Lecture 3: Special Matrices

Feedback of assignment1

- **Random matrices**
  The magic matrix commend `magic()` doesn’t give us random matrix. Random matrix means we will get different matrices each time when we produce them. But `magic(n)` will give us the same \( n \times n \) matrix no matter how many times we run it. Besides `rand()` and `randn()`, you also can try `randi()` to get a random matrix with integer elements.

- **Apostrophe**
  When we copy and paste MATLAB code from a .pdf file, the notation apostrophe (’) sometimes will be changed and the code will be not executed. The way to solve this problem is delete and retype it.

- **Solving \(Ax=b\)**
  Given two square matrices \(A\) and \(B\). The main difference of the left division (commend: \(\backslash\) or `mldivide()`) and the right division (commend: `/` or `mrdivide()`) is that, for left division

  \[
  A\backslash B \text{ or } \text{mldivide}(A,B) \Rightarrow \text{inv}(A)\ast B
  \]

  and right division

  \[
  A/B \text{ or } \text{mrdivide}(A,B) \Rightarrow A\ast\text{inv}(B).
  \]

  So when we want to solve the matrix system \(Ax = b\) for unknown \(x\), the answer \(x\) will be \(A^{-1}b\) if \(A\) is invertible. Then the commend we should use is

  \[
  A\backslash b \text{ or } \text{mldivide}(A,b).
  \]

Review of Basic Matrix Functions
Let’s start by reviewing some of the functions we’ve covered in Lecture 1, as well as elaborate on combining these functions to generate more types of matrices.

- **zeros()**
  It’s a good habit to initialize matrices with a matrix of all zeros. If we’d like to make a matrix \(A\) that consists of only zeros, we can type

  ```
  >> A = zeros(3,4);
  ```

  into the command window. This produces a matrix \(A\) of size \(3 \times 4\) of zeros, i.e.

  \[
  A = \begin{bmatrix}
  0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 \\
  \end{bmatrix}
  \]
• **ones()**
  Another function which we’ve seen previously is **ones()**, which generates a matrix with entries of 1. This is another useful way to initialize a matrix, but with nonzero entries. Set a $7 \times 7$ matrix $B$ of all ones by typing

$$B = \text{ones}(7, 7);$$

which would produce a matrix that looks like

\[
\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1
\end{bmatrix}
\]

If your goal is to memorize as few functions as possible, there’s another way to get a matrix like this using **zeros()**. Try typing

$$B' = \text{zeros}(7, 7) + 1;$$

This produces the same result. MATLAB interprets scalar addition to matrices as adding the scalar to each index. It’s a nice shorthand to use. Similarly, we can modify the values by multiplication. Try

$$C = B' \times 5;$$

in the command window. This has changed all of the entries to 5. Like addition, MATLAB interprets scalar multiplication as multiplication to each index of the matrix.

• **eye()**
  The last review function is **eye()**. This will give an identity matrix. Let’s making a $7 \times 7$ identity matrix by typing

$$E = \text{eye}(7);$$

which would give

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
\]
We can also combine these simple matrices to make slightly more complex ones. Take, for example, $B - E$. Then, we’d get a matrix of the form

\[
\begin{bmatrix}
0 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 0
\end{bmatrix}
\]

or maybe we’d like to update the top left of $B$ with $A$. Try typing

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1
\end{bmatrix}
\]

This is a way to do block matrices, which will be discussed in the next section.

- **diag()**
  This is a function we’ve yet to introduce and it has a couple nice properties.

Initially, it sounds like a way to pull out the diagonal entries of a square matrix. Indeed, it does this. Set the matrix $M = \text{magic}(3)$, like before. Now, try

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1
\end{bmatrix}
\]

What’s the result? It pulled out the diagonal entries of $M$ and stored them as a column vector in $d$. In fact, it works with non-square matrices, but only up to the smallest size. So, for example, take the matrix

\[
U = \begin{bmatrix} 2 & 3 \\ 4 & 5 \\ 6 & 7 \end{bmatrix}
\]

and I type \text{diag}(U) into MATLAB, you should see

\[
\text{ans} = \\
\begin{bmatrix} 2 \\ 5 \end{bmatrix}
\]
which are the ‘diagonal’ entries of the matrix, starting at the top left corner. This may be useful when we talk about SVD in a week or two, so it is good to keep in mind.

