CS4 Matrix Project

Due: 11:59 P, Wednesday, April 25

Contents

1 Partners

This is a pair programming project, please fill out the **Matrix Partner Signup form** by 11:59pm Sunday, April 15th. The signup link is also here:

`https://goo.gl/X9LpPh`

Filling this out will allow us to process your grades correctly.

As you work with your partner, you must use the pair programming technique. See the CS4 pair programming guide for all the details:

`http://cs.brown.edu/courses/cs004/CS4_Pair_Programming_Guide.pdf`

If you are having difficulty finding a partner, we recommend that you use the "Search for Teammates" post on the CS4 Piazza site as a way to reach out to your classmates.

2 Introduction

The project consists of three parts. In the first part (through section 5), you will implement several functions that manipulate matrices, in the second part (sections 6 and 7) you will create a function named `myGJSolver` that uses the Gauss-Jordan elimination algorithm to convert a matrix into reduced row echelon form. With this function in hand, you will then create several additional tools for working with systems of linear equations.

Coding the Gauss-Jordan elimination algorithm and finding Null Spaces are very challenging tasks for many students, we suggest you start early so that you can complete the project in full.

In the third part of this project (section 8), you will write two short functions and answer some questions about Markov Chains.

To download this project, open a Brown CS terminal window and run

```
cs4_install matrix
```

2.1 Design Recipe & Test Scripts

The CS4 Design Recipe requires you to create a set of test cases before you start to code your function.

Since, since the code you will be writing for this project needs to be fairly general, and covers an area of mathematics some of you may not be familiar with, we have provided you two pre-built testing scripts.

For your `myGJSolver` function. This script is called

`myGJSolver_test.m`

It relies on two other files we have provided,

`myGJSolver_test_solution.m`

`myGJSolver_test_no_solution.m`

You do not need to call either of them directly.

We have also provided you with a test script template called `myXXX_tests.m` which you should modify to include tests for all of your other functions.

The myXXX_tests.m test script just sets up various A's and y's. The code then checks if the performance is correct in each case.

You should add some oddly shaped matrices (focus on the underdetermined $m < n$ cases first) and expand the tests to accommodate them.

The goal here is to make sure your code is working for different sized matrices and matrices with different ranks.

You are also welcome to just code up individual cases outside of the loop.

Note: since many of the test cases involve numeric matrix and vector results, it is often a good idea check to see whether or not they are within some tolerance of what is expected. Below is an example using the L_2 norm (described below) that compares two vectors. You can use the same approach for comparing matrices.

```
assert(myNorm(A-B)<1e-10, 'Test 4) A is not equal to B!')
```

2.2 Partial Credit

If you can't make your functions work for all the cases that they should, you will receive partial credit for the ones that they do work with.

You will also receive additional points for documenting the cases for which your functions do not work. Include this information as a comments in the appropriate sections of `myXXX_tests.m` Include an example of the inputs and outputs for which the function fails. You should also add a similar section for `myGJSolver` if you need to document any shortcomings with that function.

2.3 Permitted Resources

MATLAB comes with an extensive library of built-in linear algebra functions, like `mldivide, rref, det, inv, rank, orth, null`. However, for this project, we will start from first principles and develop all the matrix manipulation functions we use from scratch. Doing so will help develop intuition as to what each function does, and how it does it.

While working on this project you are encouraged to talk to other students about the problems and how to solve them, however, you are not allowed to examine one another's code or share it. Really.

In the same spirit (and in accordance to the course collaboration policy), you are NOT allowed to inspect or copy MATLAB code that you find on the web, or within MATLAB itself. You are however encouraged to use Wikipedia and other resources (like Khan academy) to learn more about Gaussian elimination, Gauss-Jordan elimination.

`https://en.wikipedia.org/wiki/Gaussian_elimination`

Other aspects of linear algebra (like determinants) are also fair game, if you need help with them. Here is a nice video about determinants.

