ENGG 407 Assignment #2

Kyle Derby MacInnis
November 16, 2012

Question 1:

3.3 – Find the angle for which the centroid is equal to 0.75*r:

a) *Calculate the centroid using the Bisection Method*

Pseudo-Code Algorithm:

```
// Set Initial Interval Boundaries
a = 0.1 ;
b = 1.4 ;

// Set Target Centroid
Ct = 0.75*r ;

// Iterate 4 times
Loop(4)
{
      // Bisect Interval
      h = (b + a)/2 ;

      // Calculate Centroid at Boundaries
      C(a) = 2*r*sin^3(a)/(3*(a - sin(a)*cos(a))) ;
      C(b) = 2*r*sin^3(b)/(3*(b - sin(b)*cos(b))) ;

      // Calculate Centroid at Bisection
      C(h) = 2*r*sin^3(h)/(3*(h - sin(h)*cos(h))) ;

      // Output Bisected Angle and Centroid
      Output( "Angle: %1 Centroid: %2", (h, C(h)) ) ;

      // If root is found, stop iterating
      If ( C(h) == Ct )
            break;

      // Otherwise replace corresponding boundary with bisection
      Else If ( [C(h) > Ct AND C(a) > Ct] OR [C(h) < Ct AND C(a) < Ct] )
            a = h ;

      Else
            b = h ;
}
```

Results:

1: *Angle: 0.7500* *Centroid: 0.8404r*
2: *Angle: 1.0750* *Centroid: 0.6910r*
3: *Angle: 0.9125* *Centroid: 0.7700r*
4: *Angle: 0.9937* *Centroid: 0.7314r*

b) *Calculate the centroid using the Secant Method*

<u>Pseudo-Code Algorithm:</u>

```
// Set Initial points
x1 = 0.1 ;
x2 = 1.4 ;

// Set Target Centroid
Ct = 0.75*r ;

// Iterate 4 times
Loop(4)
{
      // Calculate Centroid at Boundaries
      C(x1) = 2*r*sin^3(x1)/(3*(x1 - sin(x1)*cos(x1))) ;
      C(x2) = 2*r*sin^3(x2)/(3*(x2 - sin(x2)*cos(x2))) ;

      // Calculate Slope of Secant Line
      m = (C(x2) - C(x1))/(x2 - x1) ;

      // Calculate Intercept of Secant Line
      b = C(x2) - m*x2 ;

      // Calculate new point
      xi = (Ct - b)/m ;

      // Calculate Centroid at new point
      C(xi) = 2*r*sin^3(xi)/(3*(xi - sin(xi)*cos(xi))) ;

      // Output Secant Angle and Centroid
      Output( "Angle: %1 Centroid: %2", xi, C(xi) ) ;

      // If root is found, stop iterating
      If ( C(xi) == Ct )
            break;

      // Otherwise replace corresponding point with new one
      Else If ( [C(xi) > Ct AND C(x1) > Ct] OR [C(xi) < Ct AND C(x1) < Ct] )
            x1 = xi ;

      Else
            x2 = xi ;
}
```

<u>Results:</u>

1:	*Angle: 0.7698*	*Centroid: 0.8323r*
2:	*Angle: 0.9347*	*Centroid: 0.7597r*
3:	*Angle: 0.9532*	*Centroid: 0.7509r*
4:	*Angle: 0.9549*	*Centroid: 0.7501r*

3.8 – Calculate the first two positive roots of the equation using Newton's Method

<u>Pseudo-Code Algorithm:</u>

```
// Set Starting Values for both roots
x1 = 0.1 ;
x2 = 1.4 ;

// Iterate 4 times for first root
Loop(4)
{
      // Calculate Value at point
      y(x1) = x1^2 + 4*sin(2*x1) - 2 ;

      // Calculate Slope of Tangent Line at point
      m = 2*x1 + 8*cos(2*x1) ;

      // Calculate Intercept of Tangent Line
      b = (y(x1) - m*x1) ;

      // Calculate new point
      xi = (-b)/m ;

      // Calculate Value at new point
      y(xi) = xi^2 + 4*sin(2*xi) - 2 ;

      // Output Secant Angle and Centroid
      Output( "Root: %1 Value: %2", xi, y(xi) ) ;

      // If root is found, stop iterating
      If ( y(xi) == 0 )
            break ;

      // Otherwise replace point with new one
      Else
            x1 = xi ;
}

// Iterate 4 times for second root
Loop(4)
{
      // Same Code, just substitute in x2
      ...
}
```

<u>Results:</u>

First Positive Root:

1:	*Root: 0.2487*	*Value: -0.0299*
2:	*Root: 0.2526*	*Value: 0.0000*
3:	*Root: 0.2526*	*Value: 0.0000*
4:	*Root: 0.2526*	*Value: 0.0000*

