})\mathbf{b}$.

Next, what if $\alpha \neq 2$, do you think our Lanczos algorithm would still work well? Unfortuantely, this is what you will see when computing $f(\mathbf{A})\mathbf{b} = \exp(\mathbf{A}^{\alpha/2})\mathbf{b}$ with $\alpha = 1.5$. The Lanczos method converges very slowly. To get to machine precision, you need really big m. That's not what we want.



The problem is that our function $f(t) = \exp(t^{\alpha/2})$ is not smooth anymore, for $\alpha \neq 2$. It has problems at the origin. The function is defined there, but its derivatives are not. And like most approximation theory, when derivatives don't behave well, the convergence is much slower. In this case, it's the smallest eigenvalues of our matrix (closest to zero) that cause us the most problems.

Preconditioned Lanczos Method

To accelerate the Lanczos method, we need a preconditioner. The idea is to construct a matrix $\mathbf{Z}^{-1}$ (the preconditioner) and work with the matrix function $f(\mathbf{A}\mathbf{Z}^{-1})$ rather than $f(\mathbf{A})$ itself. The matrix $\mathbf{Z}^{-1}$ should remove the eigenvalues that cause us problems, so that $f(\mathbf{A}\mathbf{Z}^{-1})\mathbf{b}$ is more accurate to compute than $f(\mathbf{A})\mathbf{b}$, but at the same time there must be a known relationship between $f(\mathbf{A}\mathbf{Z}^{-1})$ and $f(\mathbf{A})$.

Baglama et al. (1998) and Erhel et al. (1996) have both proposed the preconditioner $\mathbf{Z}^{-1}$ taking the form

$$\mathbf{Z}^{-1} = \lambda^{*}\mathbf{Q}_k\mathbf{\Lambda}_k^{-1}\mathbf{Q}_k^T + \mathbf{I} - \mathbf{Q}_k\mathbf{Q}_k^T$$

where $\lambda^{*} = (\lambda_{min} + \lambda_{max})/2, \mathbf{\Lambda}_k = \mathrm{diag}\{\lambda_1, \ldots, \lambda_k\}$, and $\mathbf{Q}_k = [\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_k]$ are formed by the k smallest eigenvalues $\{\lambda_i\}_{i=1}^{k}$ and eigenvectors $\{\mathbf{q}_i\}_{i=1}^{k}$ of the matrix $\mathbf{A}$.

Note that the product $\mathbf{A}\mathbf{Z}^{-1}$ has the same eigenvectors as $\mathbf{A}$ but its k smallest eigenvalues $\{\lambda_i\}_{i=1}^{k}$ are all mapped to λ^{*}. Hence, this preconditioner eliminates the influence of the k smallest eigenvalues on the convergence of the Lanczos method. The important observation at this point is the following relationship between $f(\mathbf{A}\mathbf{Z}^{-1})$ and $f(\mathbf{A})$

$$f(\mathbf{A})\mathbf{b} = \mathbf{Q}_k f(\mathbf{\Lambda}_k)\mathbf{Q}_k^T\mathbf{b} + f(\mathbf{A}\mathbf{Z}^{-1})\widehat{\mathbf{b}},.$$

where $\widehat{\mathbf{b}} = (\mathbf{I} - \mathbf{Q}_k\mathbf{Q}_k^T)\mathbf{b}$. If $\mathbf{A}$ is symmetric, then so is $\mathbf{A}\mathbf{Z}^{-1}$. Hence, we can apply the standard Lanczos decomposition to $\mathbf{A}\mathbf{Z}^{-1}$ with the new vector $\widehat{\mathbf{b}}$. In fact, we can show that $\mathscr{K}_m(\mathbf{A}\mathbf{Z}^{-1}, \widehat{\mathbf{b}}) = \mathscr{K}_m(\mathbf{A}, \widehat{\mathbf{b}})$, so in practice, we don't need to form $\mathbf{Z}^{-1}$ or $\mathbf{A}\mathbf{Z}^{-1}$ and multiply by $\widehat{\mathbf{b}}$, we can just simply build the Lanczos decomposition using $\mathbf{A}$ with the new vector $\widehat{\mathbf{b}}$.

Now, let's modify our code above to approximate $f(\mathbf{A})\mathbf{b} = \exp(\mathbf{A}^{\alpha/2})\mathbf{b}$ using a **preconditioned Lanczos method**.

```
clear;
n = 1000; alpha = 1.5;
A = gallery('tridiag',n); b = rand(n,1);
fAb_exact = expm(full(A)^(alpha/2))*b; % using MATLAB's function to compute matrix exponential
m = 50; % size of Krylov subspace
for kk = 0:20:100 % number of smallest e.values to be shifted
    [Qk,Lk] = eigs(A,kk,'smallestabs'); lamk=diag(Lk);  % compute kk smallest e.pairs
    term1 = Qk*diag(exp(lamk.^(alpha/2)))*(Qk'*b);
    bhat = b - Qk*(Qk'*b);
    [V, T, beta] = lanczos(A,bhat,m); % apply Lanczos to the new vector bhat
    e1 = [1;zeros(m-1,1)];
    fAb_lanczos = beta * V(:,1:m) * expm(T(1:m,1:m)^(alpha/2)) * e1;
    fAb = term1 + fAb_lanczos;
    kk, rel_err = norm(fAb_exact - fAb) / norm(fAb_exact)
    %norm(V'*V-eye(m+1,m+1),'inf') % check orthogonality
end
```

```
kk = 0
rel_err = 6.5022e-05
kk = 20
rel_err = 1.0765e-06
kk = 40
rel_err = 4.4896e-08
kk = 60
rel_err = 1.7744e-09
kk = 80
rel_err = 7.6717e-11
kk = 100
rel_err = 3.6999e-12
```

With the size of the Krylov subspace fixed, we can see how the deflated preconditioner help to improve the rate of convergence of the Lanczos method.

We will leave the computation of our solution as a matrix function vector product

$$\mathbf{u}^n = f(\mathbf{A})\mathbf{b}^{n-1}, \quad \text{where } f(\mathbf{A}) = (\mathbf{I} + D_\alpha \tau \mathbf{A}^{\alpha/2})^{-1} \text{ and } \mathbf{b}^{n-1} = \mathbf{u}^{n-1} + \tau g(\mathbf{u}^{n-1})$$

as Tutorial tasks.