Scientific Computing, Courant Institute, Fall 2021
http://www.math.nyu.edu/faculty/goodman/teaching/SciComp2021/index.html

The matrix exponential
For Assignment 3

This explains the matrix exponential and three ways to calculate it. The methods are illustrated in the posted code **MatrixExponential.py**. You should open this module in a code editor that shows line numbers.

If L is a $d \times d$ matrix, then e^L is the *matrix exponential*. This may be defined using the Taylor series that is the matrix version of the ordinary Taylor series for e^a one learns in calculus:

$$e^a = \sum_{n=0}^{\infty} \frac{1}{n!} a^n$$

$$e^L = \sum_{n=0}^{\infty} \frac{1}{n!} L^n \ . \tag{1}$$

We call e^a the *scalar* exponential to distinguish it from the matrix exponential. You can see that the Taylor series for the matrix exponential converges by comparing the terms to the corresponding terms in the scalar exponential series with $a = \|L\|$.

$$\left\| \frac{1}{n!} L^n \right\| \leq \frac{1}{n!} \|L\|^n \ .$$

You can check that if 0 is the $d \times d$ matrix of all zeros, then $e^0 = I$. It is not as obvious, but still true, that the inverse of e^L is e^{-L}, as it is for the scalar exponential.

You can replace L by tL, for a real (or complex) number, t. This gives the matrix *group* (sometimes called *semi-group* because only $t \geq 0$ is allowed for some infinite dimensional problems that are not part of this course)

$$S(t) = e^{tL} \ . \tag{2}$$

A *group*, in mathematics, is a collection of things that can be multiplied and inverted. The non-zero real or complex numbers form a group, as to the set of all invertible $d \times d$ matrices. For any fixed matrix L, the matrices $S(t) = e^{tL}$ form a group. You can check (it's not easy, but it is possible) that

$$S(t_1)S(t_2) = S(t_1 + t_2) \ . \tag{3}$$

This is like the scalar exponential formula $e^{t_1 a} e^{t_2 a} = e^{(t_1+t_2)a}$. However, the general product formula is not true

$$e^{A+B} \neq e^A e^B \quad , \quad \text{except in special cases.}$$

In fact, there is the *Baker Campbell Hausdorff* "formula" (really, approximation)

$$e^{t(A+B)} = e^{tA}e^{tB}e^{\frac{t^2}{2}(BA-AB)+O(t^3)} .$$

This formula is not important for this course right now, but it illustrates a difference between scalar and matrix exponentials, and it is easy to verify using the Taylor series formulas.

The matrix exponential arises as the solution of a linear system of differential equations. Suppose the variable $x(t)$ has d components, as

$$x(t) = \begin{pmatrix} x_1(t) \\ \vdots \\ x_d(t) \end{pmatrix} .$$

A system of linear differential equations takes the form

$$\frac{d}{dt}x(t) = \dot{x}(t) = Lx(t) . \tag{4}$$

The solution may be given using the *fundamental solution*, $S(t)$, which is a $d \times d$ matrix that satisfies

$$\dot{S}(t) = LS(t) , \quad S(0) = I . \tag{5}$$

Any vector solution is given by $x(t) = S(t)x(0)$, which you can verify by differentiating:

$$\frac{d}{dt}x(t) = \left[\frac{d}{dt}S(t)\right] x(0) = LS(t)x(0) = x(t) .$$

In this context, L is called the *generator* of the matrix group, which is the collection of matrices $S(t)$. You can check that the fundamental solution is given by the matrix exponential. One way is to show that the Taylor series formula satisfies the differential equation:

$$\begin{aligned} \frac{d}{dt}e^{tL} &= \frac{d}{dt}\left(I + tL + \frac{t^2}{2}L^2 + \frac{t^3}{6}L^3 + \cdots\right) \\ &= L + tL^2 + \frac{t^2}{2}L^3 + \cdots \\ &= L\left(I + tL + \frac{t^2}{2}L^2 + \cdots\right) \\ &= Le^{tL} . \end{aligned}$$

