# Solutions to Homework 1

**Question 1: 10 points.** Write a Python program that solves for all integer pairs $a$, $b \geq 0$,

$$\sqrt{a} + \sqrt{b} = \sqrt{n} \tag{1}$$

where $n$ = 2024.

**Hint:** The prime factorization of 2024 is $2 \cdot 2 \cdot 2 \cdot 11 \cdot 23$. You can use this fact and a little bit of number theory to see that there will only be 3 solutions to the problem, i.e., (a,b) pairs.

**Python Source Code:**

```
# ============================================================
# TestIntegerSolutions2024.py: Compute integer solutions to:
#
#         math.sqrt(a) + math.sqrt(b) = math.sqrt(2024).
#
# where a and b are integers greater than or equal to zero.
#
# Written by: Mark Austin                            January 2024
# ============================================================

import math

# main method ...

def main():
    print("--- Enter TestIntegerSolutions2024.main()    ... ");
    print("--- ======================================== ... ");

    n = 2024

    # Part 1: Use nested for loop to exhaustively test all cases.
    #       : Naively test for equality ...

    print("--- ")
    print("--- Part 1: Nested loops, naive test for equality ...")

    i = 1
    for a in range(0, n+1):
        for b in range(0, n+1):
```

```
            error = math.sqrt(a) + math.sqrt(b) - math.sqrt(n)
            if error == 0:
               print("--- soln {:2d}: a = {:6d}, b = {:6d}: solution !! ...".format(i,a,b) );
               i = i + 1

    # Part 2: Use nested for loop to exhaustively test all cases.
    #       : Account for imprecise number representation ...

    print("--- ")
    print("--- Part 2: Nested loops, account for inexact arithmetic ...")

    i = 1
    for a in range(0, n+1):
        for b in range(0, n+1):
            error = math.sqrt(a) + math.sqrt(b) - math.sqrt(n)
            if math.fabs(error) < 0.00000000000001:
               print("--- soln {:2d}: a = {:6d}, b = {:6d}: solution !! ...".format(i,a,b) );
               i = i + 1

    # Part 3: Use number theory to reduce number of cases ...

    print("--- ")
    print("--- Part 3: Use number theory to reduce number of cases to evaluate ...")

    mmax = 2
    m = list( range(mmax+1) );

    # Traverse list and verify expression values ...

    for i in m:
        j = mmax - i
        a = 506*i*i
        b = 506*j*j
        error = math.sqrt(a) + math.sqrt(b) - math.sqrt(n)
        if math.fabs(error) < 0.00000000000001:
           print("--- a = {:6d}, b = {:6d}: solution !! ...".format(a,b) );
        else:
           print("--- a = {:6d}, b = {:6d}: not a solution !! ...".format(a,b) );

    print("--- ======================================== ... ");
    print("--- Leave TestIntegerSolutions2024.main()    ... ");

# call the main method ...

if __name__ == "__main__":
    main()
```

**Program Output:**

```
--- Enter TestIntegerSolutions2024.main()    ...
--- ======================================== ...
---
--- Part 1: Nested loops, naive test for equality ...
```

```
--- soln  1: a =      0, b =   2024: solution !! ...
--- soln  2: a =    506, b =    506: solution !! ...
--- soln  3: a =   2024, b =      0: solution !! ...
---
--- Part 2: Nested loops, account for inexact arithmetic ...
--- soln  1: a =      0, b =   2024: solution !! ...
--- soln  2: a =    506, b =    506: solution !! ...
--- soln  3: a =   2024, b =      0: solution !! ...
---
--- Part 3: Use number theory to reduce number of cases to evaluate ...
--- a =      0, b =   2024: solution !! ...
--- a =    506, b =    506: solution !! ...
--- a =   2024, b =      0: solution !! ...

--- ======================================== ...
--- Leave TestIntegerSolutions2024.main()    ...
```

**Question 2: 10 points.** Leibnez's series is given by:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} + \cdots \tag{2}$$

Write a Python program to compute Leibniz's series summation for 1000 terms. First, use a while-loop construct to compute the series summation. Repeat the experiment using an array for the series coefficients, appropriate matrix element-level operations for the alternating signs. and `sum()` for the summation of series terms.