The function \texttt{diag()} also has a second use. It can be used to generate a matrix with diagonal entries, if the argument (the input inside the parantheses) is a vector. So, try

\begin{verbatim}
>> diag ([1;1])
\end{verbatim}

What does that output? It should be an identity matrix of size two. Try something else. Let’s try

\begin{verbatim}
>> diag ([1:7])
\end{verbatim}

We’ve combined two ideas here. First note that \([1:7]\) generates a vector starting at 1 and going to 7 by an increment of 1 each time. However, then we apply \texttt{diag()} to it, so we get a matrix with that vector as the diagonal. In other words, your command window should display

\begin{verbatim}
ans =
 1 0 0 0 0 0 0
 0 2 0 0 0 0 0
 0 0 3 0 0 0 0
 0 0 0 4 0 0 0
 0 0 0 0 5 0 0
 0 0 0 0 0 6 0
 0 0 0 0 0 0 7
\end{verbatim}

Feel free to try this with other vectors or functions and see what happens.

\textbf{Matrix Indexing and Block Matrices}

Before we begin, we should quickly mention how indexing works. If I have a matrix \(A\) of size \(m \times n\), then \(m\) corresponds to the number of rows and \(n\) to the number of columns. To access an element of \(A\), we use this same idea by picking the index corresponding to the row first, then column. So, if we have the matrix

\[
A = \begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
\end{bmatrix}
\]

then \(A(2,1) = 4\) and \(A(3,3) = 9\), for example. Since \((2,1)\) corresponds to the second row and first column of matrix \(A\), we get 4. Similarly, for 9.

Further, we can access entire rows or columns, parts of these, or sub-matrices. For example, suppose we want the upper \(2 \times 2\) matrix embedded in \(A\), then we can type

\begin{verbatim}
>> A (1:2 , 1:2)
\end{verbatim}
and pull out the result

\[
\begin{pmatrix}
1 & 2 \\
4 & 5
\end{pmatrix}
\]

In the reverse, we can construct a matrix from other matrices. These sub-matrices are called block matrices. Try typing this into the command window:

```matlab
>> A1 = magic(3); A2 = zeros(3,2); A3 = ones(2,3); A4 = [5,5;3,3];
```

It’s important to know and keep track of the size of these matrices. \( A1 \) is size \( 3 \times 3 \), \( A2 \) is \( 3 \times 2 \), \( A3 \) is \( 2 \times 3 \), and \( A4 \) is \( 2 \times 2 \). Now, suppose we want to create the \( 5 \times 5 \) matrix \( A \) such that

\[
A = \begin{bmatrix} A1 & A2 \\ A3 & A4 \end{bmatrix}
\]

Notice that this is a consistent matrix that is made up of smaller matrices. In fact, we can also do this in MATLAB. To do so, write them down the same way you would scalar elements:

```matlab
>> A = [A1,A2; A3,A4]
```

and the result should print out

\[
\begin{pmatrix}
8 & 1 & 6 & 0 & 0 \\
3 & 5 & 7 & 0 & 0 \\
4 & 9 & 2 & 0 & 0 \\
1 & 1 & 1 & 5 & 5 \\
1 & 1 & 1 & 3 & 3
\end{pmatrix}
\]

Again, keep the matrix sizes consistent. If we tried

```matlab
>> Anew = [A1,A3; A2,A4]
```

we’d get an error. Why? Further,

```matlab
>> Anew = [A1,A3'; A2',A4]
```

works. Why does this one work and the other doesn’t? What’s the difference between \( Anew \) and \( A \)? In general, if matrix \( A1 \) is size \( a \times b \), then \( A2 \) needs to be \( a \times c \) and \( A3 \) needs to be \( c \times b \), and then matrix \( A4 \) needing to stay consistent, needs to be size \( c \times c \).

Further, recall the idea of a diagonal matrix - a matrix that has non-zero elements only on its diagonal. This idea extends to block matrices, which we call block diagonal. If I have
sub-matrices $A_1, A_2, \ldots, A_n$, then a block diagonal matrix $A$ would satisfy

$$A = \begin{bmatrix}
A_1 & 0 & 0 & \cdots & 0 \\
0 & A_2 & 0 & \cdots & 0 \\
0 & 0 & A_3 & \cdots & 0 \\
\vdots & \vdots & \ddots & \ddots & \vdots \\
0 & 0 & 0 & \cdots & A_n
\end{bmatrix},$$