This shows that if $S(t)$ is the matrix exponential (2), then $S(t)$ satisfies the fundamental solution equations (5). Of course, the scalar exponential solves a one component linear differential equation in the same way. The matrix exponential also works for "row" differential equation systems as in the assignment.

$$\frac{d}{dt}p(t) = p(t)L \implies p(t) = p(0)S(t) = p(0)e^{tL} .$$

There are several ways to compute the matrix exponential numerically, but there is no universally best way. In fact, Cleve Moler (scientific "father" of Matlab) and Charles Van Loan (prominent researcher and co-author of the textbook *Matrix Computations*) have a **long list of methods for the matrix exponential**, all of them "dubious" (inferior to other methods in some cases). Here are three of them. They are implemented in the Python module `MatrixExponential.py`. If you can, open this file in a code editor that shows line numbers.

The direct method would be to sum the Taylor series (2). For this, you have to decide how many terms to take. One drawback is that you might have to take many terms and you have to do a matrix multiply for each term. Work for n terms is on the order of nd^3, which is more than the other methods. Another issue is cancellation, which happens even for the scalar exponential. For example,

$$e^{-t} = 1 - t + \frac{t^2}{2} - \frac{t^3}{6} + \cdots \ .$$

If t is a large positive number, it takes a lot of cancellation from big terms on the right to get the small answer on the left. Numerically, this leads to amplification of roundoff error. The function `meT` (starting on line 32) does this Taylor series sum directly. A fancy version of this would estimate the number of terms needed, possibly by computing the norms of the terms and stopping when they're within a specified tolerance.

The function `med` (for "matrix exponential differential equation") computes the matrix exponential in two steps. First, it estimates $S(\Delta t)$ for small Δt using just a few terms of the Taylor series. Later in the semester, we will see that this is equivalent to applying a *Runge Kutta* method to the differential equation system (5). This particular routine uses

$$S(\Delta t) \approx I + \Delta t L + \frac{\Delta t^2}{2} L^2 + \frac{\Delta t^3}{6} L^3 + \frac{\Delta t^4}{24} L^4 \ .$$

The error in this is order Δt^5, which can be quite small even if Δt is only a little small. The routine computes this approximation using *Horner's rule*. Horner's rule computes a polynomial starting with the highest order term, as

$$\begin{aligned} a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 &= a_0 + a_1 t + a_2 t^2 + a_3 t^3 \left(1 + \frac{a_4}{a_3} t\right) \\ &= a_0 + a_1 t + a_2 t^2 \left(1 + \frac{a_3}{a_2} t \left(1 + \frac{a_4}{a_3} t\right)\right) \\ &\ \ \vdots \\ &= a_0 \left(1 + \frac{a_1}{a_0} t \left(1 + \frac{a_2}{a_1} t \left(1 + \frac{a_3}{a_2} t \left(1 + \frac{a_4}{a_3} t\right)\right)\right)\right) \ . \end{aligned}$$

The algorithm first computes

$$m_4 = 1 + \frac{a_4}{a_3} t \ .$$

Then it computes

$$m_3 = 1 + \frac{a_3}{a_2} t\, m_4 \ ,$$

and so on until the final answer, in the form

$$m_0 = a_1 m_1 \ .$$

Line 24 computes this for the Taylor series.

$$S(\Delta t) \approx I + \Delta t L \left(I + \frac{\Delta t}{2} L \left(I + \frac{\Delta t}{3} L \left(I + \frac{\Delta t}{4} L \right) \right) \right) \ .$$

A small advantage is that you have $\frac{1}{n}$ instead of $\frac{1}{n!}$.