Second Positive Root:

1:	*Root: 1.6744*	*Value: -0.0192*
2:	*Root: 1.6701*	*Value: 0.0000*
3:	*Root: 1.6701*	*Value: 0.0000*
4:	*Root: 1.6701*	*Value: 0.0000*

```python
# ENGG 407 - Assignment 2 - Q1 - 3.3a
#
# Name: Kyle Derby MacInnis
# Date: November 16, 2012
#
# Note: Python Programming Language

# Import Math Library for Sin() and Cos()
import math

# Set Initial Boundaries
a = 0.1
b = 1.4

# Set Target Centroid
Ct = 0.75

# Iterate 4 Times
for i in range(4):

    # Bisect Interval
    h = (b + a)/2

    # Find Boundary Values
    Ca = (2*(math.sin(a))**3)/(3*(a - math.sin(a)*math.cos(a)))
    Cb = (2*(math.sin(b))**3)/(3*(b - math.sin(b)*math.cos(b)))

    # Find bisection Value
    Ch = (2*(math.sin(h))**3)/(3*(h - math.sin(h)*math.cos(h)))

    # Output Value
    print "%i: Angle: %.4f Centroid: %.4f%s" % (i+1, h, Ch, "r")

    # If root, break
    if (Ch == Ct):
        break
    # Else Replace Corresponding Boundary Position
    elif (Ch > Ct and Ca > Ct) or (Ch < Ct and Ca < Ct):
        a = h
    else:
        b = h
```

```python
# ENGG 407 - Assignment 2 - Q1 - 3.3b
#
# Name: Kyle Derby MacInnis
# Date: November 16, 2012
#
# Note: Python Programming Language

# Import Math Library for Sin() and Cos()
import math

# Set Initial Boundaries
x1 = 0.1
x2 = 1.4

# Set Target Centroid
Ct = 0.75

# Iterate 4 Times
for i in range(4):

    # Find Boundary Values
    Cx1 = (2*(math.sin(x1))**3)/(3*(x1 - math.sin(x1)*math.cos(x1)))
    Cx2 = (2*(math.sin(x2))**3)/(3*(x2 - math.sin(x2)*math.cos(x2)))

    # Find Slope of Secant Line
    m = (Cx2 - Cx1)/(x2 - x1)

    # Find Intercept of Secant Line
    b = (Cx2 - m*x2)

    # Calculate new point
    xi = (Ct - b)/m

    # Find new centroid Value
    Cxi = (2*(math.sin(xi))**3)/(3*(xi - math.sin(xi)*math.cos(xi)))

    # Output Value
    print "%i: Angle: %.4f Centroid: %.4f%s" % (i+1, xi, Cxi, "r")

    # If root, break
    if (Cxi == Ct):
        break
    # Else Replace Corresponding Boundary Position
    elif (Cxi > Ct and Cx1 > Ct) or (Cxi < Ct and Cx1 < Ct):
        x1 = xi
    else:
        x2 = xi
```

```python
# ENGG 407 - Assignment 2 - Q1 - 3.8
#
# Name: Kyle Derby MacInnis
# Date: November 16, 2012
#
# Note: Python Programming Language

# Import Math Library for Sin() and Cos()
import math

# Set Initial Boundaries
x1 = 0.1
x2 = 1.4

print "First Positive Root:"

# Iterate 4 Times
for i in range(4):

    # Find Initial Value
    yx1 = x1**2 + 4*math.sin(2*x1) - 2

    # Find Slope of Tangent Line
    m = 2*x1 + 8*math.cos(2*x1)

    # Find Intercept of Line
    b = (yx1 - m*x1)

    # Calculate new point
    xi = (-b)/m

    # Find new value
    yxi = xi**2 + 4*math.sin(2*xi) - 2

    # Output Value
    print "\t%i: Root: %.4f Value: %.4f" % (i+1, xi, yxi)

    # If root, break
    if (yxi == 0):
        break
    # Else Replace Corresponding point
    else:
        x1 = xi

print "\nSecond Positive Root:"

# Iterate 4 Times
for i in range(4):

    # Find Boundary Values
    yx2 = x2**2 + 4*math.sin(2*x2) - 2

    # Find Slope of Line
    m = 2*x2 + 8*math.cos(2*x2)

    # Find Intercept of Line
    b = (yx2 - m*x2)

    # Calculate new point
    xi = (-b)/m

    # Find value at new point
    yxi = xi**2 + 4*math.sin(2*xi) - 2

    # Output Value
    print "\t%i: Root: %.4f Value: %.4f" % (i+1, xi, yxi)
```

```
    # If root, break
    if (yxi == 0):
        break
    # Else Replace point
    else:
        x2 = xi
```

Question 2:

3.10 – Use the fixed-point iteration method to find the root of the equation

a) Find two forms of g(x) to be used for fixed-point iteration, and determine which is correct

Equation:

$$f(x) = x^2 - 5x^{\frac{1}{3}} + 1 = 0$$

First Form, g(x):

$$g(x) = \sqrt{5x^(1/3)1}$$

Second Form, g(x):

$$g(x) = (\frac{x^2+1}{5})^3$$

In order to satisfy the Lipschitz condition(and converge within the neighbourhood of the fixed point), the absolute value of the derivative of the form must be less than 1 around the fixed point, so lets test the forms around the neighbourhood of $x = 2$ and $x = 2.5$:

Derivative of First Form:

$$g'(x) = \frac{5}{6}\frac{x^{-\frac{2}{3}}}{\sqrt{5x^{\frac{1}{3}} - 1}}$$

$$|g'(2)| = |0.228040| < |1|$$

$$|g'(2.5)| = |0.188077| < |1|$$

Derivative of Second Form:

$$g'(x) = \frac{6}{5}x(\frac{x^2+1}{5})^2$$

$$|g'(2)| = |6.3075| > |1|$$

$$|g'(2.5)| = |2.4000| > |1|$$

Based on the conditional analysis, it appears that the first form of g(x) is the one we must use as it will converge on the point due to satisfying the condition above.

b) *Perform 5 iterations of the two forms of g(x) to confirm the correct choice*

Pseudo-Code Algorithm:

```
// Set Starting Value
x1 = 2 ;

// Iterate 5 times for first method
Loop(5)
{
       // Find g(x) at point using first form
       g1(x1) = sqrt(  5*x1^(1/3) - 1 ) ;

       // Set new point
       x1 = g1(x1) ;

       // Find value at new point
       f(x1) = x1^2 - 5*x1^(1/3) + 1 ;

       // Output Resulting Root and Value
       Output( "Root: %1 Value: %2", x1, f(x1) ) ;
}

// Reset Initial Point
x1 = 2 ;

// Iterate 5 times for second method
Loop(5)
{
       //Substitute in Second Form
       g2(x1) = ( (x1^2 + 1)/5 )^3 ;

       // Set new point
       x1 = g1(x1) ;

       // Output Resulting Root and Value
       Output( "Root: %1 Value: %2", x1, f(x1) ) ;

       // Find value at point
       f(x1) = x1^2 - 5*x1^(1/3) + 1 ;
}
```

Results:

First Method:

1:	*X: 2.3021*	*F(x): -0.3024*
2:	*X: 2.3668*	*F(x): -0.0613*
3:	*X: 2.3797*	*F(x): -0.0121*
4:	*X: 2.3823*	*F(x): -0.0024*
5:	*X: 2.3828*	*F(x): -0.0005*

Second Method:

1:	*X: 1.0000*	*F(x): -3.0000*
2:	*X: 0.0640*	*F(x): -0.9961*
3:	*X: 0.0081*	*F(x): -0.0042*
4:	*X: 0.0080*	*F(x): -0.0002*
5:	*X: 0.0080*	*F(x): -0.0002*

As was expected, the first form of g(x) was indeed the correct one. It converged to the fixed point within the intended region whilst the second form led to a diversion from the intended region.

3.13 – Solve a system of nonlinear equations using numerical methods

b) *Use the fixed-point interation method to solve the system of equations*

<u>Pseudo-Code Algorithm:</u>

```
// Set initial Values for variables
x1 = 1 ;
y1 = 1 ;

// Iterate 5 times
Loop(5)
{
      // Calculate New Point
      y = (4*x1^2 + 28)^(1/3)  ;
      x = ( (145 - 4*y1^2)/3 )^(1/3)  ;

      // Set New Point
      x1 = x ;
      y1 = y ;

      // Calculate Value at New Point
      f1 = 4*x1^2 - y1^3 + 28 ;
      f2 = 3*x1^3 + 4*y1^2 - 145 ;

      // Output Iterated Roots
      Output("X: %1 Y: %2 F1: %3 F2: %4", x1, x2, f1, f2) ;
}
```

<u>Results:</u>

1	*X: 3.6084*	*Y: 3.1744*	*F1: 48.0922*	*F2: 36.2539*
2	*X: 3.2675*	*Y: 4.3097*	*F1: -9.3404*	*F2: 33.9486*
3	*X: 2.8668*	*Y: 4.1345*	*F1: -9.8010*	*F2: -5.9399*
4	*X: 2.9446*	*Y: 3.9333*	*F1: 1.8344*	*F2: -6.5191*
5	*X: 3.0256*	*Y: 3.9718*	*F1: 1.9588*	*F2: 1.1925*
6	*X: 3.0107*	*Y: 4.0123*	*F1: -0.3322*	*F2: 1.2634*
7	*X: 2.9948*	*Y: 4.0048*	*F1: -0.3551*	*F2: -0.2653*
8	*X: 2.9978*	*Y: 3.9969*	*F1: 0.0974*	*F2: -0.2810*
9	*X: 3.0009*	*Y: 3.9983*	*F1: 0.1019*	*F2: 0.0205*
10	*X: 3.0003*	*Y: 3.9999*	*F1: 0.0126*	*F2: 0.0235*

As can be seen, the iterations converge to the solution of the system and and the solution is around (3, 4).

```python
# ENGG 407 - Assignment 2 - Q2 - 3.10b
#
# Name: Kyle Derby MacInnis
# Date: November 16, 2012
#
# Note: Python Programming Language

# Import library required for sqrt()
import math

# Set Initial Starting Point
x1 = 2.0

print "First Method:"

# Iterate 5 times for first method
for i in range(5):

    # Find New Point
    g1x1 = math.sqrt( 5*(x1**(0.3333)) - 1 )

    # Set New Point
    x1 = g1x1

    # Value of Equation at Point
    Fx1 = x1**2 - 5*(x1**(0.3333)) + 1

    # Output Iterated root
    print "\t%i:\tX: %.4f\tF(x): %.4f" % (i+1, x1, Fx1)

# Reset to initial point
x1 = 2.0

print "\nSecond Method:"

# Iterate 5 times for second method
for i in range(5):

    # Find new point
    g2x1 = ( (x1**2 + 1)/5 )**3

    # Set New Point
    x1 = g2x1

    # Value of Equation at Point
    Fx1 = x1**2 - 5*(x1**(0.3333)) + 1

    # Output Iterated root
    print "\t%i:\tX: %.4f\tF(x): %.4f" % (i+1, x1, Fx1)
```

```python
# ENGG 407 - Assignment 2 - Q2 - 3.13b
#
# Name: Kyle Derby MacInnis
# Date: November 16, 2012
#
# Note: Python Programming Language

# Set Initial Starting Point Coordinates
x1 = 1.0
y1 = 1.0

# Iterate 10 Times
for i in range(10):

    # Calculate New Coordinates
    x = ( (145 - 4*(y1**2))/3 )**(0.3333)
    y = (4*(x1**2) + 28)**(0.3333)

    # Set New Coordinates
    x1 = x
    y1 = y

    # Evaluate New Coordinates in System
    f1 = 4*(x1**2) - (y1**3) + 28
    f2 = 3*(x1**3) + 4*(y1**2) - 145

    # Output Results
    print "%i\tX: %.4f\tY: %.4f\tF1: %.4f\tF2: %.4f" % (i+1, x1, y1, f1, f2)
```

Question 3:

4.2 – Use Gauss Elimination to Solve the System of Equations

Pseudo-Code Algorithm:

```
// Setup Main and Temp NxN Matrix of System
A = [ [3 -2 5] : [1 -1 0] : [2 0 4] ] ;
_A = [ [0 0 0] : [0 0 0] : [0 0 0] ] ;

// Setup Main and Temp Known Vector
b = [ 14 -1 14 ] ;
_b = [ 0 0 0 ] ;

// Setup Solution Vector
x = [ x1 x2 x3 ] ;

// Size of Matrix
N = 3;

// Loop through rows with Iterator i
Loop(i:N)
{
      // If Pivot Point Coefficient is zero
      while ( A[i][i] == 0 )
      {
            // If No more rows left to switch
            If ( i+1 == N )
            {
                  // Output Error and Exit
                  Output( "Error: System has no unique solution" ) ;
                  Exit(-1) ;
            }

            // Switch Rows with one below
            Loop(m:N)
            {
                  // In Matrix
                  A_[i][m] = A[i][m] ;
                  A[i][m] = A[i+1][m] ;
                  A[i+1][m] = A_[i][m] ;

                  // In Known Vector
                  b_[i] = b[i] ;
                  b[i] = b[i+1] ;
                  b[i+1] = b_[i] ;
            }
      }

      // Set Divisor for Row
      divr = A[i][i] ;

      // Loop through remaining rows with Iterator k
      Loop(k:N)
      {
            // If Previous Rows, Skip
            If (k <= i)
            {
                  pass ;
            }
```

```
                // Otherwise Perform Elimination
                Else
                {
                      // Find Multiplier for remaining rows
                      Mulr = (-A[k][i]) ;

                      // Loop through columns of the matrix with Iterator j
                      Loop(j:N)
                      {
                            // Normalize Elements in row in respect to A[i][i]
                            A[i][j] = A[i][j] / divr ;

                            // Eliminate Elements for different rows in column
                            A[k][j] = A[k][j] + Mulr*A[i][j] ;

                      }
                      // Do the same for the Known vector
                      b[i] = b[i] / divr ;
                      b[k] = b[k] + Mulr*b[i] ;

                      // Ensure Normalization doesn't Occur again
                      Divr = 1.0 ;
                }
          }
    }
    // Set to Solution
    x3 = b[2] / A[2][2] ;
    x2 = b[1] - A[1][2]*x3 ;
    x1 = b[0] - A[0][1]*x2 - A[0][2]*x3 ;

    // Output Result
    Output("X = [ %1 %2 %3 ]", x1, x2, x3);
```

