In [2]:
```
import numpy as np
import copy
```

1. Introduction

This Jupyter Notebook will provide a an explanation of the functions necessary to complete Gauss elimination and back substitution. It is also important to understand LU factorization. After explaining these procedures, this document will provide examples of how to solve systems of equations by using LU factorization, Gauss elimination, and back substitution.

Written by Thomas Moawad

1.1 Common Functions

1.1.1 Row Replacement

Row replacements help us to replace r_i with

This function enables us to perform any operation of the form r_i_new = c * r_i - k * r_j where c and k are constains

For example, suppose we have

[[1, 1, 1],

[2, 2, 2],

[3, 3, 3]]

We can say r_1 = 100 * r_0 - 2 * r_1 such that we now have

[96, 96, 96],

[2, 2, 2],

[3, 3, 3]

In [3]:

```
"""
General function to perform a row replacement of the form: r_i = c * r_

Args:
    matrix: numpy matrix of dimensions m rows by n columns
    c: The coefficient to multiply r_i by (optional by defualt is 1)
    r_i: The index of the row we are applying the operation to
    k: The coefficient to multiply r_j by (optional by defualt is 0)
    r_j: The index of the row we are adding to r_i (optional by defualt

Returns:
    The matrix with the applied row replacement such that: r_i = c * r_
"""
def row_replacement(matrix, r_i, c = 1, r_j = 0, k = 0):
    matrix2 = copy.deepcopy(matrix)

    matrix2[r_i] = c * matrix2[r_i] - k * matrix2[r_j]
    return matrix2
```

Example of Row Replacement

In [4]:

```
matrix = np.matrix([[1, 1, 1],
                    [2, 2, 2],
                    [3, 3, 3]])

row_replacement(matrix, 0, 100, 1, 2)
```

Out[4]:

```
matrix([[96, 96, 96],
        [ 2,  2,  2],
        [ 3,  3,  3]])
```

1.2 Row Swap

Another basic operation we learned is row swapping. This is just as essential to Gauss Elimination as Row Operation is.

In [5]:

```
"""
General function to perform a row swaps

Args:
    matrix: numpy matrix of dimensions m rows by n columns
    r_i: The index of the row to swap with r_j
    r_j: The index of the row to swap with r_i

Returns:
    The matrix with the corresponding row swap
"""
def row_swap(matrix, r_i, r_j):
    matrix2 = copy.deepcopy(matrix)
    matrix2[[r_i, r_j]] = matrix[[r_j, r_i]]
    return matrix2
```

Example Row Swap

In [6]:
```
matrix = np.array([[4,3,1],
                   [5,7,0],
                   [9,9,3]])
row_swap(matrix, 0, 2)
```

Out[6]:
```
array([[9, 9, 3],
       [5, 7, 0],
       [4, 3, 1]])
```

1.3 Get Smaller Dimension

Another basic function we need is a way to get the smaller dimension. This is just an if-else statement but it will be useful later on.

Given a matrix with dimensions m by n, if m is smaller then return m, else return n

In [7]:
```
"""
Obtain the smaller dimension of the matrix. Either

Args:
    matrix: numpy matrix of dimensions m rows by n columns

Returns:
    The smaller dimension of the matrix
"""
def get_smaller_dimension(matrix):
    if len(matrix) < len(matrix.T):
        return len(matrix)
    else:
        return len(matrix.T)
```

Examples of Getting the Smaller Dimension

In [8]:
```
# 2 by 3 matrix. m is smaller
matrix = np.matrix([[1, 1, 1], [2, 2, 2]])
get_smaller_dimension(matrix)
```

Out[8]: 2

In [9]:
```
# 4 by 3 matrix. n is smaller
matrix = np.matrix([[1, 1, 1],
                    [2, 2, 2],
                    [3, 3, 3],
                    [4, 4, 4]])
get_smaller_dimension(matrix)
```

Out[9]: 3

1.4 Expanded Dot Product

Will write the dot product in expanded form as a string. This can later be evaluated using the python built in eval(string) function

In [10]:

```python
def expanded_dot(row_i, row_j):
    row_i = np.matrix(row_i)
    row_j = np.matrix(row_j)

    string_dot_product = "(" + str(row_i.item(0)) + " * " + str(row_j.i

    for i in range(1, len(row_j.T), 1):
        string_dot_product += "+ (" + str(row_i.item(i)) + " * " + str(

    try:
        return eval(string_dot_product)
    except:
        return string_dot_product
```

2: Gauss Elimination

Gauss elimination is important because it produces an upper triangular matrix.This triangular matrix will become useful for LU factorization and for Back Substitution.