Notice that there’s no requirement on the sizes of this matrices. So, for example,

$$A = \begin{bmatrix}
3 & 2 & 0 & 0 & 0 \\
1 & 5 & 0 & 0 & 0 \\
0 & 0 & 1 & 3 & 6 \\
0 & 0 & 2 & 4 & 5
\end{bmatrix},$$

is block diagonal, because

$$A_1 = \begin{bmatrix}
3 & 2 \\
1 & 5
\end{bmatrix}, \quad A_2 = \begin{bmatrix}
1 & 3 & 6 \\
2 & 4 & 5
\end{bmatrix}$$

make up the block diagonals, so that

$$A = \begin{bmatrix}
A_1 & 0 \\
0 & A_2
\end{bmatrix}$$

To produce the block diagonal matrix, if we have the diagonal matrices $A_1, A_2, A_3, \ldots, A_n$ and we want to get

$$A = \begin{bmatrix}
A_1 & 0 & 0 & \cdots & 0 \\
0 & A_2 & 0 & \cdots & 0 \\
0 & 0 & A_3 & \cdots & 0 \\
\vdots & \vdots & \ddots & \ddots & \vdots \\
0 & 0 & 0 & \cdots & A_n
\end{bmatrix},$$

we can create those zero matrices one by one or use the command `blkdiag()`. For example,

We can get

$$A = \begin{bmatrix}
A_1 & 0 & 0 \\
0 & A_2 & 0 \\
0 & 0 & A_3
\end{bmatrix}$$

by using `blkdiag(A1,A2,A3)`.

Note: In this instance, zero’s used here are matrices of zero such that the matrix size stays consistent. Further, notationally, it aligns well with the case of scalar diagonal matrices.
We’d now like to discuss two types of special matrices. Consider the matrix:

$$U = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

This is called an upper triangular matrix. It’s defined by any elements of matrix $A$ below the diagonal being zero. In math terms, this is equivalent to $U_{ij} = 0$ for all $i > j$, where $U_{ij}$ corresponds to the element in the $i^{th}$ row and $j^{th}$ column of matrix $U$.

These have nice properties. One property that’s easy to see is they are incredibly fast to solve linear equations of this form. Take matrix $U$ and $b = [3; 2, 1]$. Then, solving $Ux = b$, we can just start at the bottom and work our way up, since:

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix}$$

We can see by looking at the bottom row that $x_3 = 1$. Then, we can plug that into the next step up and get that $x_2 = 1$ also, since $x_3 = 1$ and $2 - 1 = 1$. Repeating for the top row, we have $x_1 = 1$.

In fact, this is one of the main ideas behind Gaussian elimination (or row reduction), to perform matrix operations to get a matrix down to an upper triangular matrix, then again to get a diagonal matrix. Since it’s fairly slow in terms of arithmetic operations, it’s not used as commonly in practice. However, there are other algorithms that make use of upper triangular matrix properties to solve matrix equations much faster, especially if they have a specific form.

But, not surprisingly, there are also matrices called lower triangular matrices. They might look like

$$U = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

where now all the entries above the diagonal are zero. Again, in math speak, this means $L_{ij} = 0$ for every $i < j$. There’s also the definition of a unitriangular matrix, which means the diagonal has only ones ($L_{ij} = 1$ for $i = j$). But, we won’t really focus on those too much.

Again, these matrices have nice properties. For example, they can be solve linear systems instead by working at the top of the rows and going down, which is the reverse of the upper triangular case. As an exercise, you may want to see how it works by solving $Lx = b$, with $L$ and $b$ defined above.
• **triu()**

Let's now learn some MATLAB functions that give triangular matrices. Our first example is on the function `triu()`. As the name implies, this will be giving us upper triangular matrices. So, first generate a matrix by typing

```matlab
>> A = [1,2,3;4,5,6;7,8,9];
```

Alright, and now let's try

```matlab
>> U = triu(A);
```

What happens? Well, the function sets the elements below the diagonal as zero. So, you should see

```plaintext
U =
1 2 3
0 5 6
0 0 9
```

in the Command Window. This works on matrices that aren’t square, too. Type,

```matlab
>> B = A( :, 1:2 )
```

into the Command Window. You should see

```plaintext
B =
1 2
4 5
7 8
```

So, we’ve just cut off part of $A$. Now, if you try `triu(B)`, you’ll see

```plaintext
ans =
1 2
0 5
0 0
```

So, it will still doing something that looks upper triangular, but starting at the top left corner and going down diagonally.