Results:

A = :	[1.0000	-0.6667	1.6667]
	[0.0000	1.0000	5.0000]
	[0.0000	0.0000	1.0000]
B = :	[4.6667	17.0000	3.0000]
X = :	[1.0000	2.0000	3.0000]

4.2 – Use Gauss-Jordan Elimination to Solve the System of Equations

Pseudo-Code Algorithm:

```
// Setup Main and Temp NxN Matrix of System
A = [ [3 -2 5] : [1 -1 0] : [2 0 4] ] ;
_A = [ [0 0 0] : [0 0 0] : [0 0 0] ] ;

// Setup Main and Temp Known Vector
b = [ 14 -1 14 ] ;
_b = [ 0 0 0 ] ;

// Setup Solution Vector
x = [ x1 x2 x3 ] ;
```

```
// Size of Matrix
N = 3;

// Loop through rows with Iterator i
Loop(i:N)
{
      // If Pivot Point is zero
      while ( A[i][i] == 0 )
      {
            // If No more rows left to try
            If ( i+1 == N )
            {
                  // Output Error and Exit
                  Output( "Error: System has no unique solution" ) ;
                  Exit(-1) ;
            }

            // Switch Rows with one below
            Loop(m:N)
            {
                  // In Matrix
                  A_[i][m] = A[i][m] ;
                  A[i][m] = A[i+1][m] ;
                  A[i+1][m] = A_[i][m] ;

                  // In Known Vector
                  b_[i] = b[i] ;
                  b[i] = b[i+1] ;
                  b[i+1] = b_[i] ;
            }
      }

      // Set Divisor for Row
      divr = A[i][i] ;

      // Loop through remaining rows with Iterator k
      Loop(k:N)
      {
            // If Previous Rows, Skip
            If (k == i)
            {
                  pass ;
            }

            // Otherwise Perform Elimination
            Else
            {
                  // Find Multiplier for remaining rows
                  Mulr = (-A[k][i]) ;

                  // Loop through columns of the matrix with Iterator j
                  Loop(j:N)
                  {
                        // Normalize Elements in row in respect to A[i][i]
                        A[i][j] = A[i][j] / divr ;

                        // Eliminate Elements for different rows in column
                        A[k][j] = A[k][j] + Mulr*A[i][j] ;

                  }
```

```
                // Do the same for the Known vector
                b[i] = b[i] / divr ;
                b[k] = b[k] + Mulr*b[i] ;

                // Ensure Normalization doesn't Occur again
                Divr = 1.0 ;
            }
        }
}

// Set to Solution which is now Known Vector
x1 = b[1] ;
x2 = b[2] ;
x3 = b[3] ;

// Output Result
Output("X = [ %1 %2 %3 ]", x1, x2, x3) ;
```

<u>Results:</u>

A = :	[1.0000	0.0000	0.0000]
	[0.0000	1.0000	0.0000]
	[0.0000	0.0000	1.0000]
B = :	[1.0000	2.0000	3.0000]
X = :	[1.0000	2.0000	3.0000]