Use the Python package `time` (note, packages `datetime` and `timedelta` may also work) to monitor the time needed to compute each implementation. Conduct a simple experiment to see how the relative speed of the two methods varies as a function of the number of terms in the series?

**Python Source Code:**

```
# ============================================================================
# TestSeriesSummationLeibnez.py: This program sums terms in the Leibnez series
#
#      pi/4 = 1 - 1/3 + 1/5 - 1/7 .....
#
#  to 1000 terms using:
#
#      1. A simple looping construct.
#      2. An array of series coefficient, which are then summed.
#
#  Each summation is compared to the theoretical limit, pi/4.
#
# Written by: Mark Austin                                          January 2024
# ============================================================================

import math
import numpy as np
import time

# main method ...

def main():
    print("--- Enter TestSeriesSummationLeibnez.main()    ... ");
    print("--- ========================================== ... ");

    # Part 1: Theoretical result ...

    print("Part 1: Leibnez Series          ");
    print("--------------------------------");

    dTheoreticalSum = math.pi/4.0;
    print("--- Theoretical Summation = {:16.12f} ...".format(dTheoreticalSum) );

    noTerms = 100000
```

```
print("--- ") ;
print("--- No terms in series = {:d} ".format( noTerms ) ) ;

# Part 2: Compute series summation with simple loop .

dSum  = 0.0;
i     = 1.0;
start = time.time();
while( i <= noTerms ):
   if( i%2 == 1 ):
      dSum = dSum + 1.0/(2*i-1);
   else:
      dSum = dSum - 1.0/(2*i-1);

   i = i + 1

end = time.time();
duration = end - start;

print("");
print("Part 2: Simple Loop");
print("-----------------------------------");
print("--- No terms in series = {:d} ".format( noTerms ) ) ;
print("--- Summation      = {:16.12f} ".format( dSum ) ) ;
print("--- Absolute Error = {:16.12f} ".format( (dSum - dTheoreticalSum) ) ) ;
print("--- Time duration  = {:16.12f} ".format( (duration) ) ) ;

# Part 3: Store series terms in an array before computing summation ...

start = time.time();

dTerm = np.zeros(noTerms);
i     = 1;
while( i <= noTerms ):
   if( i%2 == 1 ):
      dTerm[i-1] =  1.0/(2*i-1);
   else:
      dTerm[i-1] = -1.0/(2*i-1);

   i = i + 1

# Walk along array element and sum terms ....

dSum = sum( dTerm );

end = time.time();
duration = end - start;

print("");
print("Part 3: Array Storage + Summation");
print("-----------------------------------");
print("--- Summation      = {:16.12f} ".format( dSum ) ) ;
print("--- Absolute Error = {:16.12f} ".format( (dSum - dTheoreticalSum) ) ) ;
print("--- Time duration  = {:16.12f} ".format( (duration) ) ) ;
```

```
    print("--- ======================================== ... ");
    print("--- Leave TestSeriesSummationLeibnez.main()    ... ");

# call the main method ...

if __name__ == "__main__":
   main()
```

**Program Output:** The textual output is:

```
--- Enter TestSeriesSummationLeibnez.main()    ...
--- ======================================== ...

Part 1: Leibnez Series
------------------------------
--- Theoretical Summation =   0.785398163397 ...
---
--- No terms in series = 1000

Part 2: Simple Loop
----------------------------------
--- No terms in series = 1000
--- Summation      =   0.785148163460
--- Absolute Error =  -0.000249999937
--- Time duration  =   0.000231027603

Part 3: Array Storage + Summation
---------------------------------
--- Summation      =   0.785148163460
--- Absolute Error =  -0.000249999937
--- Time duration  =   0.000294208527

--- ======================================== ...
--- Leave TestSeriesSummationLeibnez.main()    ...
```

Here are the results of a simple experiment to see how the timing varies for the two approaches and for series of various lengths:

| No Terms in Series | Simple Loop Duration (sec) | Array Storage + Summation (sec) |
|---|---|---|
| 1,000 | 0.000231027603 | 0.000294208527 |
| 100,000 | 0.022063255310 | 0.032431125641 |
| 1,000,000 | 0.202079057693 | 0.301341056824 |

Method 2, with array storage, takes approximately 50% longer than method 1.