`https://www.youtube.com/watch?v=Ip3X9LOh2dk`)

Note: The Brown CS department courses regularly makes use of software designed to detect plagiarism, both between students and with respect to online code. The software builds a model of each submitted program's structure that is invariant with respect to changes in variable names and code re-ordering, i.e., it is very good at identifying when copying has occurred.

3 Setup and Hand-In

Run the command

```
cs4_install matrix
```

to setup the project directory and stencil code for this project. Below is a list of files that you should be handing in:

- `myNorm.m`
- `myNormalize.m`
- `myDot.m`
- `myMatrixMult.m`
- `myTranspose.m`
- `myIsSymmetric.m`
- `myGJSolver.m`
- `myDeterminant.m`
- `myInverse.m`
- `myRank.m`
- `myRange.m`

- `myNull.m`
- `myXXXX_tests.m`
- `myWeatherReport.m`
- `markov.m`

You are also welcome to include `myGJSolver_test.m` and its two support files.

To hand in this project, move all of your solution files into the matrix directory, `cd` into the directory (use `ls` to make sure all of your files are there!) and run

```
cs4_handin matrix
```

4 What Is a Vector? (5)

A vector is a list of numbers. For example, $(1, 2)$ or $(5, 6, 4, 3, 2, 2, 5)$ are vectors. Vectors can be used for many applications. We can think of a 2-element vector as the (x, y) coordinates of a point in a 2D coordinate plane, or a 3-element vector as the (x, y, z) coordinates of a point in the 3D coordinate plane. Similarly, we can conceptualize an n-element vector as the $(x_1, x_2, \cdots, x_n)$ coordinates of a point in an n-dimensional space. In MATLAB, vectors are are always stored in either row $(1 \times n)$ or column $(n \times 1)$ format.

4.1 Euclidean Norm

The norm (aka length) of an n-element vector (for our purposes) will be the L_2 norm (aka Euclidean norm) defined as the square root of the sum of the squares of the elements in the vector:

$$\|x\| = \sqrt{\sum_{i=1}^{n} x_i^2}$$

A unit vector is a vector with a norm of 1. To convert a vector into a unit vector, we divide each element in the vector by that vector's norm. This is known as normalizing the vector, i.e., $x/\|x\|$ is a unit vector that points in the same direction as x.

The L_2 norm is also used to study matrices. It is calculated by summing the squared elements of the matrix and then taking the square root.

Problem 2.1: L_2 Norm

Implement the calculation of the vector norm and the normalization of a vector by filling out the `myNorm` and `myNormalize` functions. The `myNorm` function should accept a *vector or matrix* and return its L_2 norm, and the `myNormalize` function should accept a vector and return a normalized version of that vector. Since a zero vector cannot be normalized, your `myNormalize` function should return an empty vector in this case. Your `myNormalize` function MUST use your `myNorm` function.

MATLAB has a built-in function `norm()`, which you can use as an additional check of your work, but obviously, do not use it in your function implementations. Also it behaves differently for matrices than it does for vectors - read the doc!

4.2 The Dot Product

Vectors are easy to add and subtract. The sum of two vectors is

$$(x_1, x_2, ...x_n) + (y_1, y_2, ...y_n) = (x_1 + y_1, x_2 + y_2, ...x_n + y_n)$$

and the difference is defined in a similar way. However, there is more than one way to multiply two vectors. One such way is the dot product of the vectors. The dot product is the sum of the products of each element in the vectors. The dot product can be expressed as

$$(x_1, x_2, \cdot, x_n) \cdot (y_1, y_2, \cdots, y_n) = \sum_{i=1}^{n} x_i y_i$$

Note that the dot product is only defined for vectors of the same length, and that the dot product of 2 vectors is a scalar (a single number).

Problem 2.2: Dot Product

Implement the dot product by creating a `myDot` function. The `myDot` function should accept 2 vectors and return the dot product of those vectors. If the input vectors are of unequal length, your function should return an empty vector.