As can be see from both methods (Gauss and Gauss-Jordan) the results are the same.

```python
# ENGG 407 - Assignment 2 - Q3 - 4.2
#
# Name: Kyle Derby MacInnis
# Date: November 16, 2012
#
# Note: Python Programming Language

# Setup Initial Matrix and Temp Matrix
A = [ [3.0,-2.0,5.0], [1.0,-1.0,0.0], [2.0,0.0,4.0] ]
A_ = [ [0.0,0.0,0.0], [0.0,0.0,0.0], [0.0,0.0,0.0] ]

# Setup Solution Vector and Temp Vector
b = [ 14.0, -1.0, 14.0 ]
b_ = [ 0.0, 0.0, 0.0 ]

# Number of Rows and Columns of Matrix
N = 3

# Iterate through rows
for i in range(N):

    # If Pivot Point is zero
    while ( A[i][i] == 0 ):

        # If No more rows left to try
        if (i+1 == N):
            print "Error: System has no unique solution"
            exit(-1)

        # Switch Rows with one below
        for m in range(N):
            # In Matrix
            A_[i][m] = A[i][m]
            A[i][m] = A[i+1][m]
            A[i+1][m] = A_[i][m]

            # In Known Vector
            b_[i] = b[i]
            b[i] = b[i+1]
            b[i+1] = b_[i]

    # Find Row Divisor
    DivR = A[i][i]

    # Iterate Through Remaining Rows
    for k in range(N):

        # Move to Remaining Rows
        if (k <= i):
            pass

        # Otherwise Perform Elimination
        else:

            # Find Multiplier for Other Rows
            MulR = (-A[k][i])

            # Iterate through each Element in row
            for j in range(N):

                # Normalize Elements in Row
                A[i][j] = A[i][j] / DivR

                # Eliminate Column in other Rows
                A[k][j] = A[k][j] + MulR * A[i][j]

            # Apply Same Changes to known Vector
```

```
            b[i] = b[i] / DivR
            b[k] = b[k] + MulR*b[i]

            # Ensure Normalization Occurs only once per row
            DivR = 1.0;

# Solve for Solution Vector Elements
x3 = b[2] / A[2][2]
x2 = b[1] - A[1][2]*x3
x1 = b[0] - A[0][1]*x2 - A[0][2]*x3

# Print Solution Vector
print "\nX = [ %.4f %.4f %.4f ]" % (x1, x2, x3)
```

```python
# ENGG 407 - Assignment 2 - Q3 - 4.7
#
# Name: Kyle Derby MacInnis
# Date: November 16, 2012
#
# Note: Python Programming Language

# Setup Initial Matrix and Temp Matrix
A = [ [3.0,-2.0,5.0], [1.0,-1.0,0.0], [2.0,0.0,4.0] ]
A_ = [ [0.0,0.0,0.0], [0.0,0.0,0.0], [0.0,0.0,0.0] ]

# Setup Solution Vector and Temp Vector
b = [ 14.0, -1.0, 14.0 ]
b_ = [ 0.0, 0.0, 0.0 ]

# Number of Rows and Columns of Matrix
N = 3

# Iterate through rows
for i in range(N):

    # If Pivot Point is zero
    while ( A[i][i] == 0 ):

        # If No more rows left to try
        if (i+1 == N):
            print "Error: System has no unique solution"
            exit(-1)

        # Switch Rows with one below
        for m in range(N):
            # In Matrix
            A_[i][m] = A[i][m]
            A[i][m] = A[i+1][m]
            A[i+1][m] = A_[i][m]

            # In Known Vector
            b_[i] = b[i]
            b[i] = b[i+1]
            b[i+1] = b_[i]

    # Find Row Divisor
    DivR = A[i][i]

    # Iterate Through Remaining Rows
    for k in range(N):

        # If Pivot Row Skip
        if (k == i):
            pass

        # Otherwise Perform Elimination
        else:

            # Find Multiplier for Other Rows
            MulR = (-A[k][i])

            # Iterate through each Element in row
            for j in range(N):

                # Normalize Elements in Row
                A[i][j] = A[i][j] / DivR

                # Eliminate Column in other Rows
                A[k][j] = A[k][j] + MulR * A[i][j]

            # Apply Same Changes to known Vector
```

```
            b[i] = b[i] / DivR
            b[k] = b[k] + MulR*b[i]

            # Ensure Normalization Occurs only once per row
            DivR = 1.0

# Known vector becomes Solution Vector X
x1 = b[0]
x2 = b[1]
x3 = b[2]

# Print Solution Vector
print "\nX = [ %.4f %.4f %.4f ]" % (x1, x2, x3)
```

Question 4:

4.12 – Solve the system using LU decomposition with Crout's Method

Pseudo-Code Algorithm:

```
// Set up coefficient matrix
A = [ [2 -4 1] : [6 -2 1] : [-2 6 -2] ] ;

// Set up Lower and Upper Blank Matrices
L = [ [0 0 0] : [0 0 0] : [0 0 0] ] ;
U = [ [0 0 0] : [0 0 0] : [0 0 0] ] ;

// Set up Known and Solution Vectors
b = [ 4 10 -6 ] ;
x = [ x1 x2 x3 ];
y = [ y1 y2 y3 ];

// Set up dimensions of NxN system
N = 3;

// Loop through rows with iterator i
Loop(i : N)
{
      // Set Lower First Column Elements
      L[i][0] = A[i][0] ;

      // Set Upper Diagonal Elements
      U[i][i] = 1 ;

      // Loop through columns
      Loop(j : N)
      {
            // If First Row
            If ( i == 0 )
            {
                  U[i][j] = A[i][j] / L[i][0] ;
            }
            // Calculate Lower Elements
            Else If (j <= i)
            {
                  L[i][j] = A[i][j] ;

                  Loop(k : j-1 )
                  {
                        L[i][j] -= L[i][k]*U[k][j] ;
                  }
            }
            // Calculate Upper Elements
            Else
            {
                  U[i][j] = A[i][j] / L[i][i] ;

                  Loop( k : i-1 )
                  {
                        U[i][j] -= (L[i][k]*U[k][j]) / L[i][i] ;
                  }
            }
      }
}
```

```
// Calculate y solution vector
y[0] = ( b[0] / L[0][0] ) ;
y[1] = ( b[1] - L[1][0]*y[0] ) / L[1][1] ;
y[2] = ( b[2] - L[1][0]*y[0] - L[2][1]*y[1] ) / L[2][2] ;

// Calculate x solution vector
x3 = y[2] ;
x2 = y[1] - U[1][2]*y[2] ;
x1 = y[0] - U[0][2]*y[2] - U[0][1]*y[1] ;

// Output Solution
Output( “X = [ %1 %2 %3 ]”, x1, x2, x3 ) ;
```

