The Role of Matrix Functions

Edvin Hopkins, Numerical Algorithms Group

November 2018

1 Introduction

In this report we'll be discussing functions of matrices. What are they, and why might they be of interest to you? We'll be using the NAG Library for *Python* to provide examples and we've included some code snippets that you can run if you have your own installation of the library[1].

Let's use the example of transition matrices from finance to set the scene. Suppose the vectors v_n represent some quantities of interest at time n. For example, each entry might be a credit rating or a stock price, and n might denote values obtained at the end of a particular year.

In a Markov model, the values of v_n from one year to the next are related by the equation $v_{n+1} = Pv_n$, where P is the **transition matrix**.

For example, here is a 1-year transition matrix for credit states obtained from Moody's Investment Services:

$$P = \begin{bmatrix} 0.8973 & 0.0976 & 0.0048 & 0 & 0.0003 & 0 & 0 & 0 \\ 0.0092 & 0.8887 & 0.0964 & 0.0036 & 0.0015 & 0.0002 & 0 & 0.0004 \\ 0.0008 & 0.0224 & 0.9059 & 0.0609 & 0.0077 & 0.0021 & 0 & 0.0002 \\ 0.0008 & 0.0037 & 0.0602 & 0.8545 & 0.0648 & 0.0013 & 0.0011 & 0.0019 \\ 0.0003 & 0.0008 & 0.0046 & 0.0402 & 0.8566 & 0.0788 & 0.0047 & 0.0014 \\ 0.0001 & 0.0004 & 0.0016 & 0.0053 & 0.0586 & 0.8406 & 0.0274 & 0.0066 \\ 0 & 0 & 0 & 0.0001 & 0.0279 & 0.0538 & 0.6548 & 0.2535 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0000 \end{bmatrix}.$$

What if we want to know the 6-month transition matrix? Is there anyway of obtaining it from the data above? Well after a little thought, you might be able to persuade yourself that we are after somethimg like the square root of P, namely $P^{1/2}$. But is this even defined, and if so how do we compute it? Is it unique? Could we similarly compute things like $P^{0.6}$?

The example above gives a motivation for computing non-integer powers of matrices. It is natural to then think about more general functions like exp, log, sin or cos. Can we compute them? Do they have any uses? Welcome to the wonderful world of matrix functions!

2 Defining a matrix function

Perhaps the most intuitive way of defining matrix functions is using Taylor series (various other equivalent definitions also exist). Recall that for a scalar, x,

[1] This report is also available from the NAG GitHub as a Jupyter notebook, in which you can run and edit the code snippets interactively.

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots .$$

For an n-by-n matrix, A, we can define the **matrix exponential** in a similar way:

$$e^A = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \cdots .$$

Functions such as $\sin A$, $\cos A$, $\sinh A$, $\cosh A$ can similarly be defined via their Taylor expansions. Convergence is guaranteed provided the scalar Taylor series converges at the eigenvalues of A.

2.1 Matrix powers and logarithms

Non-integer matrix powers and matrix logarithms are a little more involved because they are multivalued:

- any solution to $e^A = A$ is a logarithm of A,
- any solution to $X^p = A$ is a pth root of A.

For example, $\begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$ and $\begin{pmatrix} -2 & 0 \\ 0 & 2 \end{pmatrix}$ are both square roots of $\begin{pmatrix} 4 & 0 \\ 0 & 4 \end{pmatrix}$.

However, provided A has no eigenvalues on $\mathbb{R}^-$ we can define:

- a unique **principal logarithm**, $\log A$, with eigenvalues in $-\pi < \mathrm{Im}(z) < \pi$,
- a unique **principal pth root**, $A^{1/p}$, with eigenvalues in $-\pi/p < \arg(z) < \pi/p$.

If A is real then so are $\log A$ and $A^{1/p}$. General matrix powers can then be defined via $A^t = e^{t \log A}$.