MATLAB has a built-in function `dot()`, which you can use as an additional check of your work, but obviously do not use it in your function implementation.

5 What is a Matrix? (5)

A matrix is two-dimensional array of numbers. For example,

$$\begin{pmatrix} 1 & 5 & 1 \\ 2 & 2 & 4 \\ 0 & 1 & 0 \end{pmatrix}$$

or

$$\begin{pmatrix} 1 & 5 & 1 \\ 2 & 2 & 4 \end{pmatrix}$$

are matrices. Matrices have a wide variety of applications, such as solving systems of equations, storing data, and applying functions. Vectors are special cases of matrices. A **column vector** is a matrix with a single column. A **row vector** is a matrix with a single row. A matrix with the same number of rows as columns is a **square matrix**.

5.1 Matrix Multiplication

Like vectors, matrices are easy to add and subtract. For example,

$$\begin{pmatrix} 1 & 4 & 1 \\ 2 & 4 & 4 \\ 0 & 1 & 8 \end{pmatrix} + \begin{pmatrix} 0 & 5 & 2 \\ 2 & 6 & 4 \\ 1 & 2 & 6 \end{pmatrix} = \begin{pmatrix} 1 & 9 & 3 \\ 4 & 10 & 8 \\ 1 & 3 & 14 \end{pmatrix}$$

Also like vectors, only matrices with identical dimensions (the same number of rows and columns) can be subtracted or added to each other. However, matrix multiplication (as opposed to element-wise multiplication) is more complicated. For matrices A and B, multiplying AB requires the number of columns in A to match the number of rows in B. Note: AB is also sometimes written $A*B$.

$$\begin{pmatrix} 1 & 4 & 1 \\ 2 & 4 & 4 \end{pmatrix} \begin{pmatrix} 0 & 5 \\ 2 & 6 \\ 1 & 2 \end{pmatrix}$$

is defined, but the product

$$\begin{pmatrix} 0 & 5 \\ 2 & 6 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} 1 & 4 & 1 \\ 2 & 4 & 4 \\ 2 & 4 & 6 \end{pmatrix}$$

is not defined.

Formally, the matrix product AB, where A is $m \times n$ and B is $n \times p$ is as follows:

$$(AB)_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j}$$

AB is an $m \times p$ matrix.

Note: The above equation is equivalent to the dot product of the vector formed from row i of A and the vector formed from column j of B. Therefore, we can also define element (i, j) of the matrix product AB as the dot product of the i^{th} row of A and j^{th} column of B.

Also note that the multiplication of a row vector with a column vector is a dot product.

$$\begin{pmatrix} 0 & 5 & 5 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 2 \end{pmatrix} = 20$$

Also, note that matrix multiplication is not commutative:

$$\begin{pmatrix} 1 \\ 2 \\ 2 \end{pmatrix} \begin{pmatrix} 0 & 5 & 5 \end{pmatrix} = \begin{pmatrix} 0 & 5 & 5 \\ 0 & 10 & 10 \\ 0 & 10 & 10 \end{pmatrix}$$

Here we see in fact that the product of column vector and row vector is a matrix. Order matters!

Problem 2.3: Matrix Multiplication

Implement matrix multiplication by creating a `myMatrixMult` function. This function should accept 2 matrices and return their product. Your function should use your `myDot` function. If the product of the input matrices does not exist, your function should return an empty matrix. If the matrices are of incompatible size, your function should return an empty matrix `[]`.

Note: In MATLAB, the product of 2 matrices A and B is simply A*B, using the asterisk symbol to indicate multiplication. You may use this as an additional check of your work. Obviously, do not use this in your function implementation.