<u>Results:</u>

$$U = \begin{bmatrix} 1 & -2 & 0.5 \\ 0 & 1 & -0.5 \\ 0 & 0 & 1 \end{bmatrix}$$

$$L = \begin{bmatrix} 2 & 0 & 0 \\ 6 & -2 & 0 \\ -2 & 6 & -1 \end{bmatrix}$$

$$Y = \begin{bmatrix} 2 & 11 & 84 \end{bmatrix}$$

$$X = \begin{bmatrix} -18 & 53 & 84 \end{bmatrix}$$

```python
# ENGG 407 - Assignment 2 - Q4 - 4.12
#
# Name: Kyle Derby MacInnis
# Date: November 16, 2012
#
# Note: Python Programming Language

# Setup Initial Matrix and Temp Matrix
A = [ [2.0, -4.0, 1.0], [6.0, -2.0, 1.0], [-2.0, 6.0, -2.0] ]

# Setup Blank Lower and Upper Matrices
L = [ [0, 0, 0], [0, 0, 0], [0, 0, 0] ]
U = [ [0, 0, 0], [0, 0, 0], [0, 0, 0] ]

# Setup Blank Vectors
b = [ 4.0, -10.0, -6.0 ]
y = [ 0, 0, 0 ]

# Dimensions of System
N = 3

# Iterate through rows
for i in range(N):

    L[i][0] = A[i][0]

    U[i][i] = 1.0

    # Iterate through Columns
    for j in range(N):

        # First Row
        if ( i == 0):

            U[i][j] = A[i][j] / L[i][0] ;

        # Calculate Lower Matrix
        elif (j <= i):

            L[i][j] = A[i][j]

            for k in range(j-1):

                L[i][j] -= L[i][k]*U[k][j]

        # Calculate Upper Matrix
        else:

            U[i][j] = A[i][j] / L[i][i]

            for k in range(i-1):

                U[i][j] -= (L[i][k]*U[k][j]) / L[i][i]


# Calculate y solution vector
y[0] = ( b[0] / L[0][0] )
y[1] = ( b[1] - L[1][0]*y[0] ) / L[1][1]
y[2] = ( b[2] - L[1][0]*y[0] - L[2][1]*y[1] ) / L[2][2]

# Calculate x solution vector
x3 = y[2]
x2 = y[1] - U[1][2]*y[2]
x1 = y[0] - U[0][2]*y[2] - U[0][1]*y[1]

# Output Solution
print "X = [ %.4f %.4f %.4f ]" % (x1, x2, x3)
```

Question 5:

4.14 – Find the inverse of the system using Gauss-Jordan method

Pseudo-Code Algorithm:

```
// Setup Main and Temp NxN Matrix of System
A = [ [3 -2 5] : [1 -1 0] : [2 0 4] ] ;
_A = [ [0 0 0] : [0 0 0] : [0 0 0] ] ;

// Setup Main and Temp Known Vector
b = [ [1 0 0] : [0 1 0] : [0 0 1] ] ;
_b = [ [0 0 0] : [0 0 0] : [0 0 0] ] ;

// Size of Matrix
N = 3;

// Loop through rows with Iterator i
Loop(i:N)
{
      // If Pivot Point is zero
      while ( A[i][i] == 0 )
      {
            // If No more rows left to try
            If ( i+1 == N )
            {
                  // Output Error and Exit
                  Output( "Error: System has no unique solution" ) ;
                  Exit(-1) ;
            }

            // Switch Rows with one below
            Loop(m:N)
            {
                  // In Matrix
                  A_[i][m] = A[i][m] ;
                  A[i][m] = A[i+1][m] ;
                  A[i+1][m] = A_[i][m] ;

                  // In Known Vector
                  b_[i][m] = b[i][m] ;
                  b[i][m] = b[i+1][m] ;
                  b[i+1][m] = b_[i][m] ;
            }
      }

      // Set Divisor for Row
      divr = A[i][i] ;

      // Loop through remaining rows with Iterator k
      Loop(k:N)
      {
            // If Previous Rows, Skip
            If (k == i)
            {
                  pass ;
            }

            // Otherwise Perform Elimination
            Else
            {
```

```
                  // Find Multiplier for remaining rows
                  Mulr = (-A[k][i]) ;

                  // Loop through columns of the matrix with Iterator j
                  Loop(j:N)
                  {
                        // Normalize Elements in row in respect to A[i][i]
                        A[i][j] = A[i][j] / divr ;

                        // Eliminate Elements for different rows in column
                        A[k][j] = A[k][j] + Mulr*A[i][j] ;

                        // Do the same for the Known vector
                        b[i][j] = b[i][j] / divr ;
                        b[k][j] = b[k][j] + Mulr*b[i][j] ;
                  }

                  // Ensure Normalization doesn't Occur again
                  Divr = 1.0 ;
            }
      }
}

// Set to Solution which is b
iA = b ;

// Output Result
Output("Inverse A = %1 ", iA) ;
```