2.2 A quick example

In the code snippet below, we use the naginterfaces.library.matop submodule from the NAG Library for *Python* to compute the principal cube root of a matrix.

```
In [1]: import numpy as np
        import naginterfaces.library as ni
        from naginterfaces.library import matop as ni_matop
        np.set_printoptions(precision=4)
        "Define a matrix"
        P = np.array([[55.0, 69.0, 30.0],
                      [30.0, 46.0,  9.0],
                      [39.0, 48.0, 37.0]])
        cbrtP = ni_matop.real_gen_matrix_pow(P, 1.0/3.0)
        print("Cube root:\n{}\n".format(cbrtP))
        print("Cubing the answer should give the original matrix:\n{}\n".
              format(cbrtP @ cbrtP @ cbrtP))
```

```
Cube root:
[[  3.0000e+00   2.0000e+00   1.0000e+00]
 [  1.0000e+00   3.0000e+00   7.5809e-16]
 [  1.0000e+00   1.0000e+00   3.0000e+00]]

Cubing the answer should give the original matrix:
```

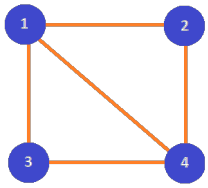
```
[[ 55.  69.  30.]
 [ 30.  46.   9.]
 [ 39.  48.  37.]]
```

3 Applications

We now know how to define a variety of matrix functions, but are they actually useful? Lets look at some applications.

3.1 Complex networks

Network are sets of **nodes** linked by **edges**. Examples include maps (in which the nodes and edges are towns and roads respectively) and social networks (in which the nodes and edges represent people and their connections). A network can be represented using an **adjacency matrix**. For example, here is a network, and its adjacency matrix:



$$A = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}.$$

When studying networks, we often wish to determine which nodes are the most important, and how well different nodes are connected. Matrix functions can help us here.

- The importance of node i is known as the **node centrality**. A measure of node centrality is given by $(e^A)_{ii}$.
- Similarly a measure of how well nodes i and j are connected is given by $(e^A)_{ij}$. This is known as the **node communicability**.

In the code below, we use the NAG Library function real_gen_matrix_exp to compute e^A for the adjacency matrix above. Perusing the output suggests that nodes 1 and 4 are the most important (they have the largest node centralities, the diagonal elements) and also the greatest communicability (the $(1, 4)$ element of e^A is the largest of the off-diagonal elements). This agrees with what we might expect from looking at the graphical representation of the network.

```
In [2]: A = np.array([[0., 1., 1., 1.],
                      [1., 0., 0., 1.],
                      [1., 0., 0., 1.],
                      [1., 1., 1., 0.]])
        expA = ni_matop.real_gen_matrix_exp(A)
        print("Matrix exponential:\n{}\n".format(expA))
```

```
Matrix exponential:
[[ 4.2482  3.0914  3.0914  3.8803]
 [ 3.0914  3.0186  2.0186  3.0914]
 [ 3.0914  2.0186  3.0186  3.0914]
 [ 3.8803  3.0914  3.0914  4.2482]]
```

3.2 Differential equations

An important application of matrix functions is in the solution of differential equations, which are ubiquitous throughout mathematics and the sciences. Consider the equation

$$\frac{d^2y}{dt^2} + Ay = 0, \quad y(0) = y_0, \quad y'(0) = y_0',$$

where y is a vector and A a square matrix. Its solution can be written, using the matrix sine, cosine and square root, as

$$y(t) = \cos(\sqrt{A}t)y_0 + \sqrt{A}^{-1}\sin(\sqrt{A}t)y_0'.$$

Alternatively, the solution can be written in terms of the exponential of a larger matrix:

$$\begin{bmatrix} y \\ y' \end{bmatrix} = \exp\left(\begin{bmatrix} 0 & -tA \\ tI_n & 0 \end{bmatrix}\right)\begin{bmatrix} y_0 \\ y_0' \end{bmatrix}.$$

We will return to this second form of the solution later.

3.3 Further comments

Markov chains are an important application of matrix functions. We encountered them earlier when we considered the roots of transition matrices. However, there is an important subtlety that we did not consider. Transition matrices are **stochastic**; that is they are non-negative, with unit row sums. Under what conditions are the matrix roots of a transition matrix themselves stochastic? This is acutally an unsolved problem.

The matrix exponential also has an application in Markov chains. The transition matrix P for a timestep t can be written as $P = e^{Qt}$, where Q is known as the transition intensity matrix. For P to be stochastic, Q must have vanishing row sums. Given a stochastic P, does such a Q exist? This is known as the embeddability problem, and it is also still unsolved.

Other applications of matrix functions include nuclear magnetic resonance spectroscopy, control theory, optics, computer graphics and particle physics.

4 How to evaluate matrix functions

Before discussing how we compute functions of matrices it is useful to know a little bit of theory.