Before you write any code, try to compute the following products by hand and check your work in MATLAB in order to ensure that you understand the algorithm.

$$\begin{pmatrix} 1 & 6 & 0 \\ 0 & 4 & 4 \\ 0 & 2 & 2 \end{pmatrix} \begin{pmatrix} 5 & 6 & 3 \\ 0.5 & 8 & 1 \\ 1 & 3 & 14 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 16 & 1 \end{pmatrix} \begin{pmatrix} 1 & 6 & 0 \\ 5 & 3 & 1 \\ 0 & 2 & 8 \end{pmatrix}$$

5.2 Matrix Transpose

The transpose of a matrix A (denoted A^t) is the matrix formed by switching the rows and columns. That is, element $(A^t)_{i,j} = A_{j,i}$. For example,

$$\begin{pmatrix} 1 & 4 & 1 \\ 2 & 4 & 7 \\ 2 & 8 & 4 \end{pmatrix}^t = \begin{pmatrix} 1 & 2 & 2 \\ 4 & 4 & 8 \\ 1 & 7 & 4 \end{pmatrix}$$

A matrix that is equal to its own transpose is called a **symmetric matrix**.

Problem 2.4: Transpose

Implement the functions `myTranspose` and `myIsSymmetric`. `myTranspose` should accept a matrix as input and return its transpose, and `myIsSymmetric` should accept a matrix as input and return true (1) if the matrix is symmetric and false (0) otherwise.

Note: The conjugate transpose of a matrix A in MATLAB is A'. You should use this to check your work. Obviously, do not use this command in your function implementation. (Note: The conjugate transpose same thing as transpose when applied a real valued matrices like we are dealing with here).

5.3 The Identity Matrix

The **Identity Matrix** is a square matrix with ones along the diagonal and zeros everywhere else. For example,

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, (1)$$

are 3-by-3, 2-by-2 and 1-by-1 identity matrices. When we multiply a matrix (of the correct size) by the identity matrix, the matrix does not change. That is, $AI = A$ and $IA = A$. To construct an $n \times n$ identity matrix in MATLAB, you can use the command `eye(n)`.

6 Gauss-Jordan Elimination (60)

As discussed in lecture, the Gauss-Jordan elimination algorithm converts a matrix to reduced row echelon form (RREF).

When applied to an augmented matrix $[AY]$, the RREF allows one to easily ascertain whether or not a solution exists for a system of linear equations associated with the system $Ax = Y$. When solution(s) do exist, the RREF provides

an easy way to read one off (i.e. determine a value of x such that $Ax = Y$). The RREF also provides information on the rank, range and null space of a matrix. As the Gauss-Jordan algorithm operates it is also possible to find the determinant of a matrix.

Here is a link to a helpful article on Gauss-Jordan Elimination:
`https://en.wikipedia.org/wiki/Gaussian_elimination`

In the next set of problems you will write a general linear equation solver based on Gauss-Jordan elimination, and then create additional functions that use it to find matrix determinants, inverses, ranks, ranges and null spaces.

Problem 2.5: Gauss-Jordan Solver

Create a function called `myGJSolver` that solves linear equations of the form $AX = Y$. Your function should return a solution, `X`, the reduced row echelon form, `RREF`, of matrix A and the determinant, `D`, of A. The signature should be

```
function [X, RREF, D] = myGJSolver(A, Y)
```

When there is no solution to $AX = Y$, return the empty array for `X`. When there is no determinant (i.e., `A` is not square) return an empty set for `D`. If Y is an empty matrix, return RREF and D as usual, but return an empty array for X. (Note: Your RREF should always match the `rref(A)`

Note: The MATLAB function `rref` produces reduced row echelon matrices. You may use it to check your work, but you may not use it, or its implementation in your design of myGJSolver.

This is a fairly complex function to code. Below are some suggestions and further information.