**Question 3: 10 points.** The modulo operator, `%`, computes the remainder that occurs after an integer `m` has been divided by a second integer `n`. For example,

```
m = 5, n = 3, 5 = 1*3 + 2  --> 5%3 evaluates to 2
m = 6, n = 3, 6 = 2*3 + 0  --> 6%3 evaluates to 0
m = 7, n = 3, 7 = 2*3 + 1  --> 7%3 evaluates to 1
m = 8, n = 3, 8 = 2*3 + 3  --> 8%3 evaluates to 2
```

and so forth. It is important to notice that `m%n` will always return an integer between 0 and (n-1).

Now let A be a $(7 \times 7)$ matrix whose elements are given by

$$\mathrm{A}[i][j] = \left[i^2 + j^2)\right] \%7. \tag{3}$$

Things to do:

1. Write a short Python program to populate a (7x7) matrix with the results of equation 3.
2. Now let `p` and `q` be integers that cover the interval 0 through 100. Extend your Python program to find combinations of `p` and `q` where $p^2 + q^2$ will be divisible by 7.
3. Prove that if $p^2 + q^2$ is divisible by 7, then it will also be divisible by 49.

**Hint.** You should use numpy to store the matrix. You can also find a fancy implementation for printing matrices and vectors in the python code distributed in class. The last part of this problem is not as difficult as it looks. Write `p` as $7 * p_1 + r_1$ and `q` as $7 * q_1 + r_2$ and then an expression for $p^2 + q^2$. The result follows directly from the expression and the matrix element values in equation 3.

**Python Source Code:**

```
# ================================================================
# TestBrainTeaser.py: Use modulo arithmetric to compute remainders
# where expressions x^2 + y^2 are divided by 7.
#
# Written by: Mark Austin                              February 2024
# ================================================================

import math
import numpy as np

# ================================================================
# Fancy function to print one- and two-dimensional matrices ...
# ================================================================
```

```
def PrintMatrix(name, matrix):
    NoColumns = 6;

    # Compute no of blocks of rows to be printed .....

    if matrix.ndim == 1:
       noMatrixRows = matrix.shape[0]
       noMatrixCols = 1

    if matrix.ndim == 2:
       noMatrixRows = matrix.shape[0]
       noMatrixCols = matrix.shape[1]

    #  Compute number of blocks to be printed ...

    if noMatrixCols % NoColumns == 0:
       iNoBlocks = noMatrixCols/NoColumns;
    else:
       iNoBlocks = noMatrixCols/NoColumns + 1;

    #  Loop over the number of blocks ...

    for ib in range( int(iNoBlocks) ):
        iFirstColumn = ib*NoColumns + 1
        iLastColumn  = min ( (ib+1)*NoColumns, noMatrixCols )

        # Print title of matrix at top of each block ....

        print("Matrix: {:s} ".format(name) );

        # Label row and column nos */

        print("row/col     ", end="")
        colList = range(iFirstColumn, iLastColumn + 1)
        for col in [ *colList ]:
            print("       {:3d}    ".format(col),end="")
        print("")

        # Loop over rows and print matrix elements ....

        ii = 1
        for row in matrix:
            print(" {:3d}     ".format(ii),end="")
            colList = range( iFirstColumn, iLastColumn + 1)
            for col in [ *colList ]:
                if matrix.ndim == 1:
                   print(" {:12.5e} ".format( matrix[ii-1] ), end="")
                else:
                   print(" {:12.5e} ".format(matrix[ii-1][col-1]), end="")
            print("")
            ii = ii + 1
        print("")

# Main function ...
```

```
def main():
    print("--- Enter TestBrainTeaser.main()             ... ");
    print("--- ======================================== ... ");

    print("---");
    print("--- Part 1: Setup (7x7) matrix ...");
    print("---");

    matrixsize = 7

    A = np.zeros( [ matrixsize, matrixsize ] )
    PrintMatrix("A", A);

    print("---");
    print("--- Part 2: Compute remainders: x^2 + y^2 is divided by 7 ...");
    print("---");

    for i in range( A.shape[0] ):           # <-- loop over the rows ...
        for j in range( A.shape[1] ):       # <-- loop over the columns ...
            A[i][j] = (i**2 + j**2) % 7;

    print("---");
    print("--- Part 3: Matrix of remainder computations ...");
    print("---");

    PrintMatrix("A[i][j] = (i^2 + j^2)%7 ", A);

    print("---");
    print("--- Part 4: Compute expression for (i,j) 0 through 100 ...");
    print("---");

    for i in range( 101 ):
        for j in range( 101 ):
            rem01 = (i**2 + j**2) % 7;
            rem02 = (i**2 + j**2) % 49;

            # Print details when expression is divisible by 7 ...

            if rem01 == 0:
               print("--- (i,j) = ({:2d},{:2d}): (i^2 + j^2)%7  --> {:3.1f} ... ".format(i,j, rem01))

            # Print additional details when expression is also divisible by 49 ...

            if rem01 == 0 and rem02 == 0:
               print("---                 : (i^2 + j^2)%49 --> {:3.1f} ... ".format(rem02));

    print("--- ======================================== ... ");
    print("--- Leave TestBrainTeaser.main()             ... ");

# call the main method ...

if __name__ == "__main__":
    main()
```