Sometimes matrix functions behave like scalar functions. For example, the identities $e^{\log A} = A$ and $\sin^2 A + \cos^2 A = I$ apply to matrices as well as to scalars. But sometimes matrix functions do not behave like scalar functions. For example, in general $e^{A+B} \neq e^A e^B$ and $\log(e^A) \neq A$.

There are some extra properties that apply specifically to matrices. For example $f(A^T) = f(A)^T$. In addition, similarity transformations can easily be applied to matrix functions: if $A = VDV^{-1}$ then $f(A) = Vf(D)V^{-1}$. In particular, this means that if the eigenvalues of A are λ_i then the eigenvalues of $f(A)$ are $f(\lambda_i)$.

4.1 Can we evaluate matrix functions using similarity transformations?

The similarity transformation property suggests a possible method to compute matrix functions. Could we diagonalize, $A = VDV^{-1}$, and use $f(A) = Vf(D)V^{-1}$? Afterall, D is diagonal so we need only apply f to the diagonal elements. The code snippet below does just this. Note that we use the SciPy routine `linalg.solve` to increase numerical stability by avoiding explicit computation of V^{-1}.

```
In [3]: from scipy import linalg
        A = np.array([[ 0.5, 0.5],
                      [-0.5, 1.5]])
        D, V = linalg.eig(A)
        sqrtA = linalg.solve(V.T, np.diag(np.sqrt(D)) @ V.T).T
        print("Square root:\n{}\n".format(sqrtA))
        print("Residual: {:1.3e}".format(linalg.norm(sqrtA @ sqrtA - A)))
```

```
Square root:
[[ 0.75+0.j  0.25+0.j]
 [-0.25-0.j  1.25+0.j]]

Residual: 1.118e-08
```

The residual of $\sim 10^{-8}$ might seem small but recall that unit roundoff in double precision arithmetic is $\sim 10^{-16}$ so we might expect to do better. The problem here is that in exact arithmetic A is not actually diagonalizable. This results in a very poorly conditioned transformation matrix V.

The NAG Library routine for computing square roots of a real matrix is called real_gen_matrix_sqrt. Rather than diagonalizing, it uses a similarity transformation called a Schur decomposition, which is numerically more stable.

```
In [4]: sqrtA = ni_matop.real_gen_matrix_sqrt(A)
        print("Square root computed by NAG routine:\n{}\n".format(sqrtA))
        print("Residual: {:1.3e}".format(linalg.norm(sqrtA @ sqrtA - A)))
```

```
Square root computed by NAG routine:
[[ 0.75  0.25]
 [-0.25  1.25]]

Residual: 2.989e-16
```

Using real_gen_matrix_sqrt gave a far smaller residual.

The definitions we introduced earlier suggest an alternative way of evaluating matrix functions: Taylor series.

4.2 Can we evaluate matrix functions using Taylor series?

The code below computes sin A and cos A by summing the first 100 terms in their respective Taylor series (the 100^{th} term is already far smaller than machine precision so adding extra terms won't help us here). The code then checks whether the identity $\sin^2 A + \cos^2 A = I$ holds.

```
In [5]: A = np.array([[  6.0, 15.4],
                      [ 15.3,  0.0]])
        Atemp = -A @ A/2; cosA=np.eye(2); sinA = np.copy(A)
        for i in range(2,100,2):
            cosA = cosA + Atemp
            sinA = sinA + Atemp @ A/(i+1)
            Atemp = -Atemp @ A @ A/((i+1)*(i+2))
        print("sin^2 A + cos^2 A:\n{}\n".format(sinA @ sinA + cosA @ cosA))
        print("||sin^2 A + cos^2 A - I|| = {:1.3e}".
              format(linalg.norm(sinA @ sinA + cosA @ cosA-np.eye(2))))
```

```
sin^2 A + cos^2 A:
[[  1.0000e+00  -6.0594e-10]
 [  1.1036e-09   1.0000e+00]]

||sin^2 A + cos^2 A - I|| = 1.702e-09
```

Again, the residual of $\sim 10^{-9}$ is large in comparison with unit roundoff. Can NAG's routines do any better?

```
In [6]: sinA, imnorm1 = ni_matop.real_gen_matrix_fun_std('SIN', A)
        cosA, imnorm2 = ni_matop.real_gen_matrix_fun_std('COS', A)
        print("sin^2 A + cos^2 A:\n{}\n".format(sinA @ sinA+cosA @ cosA))
        print("||sin^2 A + cos^2 A - I|| = {:1.3e}".
              format(linalg.norm(sinA @ sinA + cosA @ cosA-np.eye(2))))
```

```
sin^2 A + cos^2 A:
[[  1.0000e+00  -1.1102e-16]
 [ -8.3267e-17   1.0000e+00]]

||sin^2 A + cos^2 A - I|| = 1.777e-16
```

In the code above we used real_gen_matrix_fun_std, which is a general purpose routine for computing various matrix functions. It is based on Taylor expansions but uses various additional tricks to improve the accuracy of the result.