Upper Triangular Matrix: A matrix U of size m by n such that for each column_i, all elements between U[i+1][i] and U[m][i] (inclusive) are equal to 0

In [11]:

```python
def gauss_elimination(matrix):
    matrix = matrix.astype(float)
    U = copy.deepcopy(matrix)

    for i in range(len(U.T)): # for each column
        U = zeros_underneath(U, i, i) #make everything 0 underneath the

    return U
```

2.1 Making Zeros Under the Diagonal

We just defined an upper triangular matrix. In this section we will explain how that is actually found. There are two functions for this to be done

```
1. We can use a row replacement
2. We can use a row swap
```

Each function only works under specific conditions. Hence, at every iteration we will check the conditions of our matrix to determine which function to use.

In [12]:
```
"""
Creates a matrix such that all elements within the same column as a sel
    index is zero

Args:
    matrix: numpy matrix of dimensions m rows by n columns
    row_index: The row index of the selected element
    col_index: The column index of the selected element.

Returns:
    A new array such that elements from matrix.getitem(row_index + 1, c
        to matrix.getitem(m, col_index) = 0
"""
def zeros_underneath(matrix, row_index, col_index):
    matrix2 = copy.deepcopy(matrix)
    row_j = row_index #use the row above

    for row_i in range(row_index + 1, len(matrix), 1): # for each row u
        if matrix2.item(row_j, col_index) == 0:
            matrix2 = row_swap(matrix2, row_i, row_j)
        else:
            k = matrix.item(row_i, col_index) / matrix2.item(row_j, col
            matrix2 = row_replacement(matrix2, row_i, 1, row_j, k)
    return matrix2
```

Example of Setting Zeros Underneath

In [13]:
```
matrix = np.matrix([[5],
                    [6],
                    [7]])
zeros_underneath(matrix, 0, 0)
```

Out[13]:
```
matrix([[5],
        [0],
        [0]])
```

Example of Upper Triangular Matrix

This example was found on [YouTube (https://www.youtube.com/watch?v=f-zQJtkgcOE)](https://www.youtube.com/watch?v=f-zQJtkgcOE)

In [14]:
```
matrix = np.matrix([[2, 4, -2],
                    [4, -2, 6],
                    [6, -4, 2]])
gauss_elimination(matrix)
```

Out[14]:
```
matrix([[  2.,   4.,  -2.],
        [  0., -10.,  10.],
        [  0.,   0.,  -8.]])
```

3 Back Substitution

Recall that in Gauss elimination we produced a triangular matrix, U. In BackSubstitution we manipulate this triangular matrix to solve the the general equation Ux=b

In [15]:
```
"""
Solves a system of equations of the form Ux=b

Args:
    U: numpy matrix upper triangular matrix of dimensions m rows by n c
    b: numpy matrix of dimensions n rows by 1 column

Returns:
    The x matrix such that Ux = b. If there are free variables it will
"""
#THIS ONE WORKS
def back_substitution(U, b):
    U = U.astype(float)
    b = b.astype(float)
    x = [0.0] * len(U)
    is_free_variable_found = False

    for i in range(len(x) - 1, -1, -1): # substitute from the bottom of
        row = U[i].flatten()
        constant = expanded_dot(row, x)
        if (is_free_variable_found):
            x[i] = "( " + str(b[i].item()) + " - (" + str(constant) + "
        else:
            if (row.item(i) == 0): #We might have found a free variable
                if (reality_check(constant, b.item(i))): # reality chec
                    is_free_variable_found = True
                    x[i] = "?"
                else:
                    return "There is no solution"
            else:
                x[i] = (b[i].item() - constant) / row.item(i) # Solve f

    if is_free_variable_found:
        return (resolve_free_variables(x, 1.0), resolve_free_variables(
    return x
```

In [16]:
```
def reality_check(x, y):
    return x == y
```

In [17]:
```
def resolve_free_variables(row, replace_with):
    clone = copy.deepcopy(row)

    for i in range(len(clone)):
        if isinstance(clone[i], str) :
            clone[i] = eval(clone[i].replace("?", "(" + str(replace_wit
    return clone
```

4 LU Factorization

Recall that in arithmetic we learned to factor scalars such as 10 = 2 * 5. Fur-thermore, in algebra we learned to factor polynomials such as x^2 + 2x - 3 = (x- 1) (x + 3). The same holds for linear algebra. In this section we will look athow to factor a square matrix A such that A = LU

An lower triangular matrix can be found by keeping track of the k values used when finding an upper triangular matrix.

In [18]:

```
"""
Convert a matrix into it's upper triangular form
"""
def LU_factorization(matrix):
    matrix = matrix.astype(float)
    upper = copy.deepcopy(matrix)
    lower = np.identity(len(matrix))

    for i in range(len(matrix.T)): # for each column
        result = __upper_lower_column_helper(upper, i, i, lower) #Make
        upper = result[0]
        lower = result[1]

    return lower, upper
```

This function **upper_lower_column_helper** simply performs the upper triangular operatin while keeping track of these k values

In [19]:

```
"""
Finds the upper triangular matrix while
also keeping track of the k values to use in the lower triangular matri
"""
def __upper_lower_column_helper(matrix, row_index, col_index, lower):
    matrix2 = copy.deepcopy(matrix)
    row_j = row_index #use the row above

    k=0
    for row_i in range(row_index + 1, len(matrix), 1): # for each row u
        if matrix2.item(row_j, col_index) == 0:
            matrix2 = row_swap(matrix2, row_i, row_j)
        else:
            k = matrix.item(row_i, col_index) / matrix2.item(row_j, col
            matrix2 = row_replacement(matrix2, row_i, 1, row_j, k)
        lower[row_i][col_index] = k

    return matrix2, lower
```

5 Solving a System of Equations - Gauss Elimination with Back Substitution

We already implemented Gauss elimination and back substitution in the earlier sections of this document. Now it will only take a few lines of code to solve Ax=b

In [20]:

```
"""
Solves the equation Ax = b

Args:
    A: numpy matrix of dimensions m rows by n columns
    b: numpy matrix of dimensions n rows by 1 column

Returns:
    The x matrix such that Ax = b
"""
def gauss_elimination_and_back_substitution(A, b):
    # First, convert to floats. This helps avoid integer rounding
    A = A.astype(float)
    b = b.astype(float)

    augmented = np.hstack((A, b)) # Convert from A, b to [A|b]
    U = gauss_elimination(augmented)
    U, b = np.hsplit(U, [np.size(U, 1) - 1]) # Convert from [U|b] to U,
    x = back_substitution(U, b)

    return x
```

6. Examples of Gauss Elimination and Back Substitution

In [21]:

```
def print_solution(A, b):
    print("solution = ")
    print(gauss_elimination_and_back_substitution(A, b))
    print("")
    L, U = LU_factorization(A)
    print("L = ")
    print(L)

    print("U = ")
    print(U)
```

In [22]:
```
A = np.matrix([[2, 1, -1],
               [3, 2, 1],
               [2, -1, 2]])
b = np.matrix([[1],
               [10],
               [6]])

# Solution is [1, 2, 3]
print_solution(A, b)
```

```
solution =
[1.0, 2.0, 3.0]

L =
[[ 1.   0.   0. ]
 [ 1.5  1.   0. ]
 [ 1.  -4.   1. ]]
U =
[[ 2.   1.  -1. ]
 [ 0.   0.5  2.5]
 [ 0.   0.  13. ]]
```

a.

In [23]:
```
A = np.matrix([[1, -1, 2, -1],
               [2, -2, 3, -3],
               [1, 1, 1, 0],
               [1, -1, 4, 3]])

b = np.matrix([[-8],
               [-20],
               [-2],
               [4]])

# Solution is [-7, 3, 2, 2]
print_solution(A, b)
```

```
solution =
[-7.0, 3.0, 2.0, 2.0]

L =
[[ 1.  0.  0.  0.]
 [ 2.  1.  0.  0.]
 [ 1.  0.  1.  0.]
 [ 1.  0. -2.  1.]]
U =
[[ 1. -1.  2. -1.]
 [ 0.  2. -1.  1.]
 [ 0.  0. -1. -1.]
 [ 0.  0.  0.  2.]]
```

b.

In [24]:
```
A = np.matrix([[1, 1, 1],
               [2, 2, 1],
               [1, 1, 2]])

b = np.matrix([[4],
               [6],
               [6]])

print_solution(A, b)
```

```
solution =
([1.0, 1.0, 2.0], [3.0, -1.0, 2.0])

L =
[[1. 0. 0.]
 [2. 1. 0.]
 [1. 0. 1.]]
U =
[[ 1.  1.  1.]
 [ 0.  0.  1.]
 [ 0.  0. -1.]]
```

c.

In [25]:
```
A = np.matrix([[1, 1, 1],
               [2, 2, 1],
               [1, 1, 2]])

b = np.matrix([[4],
               [4],
               [6]])

print_solution(A, b)
```

```
solution =
There is no solution

L =
[[1. 0. 0.]
 [2. 1. 0.]
 [1. 0. 1.]]
U =
[[ 1.  1.  1.]
 [ 0.  0.  1.]
 [ 0.  0. -1.]]
```

In [ ]:
```

```