**Program Output:** The abbreviated textual output is:

```
--- Enter TestBrainTeaser.main()              ...
--- ======================================== ...
---
--- Part 1: Setup (7x7) matrix ...
---
Matrix: A
row/col              1             2             3             4             5             6
   1       0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00
   2       0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00
   3       0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00
   4       0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00
   5       0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00
   6       0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00
   7       0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00

Matrix: A
row/col              7
   1       0.00000e+00
   2       0.00000e+00
   3       0.00000e+00
   4       0.00000e+00
   5       0.00000e+00
   6       0.00000e+00
   7       0.00000e+00

---
--- Part 2: Compute remainders: x^2 + y^2 is divided by 7 ...
---
--- Part 3: Matrix of remainder computations ...
---
Matrix: A[i][j] = (i^2 + j^2)%7
row/col              1             2             3             4             5             6
   1       0.00000e+00   1.00000e+00   4.00000e+00   2.00000e+00   2.00000e+00   4.00000e+00
   2       1.00000e+00   2.00000e+00   5.00000e+00   3.00000e+00   3.00000e+00   5.00000e+00
   3       4.00000e+00   5.00000e+00   1.00000e+00   6.00000e+00   6.00000e+00   1.00000e+00
   4       2.00000e+00   3.00000e+00   6.00000e+00   4.00000e+00   4.00000e+00   6.00000e+00
   5       2.00000e+00   3.00000e+00   6.00000e+00   4.00000e+00   4.00000e+00   6.00000e+00
   6       4.00000e+00   5.00000e+00   1.00000e+00   6.00000e+00   6.00000e+00   1.00000e+00
   7       1.00000e+00   2.00000e+00   5.00000e+00   3.00000e+00   3.00000e+00   5.00000e+00

Matrix: A[i][j] = (i^2 + j^2)%7
row/col              7
   1       1.00000e+00
   2       2.00000e+00
   3       5.00000e+00
   4       3.00000e+00
   5       3.00000e+00
   6       5.00000e+00
   7       2.00000e+00

---
--- Part 4: Compute expression for (i,j) 0 through 100 ...
---
```

```
--- (i,j) = ( 0, 0): (i^2 + j^2)%7  --> 0.0 ...
---                : (i^2 + j^2)%49 --> 0.0 ...
--- (i,j) = ( 0, 7): (i^2 + j^2)%7  --> 0.0 ...
---                : (i^2 + j^2)%49 --> 0.0 ...
--- (i,j) = ( 0,14): (i^2 + j^2)%7  --> 0.0 ...
---                : (i^2 + j^2)%49 --> 0.0 ...

... lines of output removed ...

--- (i,j) = (98,84): (i^2 + j^2)%7  --> 0.0 ...
---                : (i^2 + j^2)%49 --> 0.0 ...
--- (i,j) = (98,91): (i^2 + j^2)%7  --> 0.0 ...
---                : (i^2 + j^2)%49 --> 0.0 ...
--- (i,j) = (98,98): (i^2 + j^2)%7  --> 0.0 ...
---                : (i^2 + j^2)%49 --> 0.0 ...

--- ======================================== ...
--- Leave TestBrainTeaser.main()             ...
```

