**Programmer’s Guide**

Welcome to George and Sean’s Matrix Algebra Package!

This package utilizes Python 3.3 and revolves around a Matrix() class. This class must be initialized with a 2D list representing a matrix. If the incoming 2D list doesn’t correspond to a normal rectangular matrix, it will be ‘normalized’, with the missing slots in the matrix replaced by zeroes. For instance:

>>> M = Matrix( [ [2,5], [1,3], [7,1,3] ] )

>>> M

[[2, 5, 0], [1, 3, 0], [7, 1, 3]]

There are many methods associated with the matrix class, which allow various operations.

**self.prettyprint()**

This outputs the matrix in a familiar table format, for easier viewing.

>>> M = Matrix( [ [2,5], [1,3] ] )

>>> M.prettyprint()

2.00 5.00

1.00 3.00

**\_\_add\_\_ (+)**

Adding matrices is implemented by a simple list comprehension.

>>> M = Matrix( [ [2,5], [1,3] ] )

>>> B = Matrix( [ [1,6], [2,4] ] )

>>> M + B

[[3,11], [3,7]]

**\_\_sub\_\_ (-)**

Subtracting matrices is also implemented by simple list comprehension

>>> M = Matrix( [ [2,5], [1,3] ] )

>>> B = Matrix( [ [1,6], [2,4] ] )

>>> M - B

[[1,-1], [-1,-1]]

**\_\_mul\_\_**

Multiplying matrices is also implemented.

>>> M = Matrix( [ [2,5], [1,3] ] )

>>> A = Matrix( [ [1,6,3], [2,1,4] ] )

>>> M \* B

[[12, 17, 26], [7, 9, 15]]

**\_\_rmul\_\_ (\*)**

Originally, scalar multiplication was implemented by a scalarMult() method, but this \_\_rmul\_\_ method works more elegantly. When Python interprets two objects being multiplied, if they are both of the same type, the \_\_mul\_\_() method is used. However, if they are of different types, \_\_rmul\_\_ is used. Therefore, if a Matrix() object is multiplied by a scalar, \_\_rmul\_\_ will run and not \_\_mul\_\_.

In this package, the scalar must always be placed before the matrix object in the multiplication.

>>> M = Matrix( [ [2,5], [1,3] ] )

>>> 5 \* M

[[10, 25], [5, 15]]

**\_\_pow\_\_ (\*\*)**

Raising to a power is implemented with a loop and reference to \_\_mul\_\_.

>>> M = Matrix( [ [2,5], [1,3] ] )

>>> M \*\* 3

[[43,120], [24,67]]

**self.trans()**

This method transposes the matrix in place, modifying the original matrix.

>>> M = Matrix( [ [2,5], [1,3] ] )

>>> M.trans()

>>> M.prettyprint()

2.00 1.00

5.00 3.00

**self.getTrans()**

Makes a copy of a Matrix object.

>>> M = Matrix( [ [2,5], [1,3] ] )

>>> M.getTrans()

[[2,1], [5,3]]

**self.det (property)**

This is a property of a Matrix object that returns the matrix’s determinant. This method is implemented by recursion and reference to a helper subMatrix() function that slices a matrix by removing a row and a column. The resulting code is very elegant and works like a charm.

>>> M = Matrix( [ [2,5], [1,3] ] )

>>> M.det

1

**self.isInvertible (property)**

This is a property of a Matrix object that returns True if the matrix is invertible, i.e., if the determinant is not zero and if the matrix is square.

>>> M = Matrix( [ [2,5], [1,3] ] )

>>> M.isInvertible

True

**self.cofactors ()**

This returns a copy of the cofactor matrix of a Matrix object.

>>> M = Matrix( [ [2,5], [1,3] ] )

>>> a = M.cofactors()

>>> a.prettyprint()

3.0 -1.0

-5.0 2.0

**self.inverse()**

This inverses a Matrix object in place. It’s implemented by referencing the .det property, the .cofactors() method and the .getTrans() method, making for a very elegant translation into code from math.

>>> M = Matrix( [ [2,5], [1,3] ] )

>>> M.inverse()

>>> M.prettyprint()

3.00 -5.00

-1.00 2.00

**self.getInverse()**

This returns a copy of the inverse of a Matrix object.

>>> M = Matrix( [ [2,5], [1,3] ] )

>>> M.getInverse()

[[3,-5], [-1,2]]

**self.rowEchelon()**

This reduce a matrix to row echelon form in place.The code is inspired from <http://en.wikipedia.org/wiki/Gaussian_elimination#Pseudocode> . It references a helper swapRows() method.

>>> M = Matrix( [ [2,5], [1,3] ] )

>>> M.rowEchelon()

>>> M.prettyprint()

2.0 5.0

0.0 0.5

**self.reduce()**

This reduces a matrix object to reduced row echelon form. It first calls .rowEchelon() and then reduces the matrix to reduced row echelon form. It does this by cycling through each row of the matrix, from the bottom up, and looking for leading coefficients. Once a leading coefficient is found, corresponding multiples of that row are subtracted from other rows by using the helper rowAdd() method.

>>> M = Matrix( [ [2,5], [1,3] ] )

>>> M.reduce()

>>> M.prettyprint()

1.0 0.0

0.0 1.0

**self.append(other)**

This appends a matrix ‘other’ to another matrix ‘self’, creating a new extended matrix.

This method will change the original matrix ‘self’, meaning that the original matrix will be modified. The code works really well with Python’s built-in list addition.

>>> M = Matrix( [ [2,5], [1,3] ] )

>>> A = Matrix( [ [1,6,3], [2,1,4] ] )

>>> M.append(A)

>>> M.prettyprint()

2.0 5.0 1.0 6.0 3.0

1.0 3.0 2.0 1.0 4.0

**self.subMatrix(row, column)**

This returns a copy of a submatrix of the original matrix, with a specified row and column removed. The default for row and column are both -1, meaning no row or column is removed. Therefore, if the user wishes, they can only remove a column and not a row or a row and not a column. This function makes the code for the determinant method much simpler and more elegant. Note that the column and row indices start at 0, meaning the ‘first’ row/column is row/column 0.

>>> Q = Matrix( [ [3,4,2], [4,7,8] ] )

>>> Q.prettyprint()

3.0 4.0 2.0

4.0 7.0 8.0

7.0 3.0 9.0

>>> Q.subMatrix(1,1).prettyprint() # Column 1 and row 1 removed

3.0 2.0

7.0 9.0

**identity(n)**

A fast way of creating an identity matrix of size n. Implemented through simple list comprehension.

>>> identity(3)

[[1,0,0], [0,1,0], [0,0,1]]

**MARKOV CHAINS**

An application to Markov Chains can be seen in the ‘Markov Chains Test Run.py’ file. Open the file for details.