This function has more features. We can also specify how much of the upper triangular matrix we want. Suppose we want everything above diagonal of $A$, but not the diagonal itself, and everything else zero. Well, we can do this by typing

```matlab
>> triu(A,1)
```

which looks like
ans =
    0   2   3
    0   0   6
    0   0   0

Similarly, if we do \texttt{triu(A,2)}, we see

\begin{verbatim}
ans =
    0   0   3
    0   0   0
    0   0   0
\end{verbatim}

The pattern should be clear. \texttt{triu(A,k)} means take the upper triangular matrix of \(A\), but with \(k\) diagonals cut off. If \(k = 0\), that’s the same as \texttt{triu(A)}. If \(k = 1\), we take away the diagonal. If \(k = 2\), we take the diagonal and the diagonal above it, etc. It also works in reverse, so if we set \(k = -1\), we get the diagonal below the main diagonal too:

\begin{verbatim}
>> triu(A,-1)
\end{verbatim}

\begin{verbatim}
ans =
    1   2   3
    4   5   6
    0   8   9
\end{verbatim}

Again, this is a useful way of modifying matrix entries. Before, we could only modify by rows or columns. This allows us to modify things diagonally.

- \texttt{tril()}
  This function is similar to \texttt{triu()}, except now it generates lower triangular matrices. Using \(A\) as an example again, we try \texttt{tril(A)} and see

\begin{verbatim}
>> tril(A)
\end{verbatim}

\begin{verbatim}
ans =
    1   0   0
    4   5   0
    7   8   9
\end{verbatim}

We can also specify how many diagonals to keep using the syntax \texttt{tril(A,k)}, where \(k\) is an integer. You should experiment with this to see how it works or how you can combine these functions to keep specific values in your matrix.

For example, suppose we only want the top right and bottom left entry of matrix \(A\). We can do this by summing two of these operations:
\[ \text{triu}(A, 2) + \text{tril}(A, -2) \]

\[
\begin{array}{ccc}
0 & 0 & 3 \\
0 & 0 & 0 \\
7 & 0 & 0 \\
\end{array}
\]

We could also only take the diagonal:

\[ \text{tril}(\text{triu}(A)) \]

\[
\begin{array}{ccc}
1 & 0 & 0 \\
0 & 5 & 0 \\
0 & 0 & 9 \\
\end{array}
\]

You also can try the command \texttt{triu(triu(A)')} . There's a lot of different ways to keep the elements you want and disregard the rest. Much of your assignment focuses on understanding how you can use these functions to get the information you want and they might require some thought.

- **Diagonal number \( k \)**
  Here are a graph about the integer parameter \( k \) we've used in commands \texttt{triu()} and \texttt{tril()}. \( k = 0 \) represents the main diagonal, \( k > 0 \) is above the main diagonal, and \( k < 0 \) is below the main diagonal.

\[ k = 0 \quad k > 0 \\

k < 0 \]

\[ \]

**Symmetric Matrices**

A specific type of matrix, one which has a lot of nice properties, are symmetric matrices. A symmetric matrix is one such that the elements reflected along the diagonal are the same.
In math terms, $A_{ij} = A_{ji}$ for every $i$ and $j$. An example of a symmetric matrix would be

\[
S = \begin{bmatrix}
1 & 2 & 4 \\
2 & 5 & 3 \\
4 & 3 & 7
\end{bmatrix}
\]

Notice how if I folded the elements across the diagonal, they would be the same. All diagonal matrices are symmetric, for example. Similarly, an immediate result is $A^T = A$. We can build symmetric matrices pretty easily. Take $A$ above, and do the following:

\[
\begin{bmatrix}
1 & 6 & 10 \\
2 & 10 & 14 \\
10 & 14 & 18
\end{bmatrix}
\]

Notice it’s symmetric. We could do similar with with just the upper triangular part. Try

\[
\begin{bmatrix}
2 & 2 & 3 \\
2 & 10 & 6 \\
3 & 6 & 18
\end{bmatrix}
\]

This is symmetric. However, $\text{triu}(A) + \text{triu}(A')$ is not. Why? But, if I do $\text{triu}(A) + \text{tril}(A')$, I get the same result as having done $\text{triu}(A) + \text{triu}(A')$. You should be able to justify why this happens.

I can also make it so that I keep the diagonal fixed. Suppose I want a symmetric matrix from $A$ that only uses the upper half of the elements, but doesn’t change the diagonal. We can do that by typing

\[
\begin{bmatrix}
1 & 2 & 3 \\
2 & 5 & 6 \\
3 & 6 & 9
\end{bmatrix}
\]

Again, there’s many other matrices that can be made using only these functions. Some of these will be part of your homework assignment. Remember, if you’re stuck on how to use a function, you can type `help triu`, for example, to pull up documentation on the function.
**Transpose and Conjugate Transpose**

We have used the command apostrophe ('') a lot for the transpose of matrices. In fact, when we are using apostrophe on a matrix $A$, we are asking MATLAB to find the **conjugate transpose matrix** of $A$. Since we have only tried real matrices so far, the conjugate transpose of a real matrix is the same thing as the transpose of it. A real matrix means each element in this matrix is a real number. So let’s see what $A'$ is if $A$ is a complex matrix.

- **Complex Number in MATLAB**
  
  To introduce the complex numer, first let’s try a classic example to see if MATLAB can recognize the complex numbers or not:

  ```matlab
  >> sqrt(-1)
  ans =
  0.0000 + 1.0000i
  ```

  which means $\sqrt{-1} = i$. So MATLAB understands what complex number is. A complex number can be expressed in the form $a + bi$, where $a$ and $b$ are real number and $i$ is the imaginary unit. In this expression, $a$ is the real part and $b$ is the imaginary part of the complex number.

  We can type complex numbers in MATLAB directly. For instance,

  ```matlab
  >> 1+2i
  ans =
  1.0000 + 2.0000i
  ```

  Here we don’t need the multiplication notation (*) between 2 and $i$.

  The **complex conjugate** of the complex number $a + bi$ is defined to be $a - bi$. We use a bar to represent the conjugate of a complex number:

  $$a + bi = a - bi$$

  The real part of $a + bi$ and its conjugate are the same. But the imaginary part of the conjugate of $a + bi$ is the changing sign of the imaginary part of it which is $-b$. We can try this in MATLAB:

  ```matlab
  >> (1+2i)'
  ans =
  1.0000 - 2.0000i
  ```
A 'transpose' of a complex number is a complex conjugate of this complex number.

**Transpose Complex matrices**

Now, if \( A \) is a complex matrix (each element of \( A \) is a complex number), such as

\[
A = \begin{bmatrix}
1+2i, & 2+3i; & 3i, & 3+4i
\end{bmatrix}
\]

We can try \( A' \) and see what happens:

\[
\begin{array}{c}
1.0000 + 2.0000i \\
0.0000 + 3.0000i
\end{array}
\begin{array}{c}
2.0000 + 3.0000i \\
3.0000 + 4.0000i
\end{array}
\]

We got the transpose of matrix \( A \) and each element of \( A' \) is the complex conjugate of the element of transpose of \( A \). We also can use the command `ctranspose()` to get the same result:

\[
\begin{array}{c}
1.0000 - 2.0000i \\
2.0000 - 3.0000i
\end{array}
\begin{array}{c}
0.0000 - 3.0000i \\
3.0000 - 4.0000i
\end{array}
\]

But sometimes we really need the transpose of a matrix only even it is a complex matrix, we can add a dot (.) to help us:

\[
\begin{array}{c}
1.0000 + 2.0000i \\
2.0000 + 3.0000i
\end{array}
\begin{array}{c}
0.0000 + 3.0000i \\
3.0000 + 4.0000i
\end{array}
\]

We only got the transpose of \( A \) without conjugating the elements. Also there is a command `transpose()` for this:

\[
\begin{array}{c}
1.0000 + 2.0000i \\
2.0000 + 3.0000i
\end{array}
\begin{array}{c}
0.0000 + 3.0000i \\
3.0000 + 4.0000i
\end{array}
\]
Exercises

Given a matrix $A=\text{magic}(5)$.

1. (Review of Basic Matrix Functions) Find a $5 \times 5$ matrix such that its diagonal entries equal the diagonal entries of $A$ and other elements are all 1.

2. (Review of Basic Matrix Functions) Find a $3 \times 3$ matrix such that its diagonal entries equal the vector $A(1, 1 : 3)$ and other elements are all 2.

3. (Upper and Lower Triangular Matrices) Find a $5 \times 5$ matrix of $A$ but cut the top right and bottom left elements off and set these two elements equal 1.

4. (Symmetric Matrices) Find a $5 \times 5$ symmetric matrix $C$ such that $\text{triu}(C)$ is $\text{triu}(A)$. 