**Proof.** From the first part of the hint we have:

$$p^2 = [7 * p_1 + r_1]^2 = 49p_1^2 + 14p_1r_1 + r_1^2 \tag{4}$$

and

$$q^2 = [7 * q_1 + r_2]^2 = 49q_1^2 + 14q_1r_2 + r_2^2. \tag{5}$$

Adding equations 4 and 5 gives:

$$p^2 + q^2 = 49\left[p_1^2 + q_1^2\right] + 14\left[p_1r_1 + q_1r_2\right] + r_1^2 + r_2^2. \tag{6}$$

Now we need to determine when equation 6 will be divisible by 7. The first term is a multiple of 49 and so it is automatically divisible by 7. Similarly, the second term will be divisible by 7 because it is a multiple of 14. Hence for the overall expression to be divisible by 7, we require that $r_1^2 + r_2^2$ be divisible by 7, i.e., $r_1^2 + r_2^2$ modulo 7 will evaluate to zero.

From the output for matrix `A` (above) we see that the only way that this will occur is when $r_1 = r_2 = 0$. Under these circumstances the second and third terms in equation 6 vanish, leaving

$$p^2 + q^2 = 49\left[p_1^2 + q_1^2\right]. \tag{7}$$

which will always be divisible by 49.

**Question 4: 10 points.** Figure 1 is a schematic of an irregular polygon having seven sides.

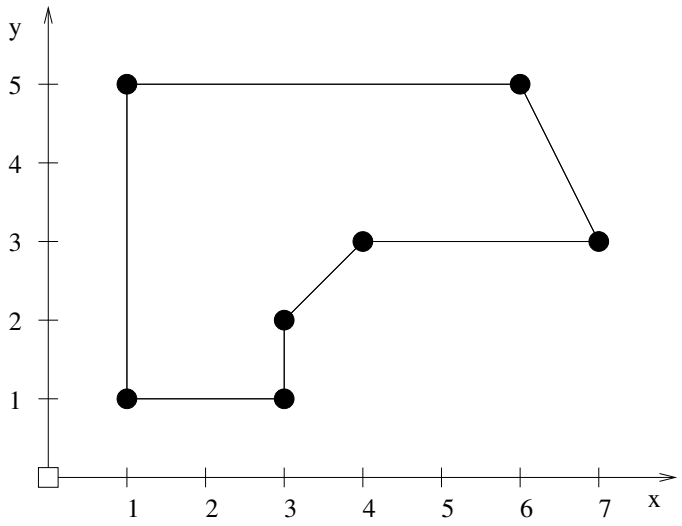


Figure 1: Seven-sided irregular polygon.

Suppose that the $x$ and $y$ vertex coordinates are stored as two columns of information in the array

```
coord = np.array( [ ( 1.0, 1.0 ),
                    ( 1.0, 5.0 ),
                    ( 6.0, 5.0 ),
                    ( 7.0, 3.0 ),
                    ( 4.0, 3.0 ),
                    ( 3.0, 2.0 ),
                    ( 3.0, 1.0 ) ] );
```

Write a Python program that will compute and print

1. The minimum and maximum polygon coordinates in both the $x$ and $y$ directions.
2. The minimum and maximum distance of the polygon vertices from the coordinate system origin.
3. Write functions perimeter() and area() to compute the perimeter and area of the polygon, respectively.