4.3 So how do we evaluate matrix functions?

The state-of-the-art algorithms in the NAG Library use a variety of techniques depending on the function being computed. They are summarized in the bullet points below.

- **Schur decomposition**; $A = UTU^*$, where T is triangular and U unitary:
  - this is a well-conditioned decomposition,
  - it is cheaper to work with triangular matrices than full matrices.
- **Explicit formulae**:
  - these are available for diagonal and superdiagonal elements of $f(T)$, when T is triangular,
  - in addition a recursion can be used to compute the square root of triangular matrix.
- **Truncated Taylor series**:
  - the truncation is guided by some error analysis,
  - expansion about the mean of the eigenvalues improves stability.
- **Padé approximant**; $f(x) \approx p_m(x)/q_n(x)$ where p_m and q_n are polynomials of degree m and n:
  - for a given accuracy this can be cheaper than a truncated Taylor series (error analysis guides the choice of parameters m and n),
  - continued fraction and partial fraction representations of the approximant can also be useful.
- **Scaling and squaring**; we can exploit $e^A = (e^{A/s})^s$, where s typically a power of 2:
  - we choose s (using *a priori* error analysis) so that $\|A/s\|$ is small, for a better Padé approximant,
  - similar formulae are available for other functions e.g. $\log A = s \log(A^{1/s})$.

5 The action of the matrix exponential

Let's return to the important area of differential equations. Finite difference or finite element discretizations of partial differential equations often lead to an ordinary differential equation of the form

$$\frac{dy}{dt} = Ay, \quad y(0) = b.$$

The solution is $y(t) = e^{tA}b$. (Recall that the solution to the differential equation we saw earlier could also be written in terms of the product of a matrix exponential with a vector.)

In real-world applications A is typically large and sparse. For example, a 500×500 mesh in a PDE might yield a matrix with $n = 250000$, but only 0.002% nonzero entries. But e^{tA} is, in general, dense. Explicity computing it for $n = 250000$ would involve storing ~500 GB of data for the matrix alone, without even considering any temporary storage arrays we might need. Even with the best algorithms, this computation is prohibitively expensive. We are therefore left with the following task: can we evaluate $e^{tA}b$ without explicitly forming e^{tA}?

5.1 Computing the action of the matrix exponential: an example

We will demonstrate how to use the NAG Library routine real_gen_matrix_actexp_rcomm to compute $e^{tA}b$ without explicitly forming e^{tA}. It does this using only matrix-vector multiplications. The routine has a **reverse communication** interface, meaning that it returns control to the calling program whenever a matrix-vector multiplication is required. This means that we are free to use any storage format we wish for the sparse matrix A.

We begin by creating a random sparse matrix for our example, and setting up a few arrays that real_gen_matrix_actexp_rcomm will use as workspace.

```
In [7]: from scipy import sparse

        # Declare the matrix size and generate A, b and t
        n = 45000
        A = sparse.rand(n,n,0.0002,'csr')
        b = np.ones((n,1))
        t = 0.8

        # Create workspace arrays required by reverse communication routine
        irevcm = int(0)
        b2 = np.zeros((n,1))
        x = np.zeros((n,2))
        y = np.zeros((n,2))
        p = np.zeros(n)
        r = np.zeros(n)
        z = np.zeros(n)
        comm = {}

        # We will count the number of matrix-vector multiplications
        count = 0
```

The routine call itself is enclosed within a while loop. Depending on the value of the integer

irevcm, we must compute various matrix products (as described in the documentation for real_gen_matrix_exp_rcomm) before calling the routine once more.

```
In [8]: while True:
            # Call the nag routine
            irevcm = ni_matop.real_gen_matrix_actexp_rcomm(irevcm, b, t, b2,
                     x, y, p, r, z, comm)
            if irevcm == 0:
                # If the routine exits with irevcm=0 then we are done
                break
            elif irevcm == 1:
                b2 = A @ b
            elif irevcm == 2:
                y = A @ x
            elif irevcm == 3:
                x = A.T @ y
            elif irevcm == 4:
                p = A @ z
            elif irevcm == 5:
                r = A.T @ z
            count = count + 1

        print("e^(tA)b = {}\n".format(b.T))
        print("Evaluation of e^(tA)b required {} matrix-vector multiplications.".
              format(count))
```

```
e^(tA)b = [[ 12.4755  76.4364  14.002  …,  37.2913  35.9649  20.957 ]]

Evaluation of e^(tA)b required 64 matrix-vector multiplications.
```

6 Sensitivity of matrix functions

In linear algebra, whenever we compute something (be it the solution to a linear system, a set of eigenvalues, or a matrix function) it is useful to be able to measure how sensitive the result is to small perturbations in our initial data.

Here is an example. In the code below, we've used real_gen_matrix_log to compute the logarithm of a matrix A.

```
In [9]: A = np.array([[-2.00011, 0.49945],
                      [-0.499451, -2.999]])
        logA, imnorm = ni_matop.real_gen_matrix_log(A)
        print("log A:\n{}".format(logA))
```

```
log A:
[[ 670.1324  669.223 ]
 [-669.2244 -668.3002]]
```

Next, lets introduce a small perturbation, given by the matrix E, and compute $\log(A + E)$.

```
In [10]: E = np.array([[0.00001, 0],
                       [0,       0]])
         A2 = A + E
         logA2, imnorm = ni_matop.real_gen_matrix_log(A2)
         print("log(A+E):\n{}\n".format(logA2))
         print("Relative size of perturbation in A: {:1.3e}".
               format(linalg.norm(E)/linalg.norm(A)))
         print("Relative size of perturbation in log A: {:1.3e}".
               format(linalg.norm(logA-logA2)/linalg.norm(logA)))
```

```
log(A+E):
[[ 2220.9356  2220.0195]
 [-2220.024  -2219.1034]]

Relative size of perturbation in A: 2.722e-06
Relative size of perturbation in log A: 2.317e+00
```

The logarithm of our new matrix $\log(A + E)$ is very different from $\log A$ – the perturbation had a very significant effect.

The sensitivity to perturbations in our data is encapsulated by the **condition number**. As a rule of thumb a condition number of 10^k may mean k digits of accuracy are lost.

A related concept is that of the **Fréchet derivative**, $L_f(A, E)$. The Fréchet derivative of a matrix function tells us about changes in a particular direction:

$$f(A + E) - f(A) = L_f(A, E) + o(\|E\|).$$

The NAG Library contains routines for computing both Fréchet derivatives and condition numbers of matrix functions. In this final code snippet, we use real_gen_matrix_cond_log and real_gen_matrix_frcht_log to compute the condition number for the matrix logarithm and the Fréchet derivative in the direction E for the matrix A that we defined above. The results confirm that the logarithm of A is highly sensitive to small perturbations.

```
In [11]: logA, condla = ni_matop.real_gen_matrix_cond_log(A)
         print("Condition number for the matrix logarithm: {:1.3e}\n".
               format(condla))
         logA, L_log = ni_matop.real_gen_matrix_frcht_log(A, E)
         print("Derivative of log(A) in the direction E, L_log(A, E):\n{}\n".
               format(L_log))
```

```
Condition number for the matrix logarithm: 2.213e+05

Derivative of log(A) in the direction E, L_log(A, E):
[[ 304.2851  304.2814]
 [-304.2821 -304.2851]]
```

7 Summary

Matrix functions have many applications, in part due to the succinct way they allow us to write down solutions to certain problems (such as differential equations). Sophisticated algorithms are available, using a variety of techniques. The best way to exploit such algorithms is to use library code. Conditioning and stability is important consideration when computing matrix functions, particularly if there are uncertainties in your initial data.

7.1 Further reading

- The NAG Library for *Python* is available from:
  - https://www.nag.com/nag-library-python.
- The standard text for further information on matrix functions is:
  - Functions of Matrices: Theory and Computation. SIAM, 2008. ISBN 9878-0-898716-46-7, by Nicholas J. Higham.
- For questions and comments on matrix functions at NAG email **edvin.hopkins@nag.co.uk**.