The product formula (3) implies that if $t = n\Delta t$, then

$$S(t) = S(\Delta t)^n \ .$$

This takes an approximation to $S(\Delta t)$ and gives an approximation of $S(t)$. You can compute a high power of a matrix by repeated squaring.

$$A \longrightarrow A^2 = A \cdot A \longrightarrow A^4 = \left(A^2\right) \cdot \left(A^2\right) \longrightarrow \cdots \ .$$

You can calculate A^n with $n = 2^k$ using just $k = \log_2(n)$ squarings. If n is not a simple power of 2, there is a more complicated algorithm that builds up A^n by multiplying A^{2^j} for the right numbers j. The result is A^n using at most $2\log_2(n)$ matrix multiplies. The function `med` chooses $n = 2^k$ for simplicity. The loop in lines 28 and 29 computes the squares. This algorithm is less dubious than the direct Taylor series summation of the function `meT` because it uses fewer matrix multiplications and because it suffers less from cancellation. However, for easy problems it achieves less accuracy than the Taylor algorithm because it requires small Δt to achieve high accuracy. This calls for a lot of squarings.

The function `mee` (for "matrix exponential eigenvalue") uses the eigenvalues and eigenvectors of L. The eigenvalue decomposition may be written

$$L = R \Lambda R^{-1} \ ,$$

where R is the matrix of right eigenvectors and Λ is a diagonal matrix with the eigenvalues. The `numpy` package comes with a sum-package `linalg` that does linear algebra calculations like this. Line 6 imports the names and definitions from `numpy.linalg` into the namespace `la` (for "linear algebra"). The function `eig` in this package computes the eigenvalues and eigenvectors. Line 10 calls this function (from the `la` namespace). It returns a Python tuple, with the eigenvalues as a numpy array and the eigenvector matrix. The matrix exponential has the same eigenvectors as L and the corresponding eigenvalues are $e^{t\lambda_j t}$. Line 13 constructs a diagonal matrix with these numbers on the diagonal. Then line 14 computes the product

$$e^{tL} = R \begin{pmatrix} e^{t\lambda_1} & 0 \\ 0 & \ddots \end{pmatrix} R^{-1} \ .$$

The inverse R^{-1} is computed by the `inv` function from `linalg`.

Many functions of a matrix may be computed using its eigenvalue/eigenvector decomposition. For example

$$L^2 = L \cdot L = \left(R\Lambda R^{-1}\right)\left(R\Lambda R^{-1}\right) = R\left(\Lambda^2\right)R^{-1} \ .$$

The matrix Λ^2 is (check this!) is the diagonal matrix with λ_j^2 on the diagonal. That is, L^2 has the same eigenvector matrix as L, and the eigenvalues are λ^2, More generally, if f is "any" function suitable function, $f(L)$, then $f(L)$ has the same eigenvectors as L and the eigenvalues are $f(\lambda)$. For example,

$$4L^2 + L^3 = R \cdot \left(\mathrm{diag}(\lambda_j^2 + \lambda_j^3)\right) \cdot R^{-1} \ .$$

You should check this for yourself.

The same reasoning applies to the Taylor series (1). You get

$$\begin{aligned}
e^{tL} &= \sum_{n=0}^{\infty} \frac{t^n}{n!} L^n \\
&= \sum_{n=0}^{\infty} \frac{t^n}{n!} R\left(\Lambda^n\right) R^{-1} \\
&= R\left[\sum_{n=0}^{\infty} \frac{t^n}{n!} \Lambda^n\right] R^{-1} \\
&= R\left[\mathrm{diag}\left(\sum_{n=0}^{\infty} \frac{t^n \lambda_j^n}{n!}\right)\right] R^{-1} \\
e^{tL} &= R\left[\mathrm{diag}\left(e^{\lambda_j t}\right)\right] R^{-1} \qquad (6) \\
& \qquad (7)
\end{aligned}$$

The matrix exponential is found using the ordinary exponential of the eigenvalues. Keep in mind that a real matrix may have eigenvalues and eigenvectors that are not real. In fact it usually does unless it has a reason not to. The left side of (6) is all real and the formula (1) is all real, but the right side of (6) has complex R and λ_j. Somehow, the imaginary part comes out to be zero. In computer arithmetic, the imaginary part will be on the order of roundoff, not zero. You will have to take the real part explicitly if you want a real result. Python automatically switches from real to complex numbers in situations like this, so you will have to *cast* the result back to real, using, for example

```
S = np.real(S)
```

In the code, you will see complex eigenvalues when you print them, even the eigenvalues are (mathematically) real.