APPM 2460
VECTORS & MATRICES II

1. Introduction

This week we're going to spend more time working on our ability to slice matrices (to "slice" a matrix or array means to grab certain portions of the matrix via indexing). In particular, we'll focus on performing row exchanges, and on maniuplating and plotting columns of a matrix.

2. Permuting Rows of a Matrix

We're going to first work on the example of permuting the rows of a matrix. "Permute" is a fancy word for "rearrange." We'll first work on the special case of swapping two rows. Eventually, we'll learn how to perform arbitrary permutations (i.e. make a whole bunch of row swaps at once.)

We'll need some matrices to play with. Let's use the command `magic(4)` to build a 4×4 matrix:

```
>> A = magic(4)

A =

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

To figure out what a "magic" matrix is, we can use `help magic`:

```
>> help magic
 magic  Magic square.
    magic(N) is an N-by-N matrix constructed from the integers
    1 through N^2 with equal row, column, and diagonal sums.
    Produces valid magic squares for all N > 0 except N = 2.
```

In our case, it's just some matrix we're going to play around with.

2.1. A clunky way.

Now, suppose we wanted to interchange the first and third rows of `A`. There are a few ways we could do this. An effective but somewhat clunky way is shown in the script below.

```
A = magic(4);
% make a temporary variable that holds the first row
temp_row = A(1,:);
% replace first row with third row
A(1,:) = A(3,:);
% and put variable holding first row back into the third row
A(3,:) = temp_row;
```

After running this code, try displaying the matrix `A`. You will see that we have, in fact, successfully swapped these two rows. However, this method is slow, because we need to store our first row so that it doesn't get lost when we overwrite. It's also hard to read and make sense of. Overall, not very elegant coding.

2.2. **A more elegant way.** Recall that if we have a vector `vect`, then we can use indexing to reorder its elements in place, as follows:

```
>> vect = (1:10).^2

vect =

     1     4     9    16    25    36    49    64    81   100

>> vect = vect([3 2 1 4 5 6 7 8 9 10])

vect =

     9     4     1    16    25    36    49    64    81   100
```

I simply input the indices of `vect`, in order, but with 1 and 3 interchanged. That let Matlab know which elements I wanted to put in each position. I then set `vect` to be the permuted version of itself. In this way, the original `vect` is replaced by the permuted `vect`.

We can perform arbitrary permutations in this way. If I wanted to reorder `vect` so that we saw the elements in the order $(10, 9, 2, 5, 3, 4, 6, 7, 1, 4, 8)$, I could enter

```
>> vect = (1:10).^2

vect =

     1     4     9    16    25    36    49    64    81   100

>> vect = vect([10 9 2 5 3 4 6 7 1 4 8])

vect =

   100    81     4    25     9    16    36    49     1    16    64
```

Now, let's extend this to matrices. We can work much the same way we did with vectors, except we'll have to be careful about whether we're indexing the rows or columns. So, for example, we could do:

```
>> A = magic(4)

A =

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

>> A = A([3 1 2 4],:)

A =

     9     7     6    12
    16     2     3    13
     5    11    10     8
     4    14    15     1
```

This efficiently and elegantly (one line!) swaps rows 1 and 3 of the matrix `A`. We could exchange *columns* using the command `A = A(:,[3 1 2 4])`.

3. More advanced permutations

The method above is nice, but we wouldn't want to enter the indices individually for long vectors (i.e., if I have 100 elements, I wouldn't want to enter `vect([1 3 2 4 5 6 ...])` because I'd have to type out all that crap. To make this more efficient, we can use the `end` command. For example, we can reform the earlier code that swaps the first and third element of a vector as follows:

```
>> vect = (1:10).^2

vect =

     1     4     9    16    25    36    49    64    81   100

>> vect = vect([3 2 1 4:end])

vect =

     9     4     1    16    25    36    49    64    81   100
```

We can read the last portion of this indexing as "4 through end." This would work well regardless of how large the vector `vect` is.

Now suppose we wanted had a large matrix:

```
>> A = magic(12)

A =

   144     2     3   141   140     6     7   137   136    10    11   133
    13   131   130    16    17   127   126    20    21   123   122    24
    25   119   118    28    29   115   114    32    33   111   110    36
   108    38    39   105   104    42    43   101   100    46    47    97
    96    50    51    93    92    54    55    89    88    58    59    85
    61    83    82    64    65    79    78    68    69    75    74    72
    73    71    70    76    77    67    66    80    81    63    62    84
    60    86    87    57    56    90    91    53    52    94    95    49
    48    98    99    45    44   102   103    41    40   106   107    37
   109    35    34   112   113    31    30   116   117    27    26   120
   121    23    22   124   125    19    18   128   129    15    14   132
    12   134   135     9     8   138   139     5     4   142   143     1
```

and we wanted to put the first 4 rows the matrix on the end. We know that we can do this via the command `A = A([5 6 7 8 9 10 11 12 1 2 3 4],:)`, but we'd like to be a bit slicker. The following code performs the same permutation but is much easier to read:

```
>> A([5:end 1:4], :)

ans =

    96    50    51    93    92    54    55    89    88    58    59    85
    61    83    82    64    65    79    78    68    69    75    74    72
    73    71    70    76    77    67    66    80    81    63    62    84
    60    86    87    57    56    90    91    53    52    94    95    49
    48    98    99    45    44   102   103    41    40   106   107    37
   109    35    34   112   113    31    30   116   117    27    26   120
   121    23    22   124   125    19    18   128   129    15    14   132
    12   134   135     9     8   138   139     5     4   142   143     1
   144     2     3   141   140     6     7   137   136    10    11   133
    13   131   130    16    17   127   126    20    21   123   122    24
    25   119   118    28    29   115   114    32    33   111   110    36
   108    38    39   105   104    42    43   101   100    46    47    97
```

You can easily and quickly make permutations if you make effective use of the `end` command.

4. Homework

Perform the following steps in a script (no custom functions are necessary for this assignment):

- Form a 5×5 identity matrix using the command `eye(5)`.
- Swap the first and fourth rows of the matrix. Call the result `P`.
- Form a 5×5 magic matrix using `magic(5)`. Call it `A`.
- Left multiply `A` by `P` (that is, form the product `P*A`). What is the effect on `A` of this multiplication? Compare to the original matrix `A`.
- Right multiply `A` by `P` (that is, form the product `A*P`). What is the effect on `A` of this multiplication? Compare to the original matrix `A`.