- Begin by coding the basic RREF algorithm, and verify you solutions match those of MATLAB's `rref` function. After that add support for computing determinants, and verify they are correct. Once that is working add support for solving equations.
- Be sure your function does not work only on square matrices. Also consider cases where there are infinite solutions or a solution does not exist
- When Y has multiple columns, i.e., when Y has n columns – i.e. $Y = [y_1, y_2, ..., y_n]$, the returned X should also have n columns – i.e. $X = [x_1, x_2, ..., x_n]$, where $Ax_1 = y_1, Ax_2 = y_2, ...Ax_n = y_n$.
- For each row in an row echelon form, if the row is not all zeros, then the left-most non-zero entry is called the pivot element (or leading coefficient) of that row. If you run into nonsensical results, or numerical stability issues, as you construct the RREF, select the row with largest possible pivot element, and use it to zero out the other elements in its column.

You may also want to ignore (or even zero out) elements smaller than 1e-10.

- Use array indexing to operate on entire rows in order to make your code cleaner and run faster. The find command is also very helpful for determining where non-zero entries are, etc.
- Hint: Your solver should handle undetermined systems (`rank(A)<m`)

```
>> A
A =
     1     1     0
     0     0     1
     0     0     0
>> y
y =
     1     1
     0     1
     0     0
```

In this example, $Ax = y(:, 2)$ corresponds to the equations

```
x(1)+x(2) = 1
     x(3) = 1
```

Since the A is in RREF form it is OK to set the slack variables in a row (all variables following the pivot) zero. So, one particular solution to this system is x(1) = 1, x(2) = 0 (from the first row of A) and x(3)=1 (from the second row, third column of A)

In general the rule you need to use is as follows: initialize X to zero, and the assign every pivot variable the corresponding value it refers to in y. A pivot in the ith row and jth column of A refers to x(j) and should be assigned the value y(i).

6.1 Matrix Determinants & Inverses

Problem 2.6: Determinants

Create a `myDeterminant` function that computes the determinant of a matrix A. When no determinant exists (because the matrix is not square) `myDeterminant` should return an empty matrix. Implement this functions with a call to `myGJSolver` (this is a very short function!).

Problem 2.7: Inverses

Create a `myInverse` function that returns the inverse of a matrix A. If A^{-1} does not exist (i.e., A is not invertible or isn't square) your function should return an empty matrix. Your implementation should use `myGJSolver` to find the solution of $AX = I$, where I is an identity matrix the same size as A.

6.2 General Solutions to Linear Equations

The general solution to a set of linear equations $Ax = y$ consists of a particular solution x_0 to $Ax = y$, plus any element from null space of A, i.e., for an m-by-n matrix A,

$$G = \{g : g = x_0 + z, z \in range(null(A))\}$$

The following three functions are very useful for understanding linear functions and equations systems, and for creation of general solutions.

6.3 Rank and Range

The rank of a matrix A is the dimension of the vector space generated (or spanned) by its columns.

Problem 2.8: Ranks

Create a `myRank` function that returns the rank of a matrix. Use the `RREF` returned by your `myGJSolver` to do this, i.e. use the `RREF` returned by `myGJSolver(A,[])`, and count the number of rows with leading 1s.

Hint: You can determine the rank of your matrix as you are converting it to a RREF. You'll need to define a variable that you update appropriately as the RREF is calculated.

Problem 2.9: Ranges

Each column of the RREF that contains a pivot element forms a linearly independent set of columns in both the RREF and the original matrix A. For example,

$$rref(A) = \begin{pmatrix} 1 & 0 & 3 & 0 \\ 0 & 1 & 4 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

has pivot elements in columns 1, 2 and 4. Therefore columns 1, 2 and 4 of A form a basis for the range of A:

$$range(A) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

Create a `myRange` function that accepts a matrix A and returns a matrix of linearly independent column vectors that span the range of A (i.e., a basis for the range of A).

One challenge with developing bases is that they are not unique. However, one can use rank to test whether or not two bases are the same. One approach is to check the dimension and span, i.e.

```
R = myRange(A);
assert(size(R,2)==myRank(A), 'myRank dimension problem')
assert(myRank([R A])==myRank(A), 'myRank span problem')
```

Note: The MATLAB `orth` function returns an orthonormal basis for a matrix. You may use it to check your work, but you may not use it, or it's implementation, in your design.

7 Null Spaces (15)

Problem 2.10: Null Spaces

The null space of a matrix A consists of all vectors x such that $Ax = 0$. The basis for the null space of a m-by-n matrix, consists of $(n - r)$ column vectors, where $r = rank(A)$.

The RREF provides a slick way to find these vectors.

Swapping columns i and j of the RREF yields a system equivalent to the original system, where the locations of variables x_i and x_j have been exchanged. Permuting (rearranging) the columns of the RREF so that the upper left corner contains the identity matrix, and all other rows below that (if there are any) consist entirely of zeros, allows one to easily determine the null space of the system using the following construction:

For a m-by-n, rank-r RREF of A, the rearranged form R will have the following structure:

$$R = \begin{pmatrix} I_1 & U \\ 0 & 0 \end{pmatrix}$$

Where I_1 is a r-by-r identity matrix and U is a r-by-(m-r) matrix whose elements are coefficients of the free variables in the system.

Columns of the matrix Z below form a a basis for the null space of R.

$$Z = \begin{pmatrix} -U \\ I_2 \end{pmatrix}$$

Where I_2 in this case is a (n-r)-by-(n-r) identity matrix. (To verify that Z is basis for the null space, multiply R times Z, and check that the rank of Z is m-r.)

Unpermuting the columns of R and corresponding rows of Z yields the original RREF A and a basis for the null space of the original system. Below is an example:

$$rref(A) = \begin{pmatrix} 1 & 0 & 3 & 0 \\ 0 & 1 & 4 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

In this example, $m = 4, r = 3$. We interchange columns 3 and 4 in order to get an the largest identity matrix (of size r-by-r) in the top left corner. This yields:

$$R = \begin{pmatrix} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 4 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Which mean that for this case,

$$I_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

and

$$U = \begin{pmatrix} 3 \\ 4 \\ 0 \end{pmatrix}$$

We must now form the Z matrix:

$$Z = \begin{pmatrix} -3 \\ -4 \\ 0 \\ 1 \end{pmatrix}$$

I_2 in the Z matrix is equivalent to a (n-r)-by-(n-r) identity matrix. In this case, I_2 is a 1-by-1 identity matrix, which is simply (1). The top portion of Z is the negation of U.

Since we swapped columns 3 and 4 when originally forming matrix R, we must now swap rows 3 and 4 of Z to get the null space of A. Therefore, the null space of A is spanned by the columns of:

$$Z = \begin{pmatrix} -3 \\ -4 \\ 1 \\ 0 \end{pmatrix}$$

Create a `myNull` function that accepts a matrix A and returns a matrix of linearly independent column vectors that span the null space of A. When A is of full rank (i.e. for A m-by-n, rank(a)=n) return an empty set.

Implement your function using information returned in the `RREF` of `myGJSolver(A, [])`.

Note: The matlab function `null` returns a basis for the null space of a matrix. You can try and use this function to check your work, but do not use it, or it's implementation in your own solution.

In general, null(A) and your myNull(A) function may return different things, but that's OK.

One way to test myNull(A) is to check that A*myNull(A) gives a zero matrix and that the rank of myNull(A) is m-rank(A), where A is an m-by-n matrix.

8 Markov Chains (15)

Place all answers to this section inside a script titled `markov.m`, or in the appropriate function file if specified.

Hint: In general vectors (rows or columns) of probabilities for a action (say transitioning from a state i to each possible state j) should always sum one. Probabilities are between 0 and 1 inclusive.