**Note.** For Parts 1 and 2, use the `max()` and `min()` methods in python. In Part 3, use the fact that the vertices have been specified in a clockwise manner.

**Python Source Code:**

```
# ================================================================
# TestPolygonSevenSided.py: Compute geometric properties of a
# seven-sided polygon.
#
# Written by: Mark Austin                                January 2024
# ================================================================

import math
import numpy as np
from numpy.linalg import matrix_rank

# ================================================================
# Function to print two-dimensional matrices ...
# ================================================================

def PrintMatrix(name, a):
    print("Matrix: {:s} ".format(name) );
    for row in a:
        for col in row:
            print("{:8.4f}".format(col), end=" ")
        print("")

# ================================================================
# polygon perimeter and area ...
# ================================================================

# Compute polygon perimeter ...

def perimeter( coord ):
    norows = coord.shape[0];

    dperimeter = 0.0;
    for i in range(1,norows):
        dx = coord[i][0] - coord[i-1][0];
        dy = coord[i][1] - coord[i-1][1];
        dperimeter = dperimeter + math.sqrt( dx*dx + dy*dy );

    dx = coord[norows-1][0] - coord[0][0];
    dy = coord[norows-1][1] - coord[0][1];
    dperimeter = dperimeter + math.sqrt( dx*dx + dy*dy );

    return dperimeter;

# Compute polygon area ...

def area( coord ):
    norows = coord.shape[0];

    darea = 0.0;
    for i in range(1,norows):
        dx = coord[i][0] - coord[i-1][0];
        dy = coord[i][1] + coord[i-1][1];
        darea = darea + dx*dy/2.0;

    dx = coord[0][0] - coord[norows-1][0];
```

```
    dy = coord[0][1] + coord[norows-1][1];
    darea = darea + dx*dy/2.0;

    return darea;

# ================================================================
# main method ...
# ================================================================

def main():
    print("--- Enter TestPolygonSevenSided.main()       ... ");
    print("--- ======================================== ... ");

    print("--- ");
    print("--- Part 1: Initialize coefficients for matrix equations ... ");

    coord = np.array( [ ( 1.0, 1.0 ),
                        ( 1.0, 5.0 ),
                        ( 6.0, 5.0 ),
                        ( 7.0, 3.0 ),
                        ( 4.0, 3.0 ),
                        ( 3.0, 2.0 ),
                        ( 3.0, 1.0 ) ] );

    PrintMatrix("Polygon Coordinates", coord);

    print("--- ");
    print("--- Part 2: Max/min coordinate positions ... ");
    print("--- ");

    print("--- Min x = {:f} ...".format( min ( coord[:,0] ) ) );
    print("--- Max x = {:f} ...".format( max ( coord[:,0] ) ) );
    print("--- Min y = {:f} ...".format( min ( coord[:,1] ) ) );
    print("--- Max y = {:f} ...".format( max ( coord[:,1] ) ) );

    print("--- ");
    print("--- Part 3: Compute and print distance of coords from origin ... ");
    print("--- ");

    distance = np.zeros( coord.shape[0] )
    for i in range(7):
        x = coord[i][0];
        y = coord[i][1];
        distance[i] = math.sqrt(x**2 + y**2)

    print(distance)

    print("--- Min distance from origin = {:f} ...".format( min ( distance ) ) );
    print("--- Max distance from origin = {:f} ...".format( max ( distance ) ) );

    print("--- ");
    print("--- Part 4: Compute and print perimeter and area of polygon ... ");
    print("--- ");

    print("--- Polygon perimeter = {:f} ...".format( perimeter(coord) ) );
```

```
    print("--- Polygon area      = {:f} ...".format( area(coord) ) );

    print("--- ======================================== ... ");
    print("--- Leave TestPolygonSevenSided.main()       ... ");

# call the main method ...

if __name__ == "__main__":
    main()
```

