SOLUTION FOR ASSIGNMENT 2

Problem 2

There are a few ways to compute the matrix exponential function of an $n \times n$ matrix $\mathbf{A}$ by summing up the series

$$\exp(\mathbf{A}) = \sum_{k=0}^{\infty} \frac{1}{k!}(\mathbf{A})^k = \mathbf{I} + \mathbf{A} + \frac{1}{2!}\mathbf{A}^2 \cdots + \frac{1}{k!}\mathbf{A}^k \cdots,$$

up to a maximum of M terms. One expect the accuracy of the result to increase with M. The differences of these ways have to do with the way how the series is summed up.

The general term of the series is $\frac{1}{k!}\mathbf{A}^k$. Since for any given integer k and any square matrix $\mathbf{A}$, there are functions to compute $\mathbf{A}^k$ and $k!$ in Matlab, so one may simply transcribe the formula into a program, as shown as follows.

```
    % 1.1 An even dumber implementation
EXPMADUMBER = eye(n);                    % initialize sum to identity matrix
AK = eye(n);                             % initialize AK to I
fac = 1.0;                               % initialize k! to 1
for k=1:maxLoop
    AK = A^k;                            % compute A^k
    fac = factorial(k);                  % compute k!
    EXPMADUMBER = EXPMADUMBER+AK/fac;    % compute & accumulate A^k / k!
end
```

Here the maximum number of terms to be summed has been initialize:

```
maxLoop = 30;     % max number of terms summed in the series method
```

This is probably the dumbest way to sum up the terms. For the k-th term, the computation *essentially* involves k matrix multiplications of $\mathbf{A}$ with itself in order to get `AK` and k ordinary multiplications to get `fac`. (Matlab actually uses better algorithms to compute these quantities using fewer number of operations.) The calculation also involves a division of a matrix by a scalar and a matrix addition.

Instead of repeatedly multiplying $\mathbf{A}$ with itself, and repeatedly multiply $1 \times 2 \times \cdots$ to obtain the factorial, a much better way is to have a variable `AK` that successively compute $\mathbf{A}$ to a higher and higher power, and a variable `fac` for the factorial. The resulting Matlab code fragment is shown below.

```
% 1. Compute by summing its series expansion:
    % 1.1 A good implementation
EXPMADUMB = eye(n);                     % initialize sum to identity matrix
AK = eye(n);                            % initialize AK to I
fac = 1.0;                              % initialize k! to 1
for k=1:maxLoop
    AK = AK*A;                          % compute A^k
    fac = fac*double(k);                % compute k!
    EXPMADUMB = EXPMADUMB+AK/fac;       % compute & accumulate A^k / k!
end
```

To compute the k-th term, only one matrix multiplication is need to compute $\mathbf{A}^k$, and only one ordinary multiplication is needed to compute $k!$. The calculation also involves a division of a matrix by a scalar and a matrix addition. The most time consuming part of the calculation (involving matrix multiplication) has now been made much more efficient.

An even better implementation can be obtained by noticing that if we already have computed the $(k-1)$-th term, then the kth term can be obtained by simply multiplying by $\mathbf{A}$ and dividing by k. The resulting code fragment is shown below.

```
    % 1.3 The best method for summing the series
SK = eye(n);                            % the kth term in the series for exp(A)
EXPMA1 = eye(n);
for k=1:maxLoop
    ik = 1/k;                           % a speed-up of several percent
    SK = ik*A*SK;
    EXPMA1 = EXPMA1 + SK;
end
```

Although when compared with the previous implementation in terms of the number of operations per loop, this implementation is faster only because it has one fewer ordinary multiplications (since the factorial is not computed separately). Although this increase in speed is hardly noticeable, it is better in that it avoids the overflow problem when `maxLoop` is large.

For $\mathbf{A}_1$ both methods yield results essentially identical to the result obtained from Matlab:

```
EXPMAMATLAB =
          11.4019093758234          -8.68362754736431
          -8.68362754736431          11.4019093758234
```

However in the case of $\mathbf{A}_2$, the relative error for method 1 is about 1×10^{-9}, while the relative error for method 2 is about 1×10^{-14}. The actual output from Matlab is:

```
EXPMAMATLAB =
        -0.735758758144693          0.551819099658051
         -1.47151759908814           1.10363824071548
```

For a randomly created matrix whose elements lie between 0 and 1, the elements of $\exp(\mathbf{A})$ become large. It is better to consider the relative errors in the resulting matrix when we compare with the result obtained from Matlab's `expm` function.

When one is testing to see which of the two methods run faster, it is important to use a matrix large enough so that the run time is at least a good fraction of a second since Matlab's timing mechanism may not be accurate to tens of milliseconds. It is also important to use a value for `maxLoop` that just gives results having roughly the same errors as for method two (roughly like 1×10^{-14}). It is unfair for method one if a higher value is used. We should also not compare the run-times if too small a value is used since the results will have different accuracies. With this in mind, for example, for a matrix of size $n = 100$ and `maxLoop`$= 120$, method 2 is found to be faster than method 1 by about a factor of 2.

Method 2 is more accurate and robust in general. When the matrix has eigenvalues of vastly different magnitudes (for example $\mathbf{A}_2$ whose eigenvalues are -17 and -1, or when we use a random matrix of a large size), method 1 gives inaccurate results. This is because when we repeated multiply the matrix by itself, the elements will have magnitudes differing by many orders of magnitudes. Some accuracy is then lost.

This does not happen for $\mathbf{A}_1$ since its two eigenvalues (3 and 1) have more comparable magnitudes. The results for method 1 is therefore also less robust since the accuracy of the results depends on the relative sizes of the eigenvalues of the matrix.