8.1 Create a Markov Model

A transition probability matrix can be used to represent changes to the weather from a given day to the next. We will assume there are four types of possible weather in CS4 world: sunny, cloudy, rainy, and snowy. Say that the following probabilities exist:

- If it is sunny today, there is a 60% chance it will be sunny tomorrow, a 10% chance it will be rainy tomorrow, a 20% chance it will be cloudy tomorrow, and a 10% chance it will be snowy tomorrow.
- If it is rainy today, there is a 30% chance it will be sunny tomorrow, a 20% chance it will be rainy tomorrow, a 30% chance it will be cloudy tomorrow, and a 20% chance it will be snowy tomorrow.
- If it is cloudy today, there is a 20% chance it will be sunny tomorrow, a 30% chance it will be rainy tomorrow, a 30% chance it will be cloudy tomorrow, and a 20% chance it will be snowy tomorrow.
- If it is snowy today, there is a 10% chance it will be sunny tomorrow, a 30% chance it will be rainy tomorrow, a 30% chance it will be cloudy tomorrow, and a 30% chance it will be snowy tomorrow.

Problem 2.11 Model Use

a) Create a cell array `s` which the names of each state in this system. Create a transition matrix (TPM) `M` for the weather of CS4 world, where element `M(i,j)` corresponds to the probability of moving to state `s(i)` from state `s(j)`.

b) Say that today is Monday, and it is definitely sunny today. That is, today's state vector v is:

$$v = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Calculate Tuesday's (tomorrow's) state vector.

c) Create a `myWeatherReport(s,M,v,n)` function that takes in state name vector s, TPM M, state vector v, the number of days in the future n, and prints out a forecast as it were a weatherperson on TV (i.e. "In 2 days there is a 20% chance of rain, a 50% chance of sun, a 30% chance of clouds and no chance of snow"). Your function should also return the state vector for n days in the future.

HINTS: Use `sprintf` to form the string. When using sprintf and fprintf you need to use %% to include a single % sign inside of a string. But, since will then use this string again in fprintf you will need to use %%%% for each % that is to be printed.

Some sample output is shown below:

```
>> v = [1 0 0 0]';
>> [str, ~] = myWeatherReport(s,M,v,4);
>> fprintf(str)

In 4 days there is a
 35.36% chance of sunny, a
 20.49% chance of rainy, a
 26.22% chance of cloudy, a
 17.93% chance of snowy.


>> [str, ~] = myWeatherReport({'bull market', 'flat market', ...
'declining market', 'bear market'},M,v,2);
>> fprintf(str)

In 2 days there is a
 44.00% chance of bull market, a
 17.00% chance of flat market, a
 24.00% chance of declining market, a
 15.00% chance of bear market.
```

d) What will Thursday's state vector be? Use `myWeatherReport` to print out your forecast.

Problem 2.11.1 Yesterday's Weather

For this problem and the next, assume today's state vector v is:

$$v = \begin{pmatrix} 0.295 \\ 0.230 \\ 0.270 \\ 0.205 \end{pmatrix}$$

What was yesterday's state vector? Given yesterday's weather what was the forecast for today? Use your `myWeatherReport` function and `fprintf` to print it out.

HINT: Solve $Ax = b$.

Create a file called `markov.m` and include appropriate code and comments to answer a),b),c) and d) above.

Problem 2.12 Predicting the Weather

Add your comments for this problem to `markov.m`.

What will the state vector describing the weather be in 10 days? How about 20 days? 30 days? How do the results look? Why do you think this happened? Does it matter what the initial state vector is?

HINT: Examine M at each of these powers.

More information on steady state phenomena can be found on this Wikipedia page about Markov Chains: `http://en.wikipedia.org/wiki/Markov_chain#Steady-state_analysis_and_limiting_distributions`

Problem 2.13 Non-Steady State

Can you think of a 4 x 4 transition matrix and an initial state vector that never converge to a stationary vector?

Include an example TPM and state vector and some tests and commentary in `markov.m` that show that your system does not have a stationary vector.