Math 236 **Problem Set 4** **Spring, 2001**

Due date *postponed*: Thurs., March 8, 2:30 p.m., LGRT 1623D mailbox

1. (Continuation of Set 3, Problem 4.) In MATHEMATICA use the graphics primitive `Line` along with any other appropriate functions to draw a figure that is *roughly* like the face on page 47—except use straight line segments instead of a circle and curved arcs. For each of the matrices in Exercises 24, 26, 28, and 30, page 47, define a MATHEMATICA function that is the linear transformation whose matrix is the given matrix. Then in MATHEMATICA calculate and plot the image of the face under that linear transformation. You may wish to use the functions `view` and `image` from notebook `View.nb`.

2. (a) Do page 61, Exercise 6.
   (b) Use your answer to (a) to do page 61, Exercise 7.

3. (a) Do page 61, Exercise 10.
   (b) Use your answer to (a) to do page 61, Exercise 11.

4. Do page 62, Exercise 24.

5. Do page 64, Exercise 44.

6. (*Counts as* three *problems.*) For this problem, you will modify my function `gaussJordan` to create a new function named `GJ`, then test the latter. You must do this as prescribed below.

   My function `gaussJordan` uses the Gauss-Jordan algorithm to return as its result the reduced row-echelon form of any numeric matrix supplied as the argument. For example:

```
   gaussJordan[Partition[{3, 2, -1, 1, 0, 6, 6, 2, 12, 0,
                          3, -2, 1, 11, 0, 0, 0, 0, 0, 0}, 5]]

{{1, 0, 0, 2, 0}, {0, 1, 0, -1, 0}, {0, 0, 1, 3, 0}, {0, 0, 0, 0, 0}}
```

   My `gaussJordan` and various functions it references are all found in the notebook `AboutGJ.nb` that you download. Among them are `swap`, `scale`, and `roundoff`. (You may use your own `swap` and `scale` provided that you are absolutely certain they are correct.)

   The *only* changes you should make to `gaussJordan` (besides changing its name to `GJ`) are designed to minimize roundoff error. These changes are:

   - Before beginning the step-by-step reduction, apply `roundoff` to change "small" entries of the matrix to zero exactly.
   - After each row scaling step and after each "zeroing-out" step, again apply `roundoff` to change small entries to zero exactly. (Note that the "zeroing-out" in a given column above and below the leading 1 is done all at once, for all rows, by means of my function `zeroUpDown` rather than one row at a time by means of `addrow`.)
   - When scaling a row, make its first nonzero entry exactly `1`. Carry this out by means of a new function `putOne` that changes the first nonzero entry to `1` exactly. Evidently, `putOne` should have as arguments simply the position of the first nonzero entry and the matrix itself (so `putOne` "figures out" what the scaling factor needs to be, uses `scale` to do the scaling, and finally changes the first nonzero entry to `1` exactly. Thus `GJ` does not use `scale` directly, but only indirectly by using `putOne`.

- (The most significant change!) Modify the Gauss-Jordan algorithm, as implemented in my `gaussJordan`, so that it now uses *partial pivoting*:

  With the usual Gauss-Jordan algorithm, you swap rows, if necessary, to bring into the pivot position (in the next nonzero column) the first nonzero entry below the pivot position. With partial pivoting, however, you swap rows to bring into the pivot position the entry *at or below* the pivot position whose *magnitude* (absolute value) is greatest among all such entries at or below the pivot position.

  To help do the partial pivoting, define a separate little function `pivotLocate` that locates the index of the requisite row to be swapped to put its first nonzero entry into the pivot position.

Keep your definitions clean, reasonably brief, and easy to read (by somebody who knows MATHEMATICA). Besides `putOne` and `pivotLocate`, which you must define, you may want to define other subsidiary functions in order to further "modularize" the program into smaller chunks.

Test separately your `putOne` and `pivotLocate` with examples of your own. Then test your entire `GJ` on some examples of your own. Do *not* turn in this testing of your own.

Turn in printouts of your `GJ`, `putOne`, `pivotLocate`, and any other subsidiary functions you happen to define. However, you will get little credit for your work unless you also ***validate*** your function `GJ` as follows. Turn in a printout of that validation.

To validate your `GJ`, be sure you have evaluated all the cells in your notebook involved in defining `GJ`. Then follow the instructions in `AboutGJ.nb` about downloading and using my encoded package `vdGJ.m`

When you evaluate the appropriate cells in `AboutGJ.nb`, the encoded package you already downloaded will automatically be read into MATHEMATICA. The package will generate some *test data*—matrices custom-made for you—which it will display. Then it will automatically evaluate your `GJ` with each test data matrix as argument and show you the result (if any).

Naturally, you should be sure your `GJ` not only completes executing without error, but also gives the correct result for each such test. If not, it's "back to the drawing board". A `GJ` that does not pass *all* the tests may receive significantly reduced credit!

Tip: Before validating, carefully design `putOne` and `pivotLocate` (and any other auxiliary functions you want) and revise `gaussJordan`; then manually test each one yourself, paying special attention to unusual cases for input. Don't run the validator the first moment you think you *might* have something that works! Running the validator over and over again while trying to debug your functions is likely to waste a lot of time and effort!