Results:

$$\text{Inverse A} = \begin{bmatrix} 0.5 & 1.0 & 0.0 \\ -0.3 & 0.1 & 0.0 \\ 1.6 & -0.2 & 1.0 \end{bmatrix}$$

4.15 – Use the Gauss-Seidel Iterative Method to carry-out the first three iterations

Pseudo-Code Algorithm:

```
// Set initial values to zero
x1 = 0 ;
x2 = 0 ;
x3 = 0 ;

// Iterate 3 times
Loop(3)
{
      // Recalculate new values
      x1 = ( 51 - 2*x2 - 3*x3 ) / 8 ;
      x2 = ( 23 - 2*x1 -x3 ) / 5 ;
      x3 = ( 20 + 3*x1 - x2 ) / 6 ;

      // Output Values
      Output( "X = [ %1 %2 %3 ]", x1, x2, x3 ) ;
}
```

Results:

1: X = [6.0000 2.0000 6.0000]
2: X = [3.0000 2.0000 4.0000]
3: X = [4.0000 2.0000 5.0000]

```python
# ENGG 407 - Assignment 2 - Q5 - 4.14
#
# Name: Kyle Derby MacInnis
# Date: November 16, 2012
#
# Note: Python Programming Language

# Setup Initial Matrix and Temp Matrix
A = [ [2.0, -4.0, 1.0], [6.0, -2.0, 1.0], [-2.0, 6.0, -2.0] ]
A_ = [ [0.0,0.0,0.0], [0.0,0.0,0.0], [0.0,0.0,0.0] ]

# Setup Solution Vector and Temp Vector
b = [ [1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0] ]
b_ = [ [0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0] ]

# Setup Blank Inverse Matrix
X = [ [0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0] ]

# Number of Rows and Columns of Matrix
N = 3

# Iterate through rows
for i in range(N):

    # If Pivot Point is zero
    while ( A[i][i] == 0 ):

        # If No more rows left to try
        if (i+1 == N):
            print "Error: System has no unique solution"
            exit(-1)

        # Switch Rows with one below
        for m in range(N):
            # In Matrix
            A_[i][m] = A[i][m]
            A[i][m] = A[i+1][m]
            A[i+1][m] = A_[i][m]

            # In Known Vector
            b_[i] = b[i][m]
            b[i] = b[i+1][m]
            b[i+1] = b_[i][m]

    # Find Row Divisor
    DivR = A[i][i]

    # Iterate Through Remaining Rows
    for k in range(N):

        # Move to Remaining Rows
        if (k <= i):
            pass

        # Otherwise Perform Elimination
        else:

            # Find Multiplier for Other Rows
            MulR = (-A[k][i])

            # Iterate through each Element in row
            for j in range(N):

                # Normalize Elements in Row
                A[i][j] = A[i][j] / DivR

                # Eliminate Column in other Rows
```

```
                A[k][j] = A[k][j] + MulR * A[i][j]

                # Apply Same Changes to known Vector
                b[i][j] = b[i][j] / DivR
                b[k][j] = b[k][j] + MulR * b[i][j]

            # Ensure Normalization Occurs only once per row
            DivR = 1.0;

# Solve for Solution Vector Elements
X = b

# Print Solution Vector
print X
```

```python
# ENGG 407 - Assignment 2 - Q5 - 4.15
#
# Name: Kyle Derby MacInnis
# Date: November 16, 2012
#
# Note: Python Programming Language

# Set Initial Values
x1 = 0
x2 = 0
x3 = 0

# Iterate 3 times
for i in range(3):

    # Calculate new values
    x1 = ( 51 - 2*x2 - 3*x3 ) / 8
    x2 = ( 23 - 2*x1 - x3 ) / 5
    x3 = ( 20 + 3*x1 - x2 ) / 6

    # Output Values
    print "%i:\tX = [ %.4f %.4f %.4f ]" % (i+1, x1, x2, x3)
```