**Program Output:** The textual output is:

```
--- Enter TestPolygonSevenSided.main()       ...
--- ======================================== ...
---
--- Part 1: Initialize coefficients for matrix equations ...

Matrix: Polygon Coordinates
  1.0000   1.0000
  1.0000   5.0000
  6.0000   5.0000
  7.0000   3.0000
  4.0000   3.0000
  3.0000   2.0000
  3.0000   1.0000

---
--- Part 2: Max/min coordinate positions ...
---

--- Min x = 1.000000 ...
--- Max x = 7.000000 ...
--- Min y = 1.000000 ...
--- Max y = 5.000000 ...

---
--- Part 3: Compute and print distance of coords from origin ...
---

[1.41421356 5.09901951 7.81024968 7.61577311 5.0 3.60555128 3.16227766]

--- Min distance from origin = 1.414214 ...
--- Max distance from origin = 7.810250 ...
---
--- Part 4: Compute and print perimeter and area of polygon ...
---
--- Polygon perimeter = 18.650282 ...
--- Polygon area      = 15.500000 ...

--- ======================================== ...
--- Leave TestPolygonSevenSided.main()       ...
```

**Question 5: 10 points.** Write a Python program that will compute and print a list of $(x, y)$ pairs for:

$$y(x) = \left[ \frac{x^2 + \left[ \frac{x}{sin(x)} \right]}{x - 2} \right] \tag{8}$$

over the range $-10 \leq x \leq 10$ and in intervals of 0.25.

**Hint:** You should find that $y(0)$ and $y(2)$ evaluate to not-a-number (NaN) and positive infinity, respectively, and that Python 3 provides remarkably good builtin support for handling of run-time errors. Create a plot of $y(x)$ vs $x$ – you should find that errors will be automatically handled within the matplotlib.pyplot environment.

**Python Source Code:**

```
# ============================================================
# TestDifficultFunction02.py: Compute difficult function and see
# how Python handles divide by zero and NaN at runtime.
#
# Written by: Mark Austin                                February 2024
# ============================================================

import math
import numpy as np
import matplotlib.pyplot as plt

# Implement test function, ignore difficulties ...

def testFunction(x):
    result = (x**2 + (x/math.sin(x)))/(x-2)
    return result

# main method ...

def main():
    print("--- Enter TestDifficultFunction02.main()     ... ");
    print("--- ======================================== ... ");
    print("");

    print("==============================");
    print("          Coord          Value");
    print("            (x)            (y)");
    print("==============================");

    # Define problem parameters.

    xcoord = np.linspace(-10,10,81)

    # Create list for y coordinates ...
```

```
    ycoord = [];

    # Traverse xcoord array and compute y values ...

    for x in xcoord:
        result = testFunction(x)
        print("        {:7.2f}   {:12.3f}".format(x,result) );
        ycoord.append(result)

    print("==============================");
    print("");

    # Plot y vs x for test function ...

    plt.plot(xcoord,ycoord)
    plt.title('plot y vs x for difficult function')
    plt.ylabel('y')
    plt.xlabel('x')
    plt.xlim( -10, 10)
    plt.grid(True)
    plt.show()

    print("--- ====================================== ... ");
    print("--- Leave TestDifficultFunction02.main()     ... ");

# call the main method ...

if __name__ == "__main__":
   main()
```

**Program Output:** The abbreviated textual output is:

```
--- Enter TestDifficultFunction02.main()     ...
--- ====================================== ...

==============================
          Coord          Value
            (x)            (y)
==============================
         -10.00         -6.802
          -9.75         -5.493
          -9.50          3.145

        ... output removed ...

          -0.50         -0.517
          -0.25         -0.477
           0.00            nan
           0.25         -0.613
           0.50         -0.862

        ... output removed ...
```

```
            1.75        -19.364
            2.00            inf
            2.25         31.817

         ... output removed ...

            9.50         -4.822
            9.75          8.329
           10.00         10.202
==============================

--- ======================================== ...
--- Leave TestDifficultFunction02.main()     ...
```

The graphical output is: