MAE 5930 - Optimization

Fall 2019

Homework 5

Jared Hansen

A-number: A01439768

e-mail: jrdhansen@gmail.com

Due: 11:59 PM, Thursday October 24, 2019

Purpose: the problems assigned help develop your ability to

- recognize and formulate integer programs.
- convert formulations into code using MATLAB's `intlinprog`.
- use piecewise linear approximations to solve non-convex programs to global optimality.
- read a paper on optimization.

NOTE: please write or type your formulations clearly so that a reader can understand what you are doing. You are welcome to use the equivalent functions in Python.

1. Consider the following nonlinear, nonconvex optimization problem:

$$\min_x f(x) = x^2 \text{ subject to } g(x) = x^2 sin(x) \leq 0 \text{ on } x \in [1, 12]$$

(a) Plot the functions and visually confirm that the optimal point is $x \approx 3$.

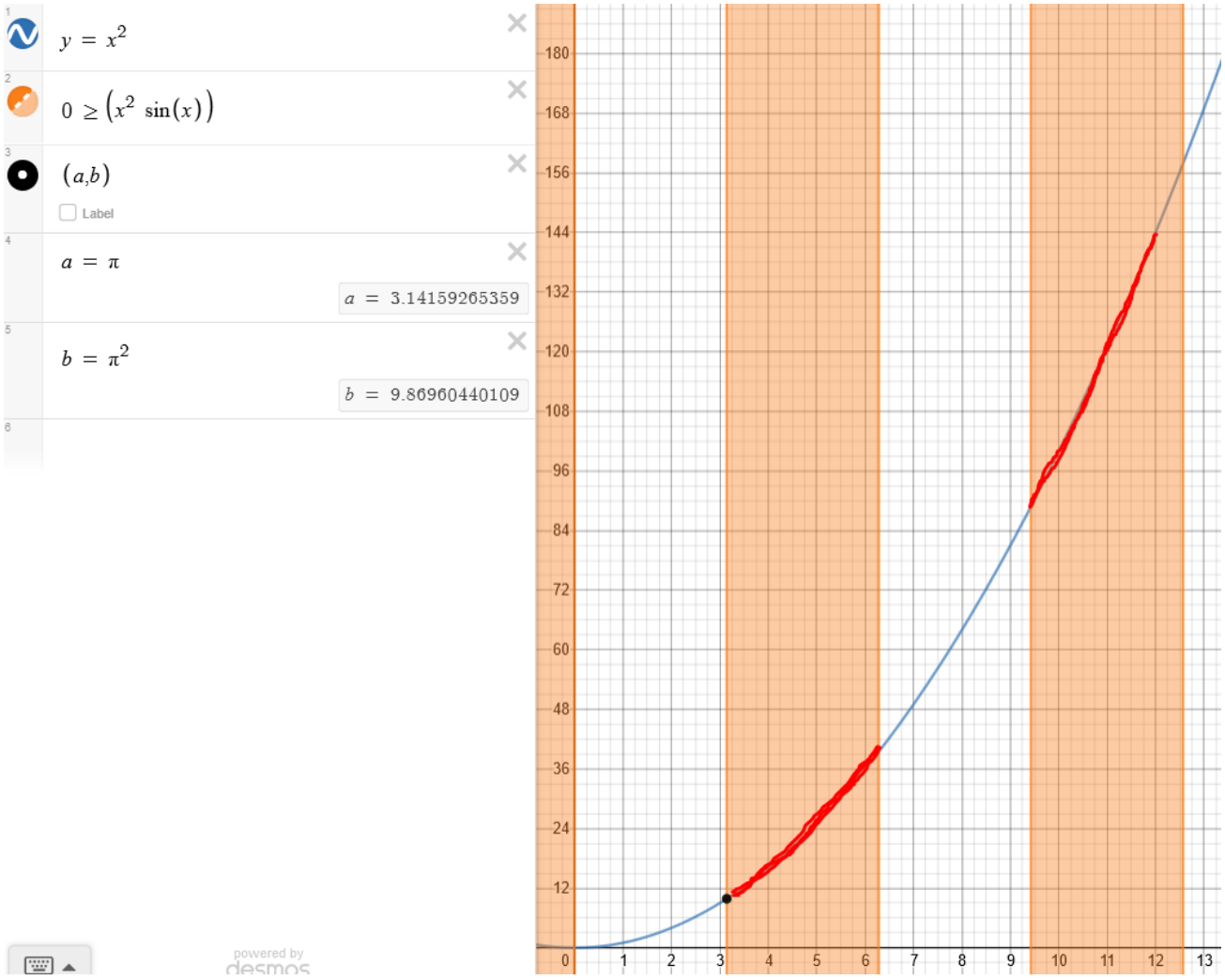

Figure 1: Graphical plot of the optimization problem. See the bullet points below for further discussion, or MATLAB plot of just the functions on the next page.

- The blue line represents $f(x) = x^2$ and the orange areas are where $\left[g(x) = x^2 sin(x)\right] \leq 0$.
- A feasible point must be: on the "blue line" in an "orange area" where $x \in [1, 12]$. Therefore, the collection of points that fall on the red-highlighted segments of f are candidate minimizers since they satisfy $f(x) = x^2$, they satisfy $g(x) \leq 0$, and they are $\in [1, 12]$.

- **As the prompt specifies, we can see that this occurs at $x \approx 3$, which we can deduce is $x^* = \pi$ (see the black point in the graphic). We know that this is π since:**
  - **$\left[g(\pi) = \pi^2 sin(\pi) = \pi^2(0) = 0\right] \leq 0$ so the constraint $g(x) \leq 0$ is satisfied**
  - **$f(\pi) = \pi^2$ is a valid (input, output) pair of the function f**
  - **$\pi \in [1, 12]$**
  - **We can visually observe that $x^* = \pi$ the minimum of the set of feasible points (red highlighted segments above)**

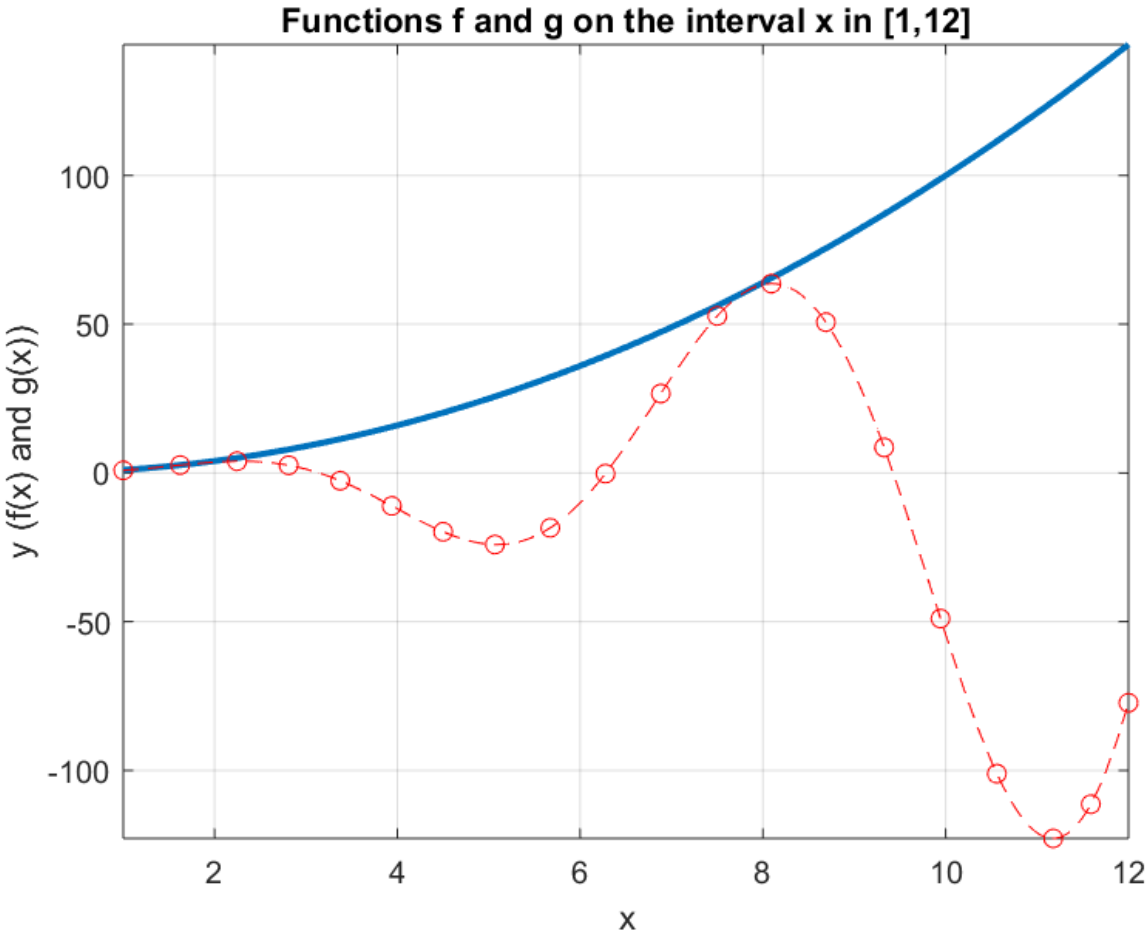

Figure 2: Graphical plot of the optimization problem with just the functions $f(x)$ and $g(x)$ (not explicitly plotting the constraint $g(x) \leq 0$, but rather just the function g itself). We can see that the first place on $x \in [1, 12]$ for which $g(x) \leq 0$ is at $x \approx 3$, so this will be where we minimize f subject to $g(x) \leq 0$. In other words, $x \approx 3$ is the least element of $x \in [1, 12]$ for which $g(x) \leq 0$ is satisfied, therefore it will be our minimizer.

(b) Solve the problem using MATLAB's `fmincon` with initial guesses $x = 1, 5$, and 9. Comment if the optimal solution is found for any of these initial guesses.

- The first image below is the code I used to attempt solving the problem with `fmincon`. Note that the nonlinear constraint function must be placed in its own Matlab file. I simply included it in this screenshot (implying all the code is in one file, which it isn't) for ease in visualizing all needed code in one image.
- Below is an image of the results of initial guesses $x_0 \in \{1, 5, 9\}$. See further discussion below the image itself.

```
%=================================================================
%== 1B: solve the problem using fmincon, x0 in {1,5,9}
%=================================================================

% NOTE: the nonlinear constraint must be defined in its own .m file.
%       I've only included it here for visualizing's sake. This code
%       wouldn't run as-is. Put the 4 lines below in their own file.
function [c,ceq] = ineqConFun(x)
c = x^2*sin(x);    % defining nonlinear ineq. constraint function
ceq = [];          % defining nonlinear, eq. constraint function (n/a)
end;
obj = @(x) x^2 ;  % defining the objective function
nonlcon = @ineqConFun;  % specifying nonlinear constraints
% Solutions for different guesses of x0: 1, 5, and 9
xOptim_guess1 = fmincon(obj,1,[],[],[],[],1,12,nonlcon);
xOptim_guess5 = fmincon(obj,5,[],[],[],[],1,12,nonlcon);
xOptim_guess9 = fmincon(obj,9,[],[],[],[],1,12,nonlcon);
```

Figure 3: MATLAB code used to attempt solving the optimization problem using `fmincon` (see notes above about how code should actually be organized in separate files).

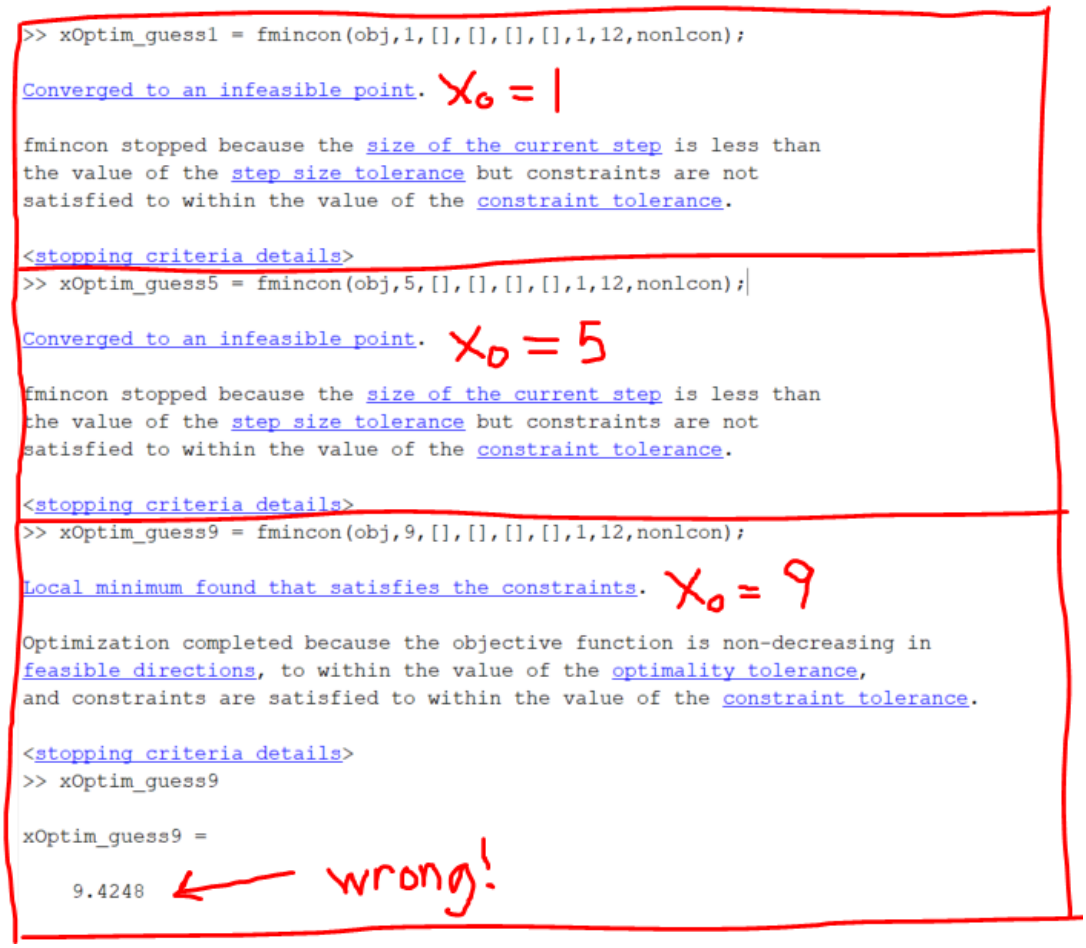

Figure 4: Results of code attempting to solve the optimization problem using `fmincon`. See bullets below for discussion.

- We can see from the MATLAB output in Figure 4 that both initial guesses $x_0 = 1$ and $x_0 = 5$ result in failure to converge to a feasible point. Therefore, neither of these initial guesses result in `fmincon` finding the optimal solution. Both of them converged to $x = 1$ which is the minimum of f for $x \in [1, 12]$ BUT it doesn't satisfy the $g(x) \leq 0$ constraint, and is therefore an infeasible point.
- This seems like it would make sense for $x_0 = 1$ since 1 isn't in a region for which the $g(x) \leq 0$ constraint is satisfied (see Figure 1).
- However, this logic doesn't hold for $x_0 = 5$. Since $x = 5$ is in a region for which $g(x) \leq 0$ (again, see Figure 1) I would've guessed that it would converge to the least x in that "orange band" (area where $g(x) \leq 0$) of $x = \pi$ but unfortunately it doesn't.
- Oddly enough, the logic that I thought would've played out for $x_0 = 5$ (converging to the least x value for that "orange band" of $g(x) \leq 0$) does hold for $x_0 = 9$ even though $x = 9$ isn't in a region that satisfies $g(x) \leq 0$. This time, `fmincon` returns a value of $x = 9.4248$, which is $x = 3\pi$, a local minimum very close in distance to $x_0 = 9$. We can see that this is a feasible point, but it is certainly not the minimum of $x^* = \pi$ that we graphically identified in part A.
- **In summary, all initial guesses $x_0 \in \{1, 5, 9\}$ fail to converge to $x^* = \pi$ when using `fmincon` to solve.**

(c) Formulate the problem as a MILP by piecewise linearly approximating $f(x)$ and $g(x)$.

- Start by defining n nodes along the x-axis in the given domain (here, $x \in [1, 12]$). Thus we have $x_1, x_2, \ldots, x_n$.
- This gives us corresponding discrete y values of our objective function: $f(x_1), f(x_2), \ldots, f(x_n) = y_1, y_2, \ldots, y_n$.
- Also, we'll need to have corresponding discrete z values of our nonlinear, inequality constraint function $g(x)$: $g(x_1), g(x_2), \ldots, g(x_n) = z_1, z_2, \ldots, z_n$.
- Now let's introduce $n-1$ binary variables, b_i, used to indicate whether a segment is active. (We have n nodes, therefore we have $n-1$ segments and one binary for each of these $n-1$ segments.)
- Only one of these intervals can be active, therefore we have the constraint $\sum_{i=1}^{n-1} b_i = 1$.
- Now we introduce a weight for each node, w_i, for $i \in \{1, 2, \ldots, n\}$.
- If an interval is active, we can have x at somewhere along that interval (it doesn't have to be exactly at x_1 or x_2, etc.) For example, if $w_3 = 0.7$ and $w_4 = 0.3$ that means we are active on the third segment, closer to the x_3 node than to the x_4 node.
- Formalize this with constraints: $\left[0 \leq w_i \leq 1\right], \left[\sum_{i=1}^{n} w_i = 1\right]$
- Additionally, weights can only be active (non-zero) on an active (non-zero) interval. We introduce SOS (special ordered set) constraints to handle this:

$$\begin{aligned} w_1 &\leq b_1 \\ w_2 &\leq b_1 + b_2 \\ w_3 &\leq b_2 + b_3 \\ &\vdots \\ w_n &\leq b_{n-1} \end{aligned}$$

- We need to add an additional constraint now that we have defined the weights. We will define $\left[z := w_1 z_1 + w_2 z_2 + \ldots + w_n z_n\right]$. Re-writing the constraint $g(x) \leq 0$, we can say that $\left[g(x) \leq 0\right] \equiv \left[z \leq 0\right]$
- Now we can define our optimal x and y as:
  $\left[x = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n\right]$
  $\left[y = w_1 y_1 + w_2 y_2 + \ldots + w_n y_n\right]$
- **Our final formulation is:** $\left[\min f(x) = \min y = \min \sum_{i=1}^{n} (w_i)(y_i)\right]$ **subject to:**
  - i. $\left[b_i \in \{0, 1\} \forall i \in \{1, 2, \ldots, n-1\}\right]$
  - ii. $\left[w_i \in [0, 1] \forall i \in \{1, 2, \ldots, n\}\right]$
  - iii. $\left[\sum_{i=1}^{n-1} b_i = 1\right]$
  - iv. $\left[\sum_{i=1}^{n} w_i = 1\right]$
  - v. $\left[x = \sum_{i=1}^{n} (w_i)(x_i)\right]$
  - vi. $\left[y = \sum_{i=1}^{n} (w_i)(y_i)\right]$
  - vii. $\left[z = \sum_{i=1}^{n} (w_i)(z_i)\right]$
  - viii. $\left[z \leq 0\right]$
  - ix. $\left[\text{ SOS (special ordered set) constraints (see above) }\right]$

(d) Solve the MILP using 20 nodes, 50 nodes, and 100 nodes. Please provide your code and output showing the correct answer. Comment on any observations you have.

```
%=========================================================================
%== 1D: solve the MILP using 20, 50, and 100 nodes.
%=========================================================================

% CREATE THE NODE POINTS
n = 20;                 % How many nodes ?
x = linspace(1,12,n);   % Creating discrete x    values
y = x.^2;               % Creating discrete f(x) values
z = x.^2.*sin(x);       % Creating discrete g(x) values

% SET UP THE OPTIMIZATION PROBLEM
prob = optimproblem('ObjectiveSense','minimize');

% DEFINE VARIALBLES (BINARIES AND WEIGHTS)
b = optimvar('b',n-1,1, 'Type','integer',    'LowerBound',0,'UpperBound',1);
w = optimvar('w',n,1,   'Type','continuous','LowerBound',0,'UpperBound',1);

% DEFINE THE OBJECTIVE FUNCTION
prob.Objective = sum( y*w );

% DEFINE CONSTRAINTS
prob.Constraints.bsum = sum(b) == 1;      % all binaries must sum to 1
prob.Constraints.wsum = sum(w) == 1;      % all weights  must sum to 1
prob.Constraints.gIneq = sum(z*w) <= 0;   % discretization of g(x) <= 0

% CREATE SOS (Special Ordered Set) CONSTRAINTS
SOS = zeros(n,n-1);
SOS([1:n+1:end]) = 1;
SOS([2:n+1:end]) = 1;
prob.Constraints.sos = SOS*b >= w;

% DETERMINE THE SOLUTION
sol = solve(prob)
xsol = x*sol.w
ysol = y*sol.w

% PLOT RESULTS
plot(x,y,'+'), grid on, hold on
plot(x,z,'*'), grid on
plot(xsol,ysol,'ko','markerfacecolor','k')
xlabel('x'), ylabel('f(x)=o and g(x)=*')
title(strcat(['Piecewise Linear Approx solution with ', int2str(n), ' nodes']));
```

Figure 5: Here is the MATLAB code used to generate my solutions. NOTE that the "n" value at the top of this code is the only thing that needs to be changed. I ran it for $n \in \{20, 50, 100\}$ as instructed, output below.

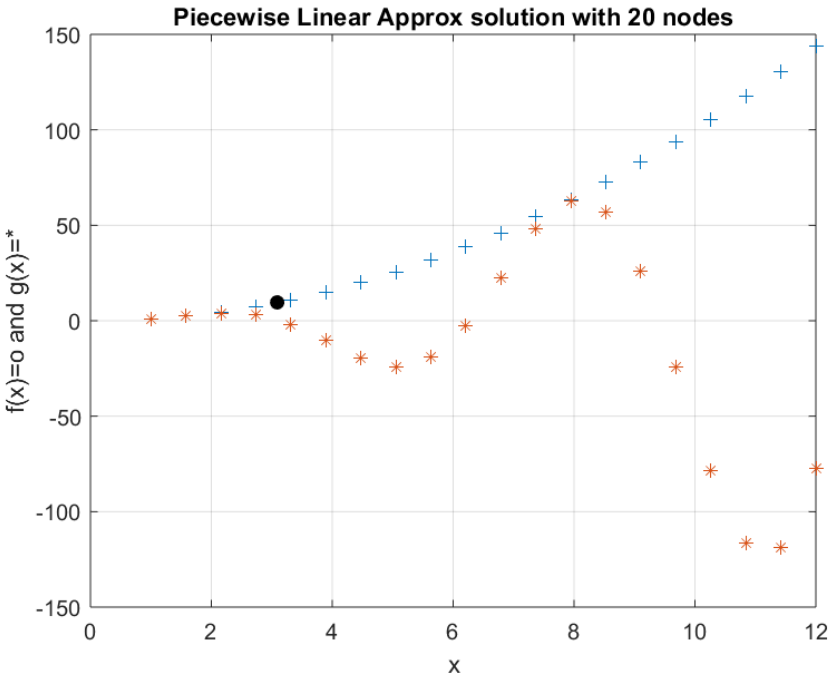

Figure 6: Plot for 20-node approximation (red asterisks are $g(x)$ values, blue plusses are $f(x)$ values, black dot is approximated solution).

```
Optimal x, f(x) for 20 nodes

xsol =

    3.0886


ysol =

    9.6192
```

Figure 7: Optimal values for 20-node approximation.

See below the bottom-most plot (100-node approximation) for comments on piecewise linear approximation solutions.

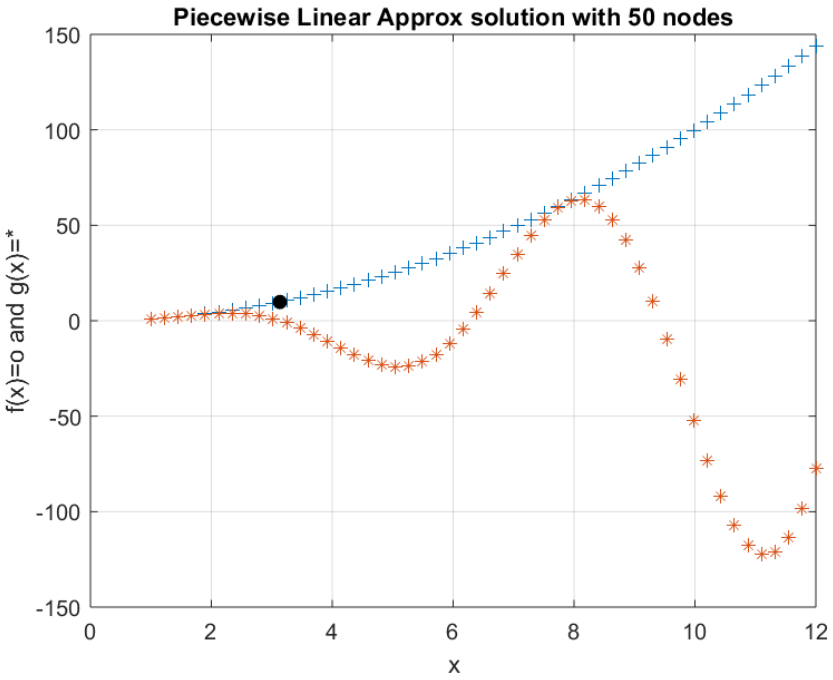

Figure 8: Plot for 50-node approximation (red asterisks are $g(x)$ values, blue plusses are $f(x)$ values, black dot is approximated solution).

```
Optimal x, f(x) for 50 nodes

xsol =

    3.1335


ysol =

    9.8316
```

Figure 9: Optimal values for 50-node approximation.

See below the bottom-most plot (100-node approximation) for comments on piecewise linear approximation solutions.

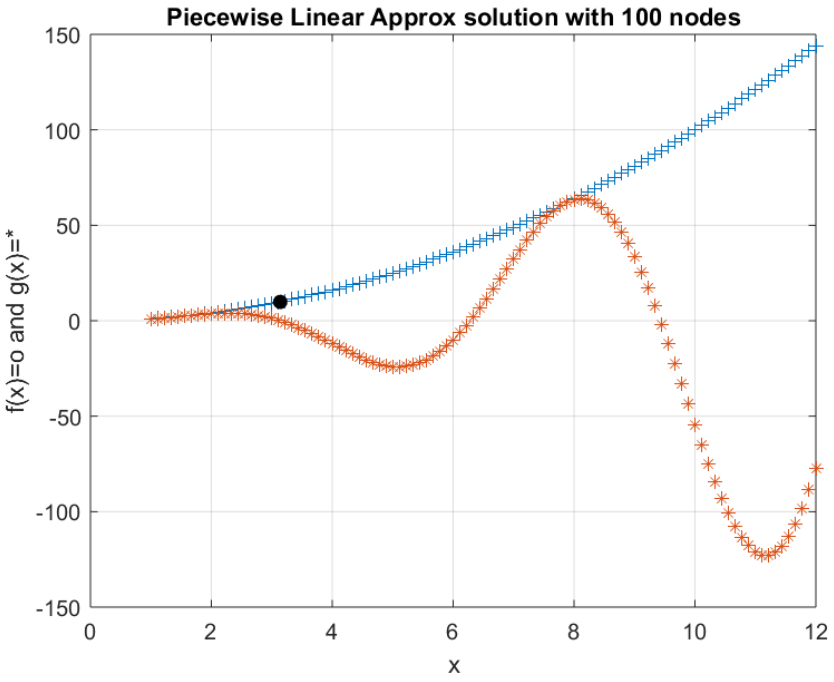

Figure 10: Plot for 100-node approximation (red asterisks are $g(x)$ values, blue plusses are $f(x)$ values, black dot is approximated solution).

```
Optimal x, f(x) for 100 nodes

xsol =

    3.1401


ysol =

    9.8625
```

Figure 11: Optimal values for 100-node approximation.

Observations regarding piecewise linear approximation solutions:

- We can see that the approximation improves in precision as the number of nodes increases. This is a good thing to see, as it's what our intuition would be.
- Phrased mathematically, as n increases, $|x^* - \pi| \to 0$ where x^* is the x that optimizes the given problem for n discrete nodes.
- Another thing to observe is that none of the given solutions are actually feasible. For example, even the most precise solution $x^* = 3.1401$ when input into the nonlinear inequality constraint function $g(x)$ gives $g(3.1401) = (sin(3.1401)(3.1401)^2) = 0.014718 \nleq 0$.
- From our graphical solution in part A, we know that $x^* = \pi$. These solutions come closer and closer to π, but since they are coming from an infeasible region, $x \in [1, \pi)$, they are all technically infeasible. But we can see this and make the necessary adjustment.

2. Formulate and solve the traveling salesman problem.

(a) Formulate the problem by defining the decision variables, objective, and constraints. As shown in class, the most obvious formulation does not avoid "sub-tours". Think of constraints you can add to avoid sub-tours. If you get tired of thinking, read the attached paper and implement their constraints.

- In my Python implementations of the traveling salesman problem (for both the PuLP and Gurobi packages) I employed the "sub-tour formulation" from the paper by Pataki recommended by Dr. Harris. I will describe this formulation here.
- **DECISION VARIABLES:**
  Define decision variables as $x_{i,j} := \left\{ \begin{array}{ll} 1 & \text{if go city}_i \rightarrow \text{city}_j \\ 0 & \text{if don't go city}_i \rightarrow \text{city}_j \end{array} \right\} \forall i, j$ where $i \neq j$, and $i, j \in \{1, 2, \ldots, n\}$
  To make this more familiar, we can think about our decision variables being organized in a matrix $X_{n \times n}$, where the entry $x_{i,j}$ is as described above. For example, if $x_{1,3} = 1$ this would mean that after the salesman travels from city 1 to city 3. (This matrix isn't completely necessary for representation of the variables, but it helps us nail down the concept more concretely. If we did use this, all diagonal entries would necessarily be 0 since we can't go from a city back to itself.)

- **OBJECTIVE:**
  Our objective is: $\min_x \sum_{i,j}^{n} (x_{i,j})(d_{i,j})$ where $d_{i,j}$ is the straight-line distance between city_i and city_j, and $x_{i,j}$ indicates whether the salesman departs-from city_i and arrives-to city_j. This means that we are minimizing the distance traveled in order to visit each city exactly once. (For sake of completeness in the code, we then go from the final city back to the origin city to 'complete the tour.')

- **CONSTRAINTS:**
  Assignment Constraints:
  - Thought of in terms of graph theory, this would mean that each node has exactly one incoming-directed edge and exactly one outgoing-directed edge.
  - Thought of in terms of cities and salesman, this would mean that each city is arrived-to exactly once by the salesman, and each city is departed-from exactly once by the salesman.
  - Mathematically:
    i. $\left[\sum_{i=1}^{n} x_{i,j} = 1, \forall j \right]$ ("each city arrived-to exactly once" (summing for each column of X matrix))
    ii. $\left[\sum_{j=1}^{n} x_{i,j} = 1, \forall i \right]$ ("each city departed-from exactly once" (summing for each row of X matrix))

  Sub-tour Elimination Constraints:
  - A sub-tour is a 'directed cycle' in a graph. A directed cycle is a non-empty directed trail in which the first and last vertices are repeated (may be one and the same).
  - In the context of this problem, that would involved the salesman coming back through a city which he's already visited. This violates our assignment constraint of departing-from a city only once since he'd have already been there, meaning he's also already departed from there once.
  - To prevent this more strictly than just with our assignment constraints:
    $\left[\sum_{i \in S, j \in S} x_{i,j} \leq |S| - 1 \right]$ where $\left[\big(S \subset V\big), \big(|S| > 1\big), \big(\forall i \neq 1, \forall j \neq 1\big) \right]$
  - Here, S is a set of unique cities. V is another (different from S) set of unique cities. $\big(S \subset V\big)$ means that S is a proper subset of V (all cities in S are also in V, but there is at least one city in V that isn't in S).
  - Also, here $|S|$ means the cardinality (number of elements) in S rather than absolute value.
  - This constraint effectively says that, for the cities in S, the number of elements in S minus 1 must be at least the sum of the binary variables for all of the cities in S. Since the sum is one less than the number of cities, this prevents the salesman from looping back to the first (or other) cities in S, thus eliminating any sub-tours for the set of cities in S.
  - This constraint does introduce some redundancy since there are many sets S and V that are possible for all cities. This is taken care of by a *separation algorithm* which we didn't talk about, nor does the Pataki paper. (The Python code will take care of this for us though.)

(b) Solve the problem where the locations of the cities are:
(0.0, 0.0), (1.0, 1.0), (2.0, 0.1), (10.0, -0.1), (11.0, 1.0), (12.0, 0.0).

NOTE: since the beginning of the assignment says "You are welcome to use the equivalent functions in Python" that is what I did. See Python code at the bottom of this PDF.
In this case, I used the PuLP library from Python to do the solving (I also used this for the 20 and 40 city problems, uses the same code, but with custom-made points as given above.)

```
The solver took 0:00:00.199336 to run for 6 cities.
(time is in hours : minutes : seconds.frac_of_second)
Optimal
```

Figure 12: How long it took Python formulation to solve the 6-city problem.

```
Path travelled
--------------
  0  to  2
  2  to  3
  3  to  5
  5  to  4
  4  to  1
  1  to  0
Total distance traveled is: 24.8359 units
```

Figure 13: The order of the optimal tour.

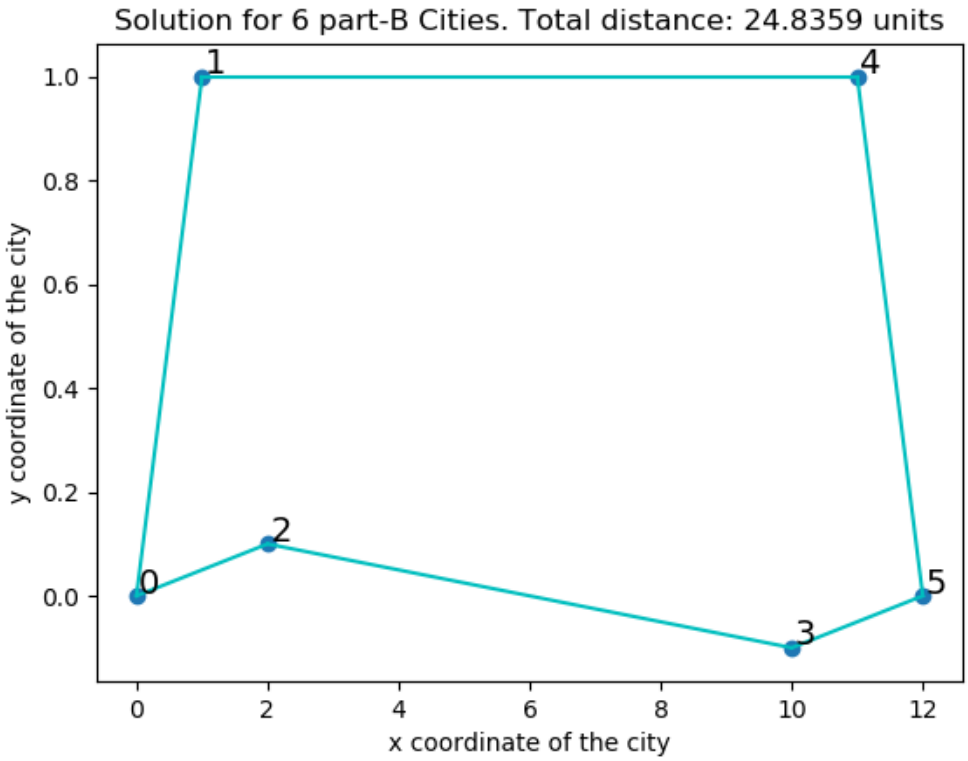

Figure 14: The plot of the optimal routing for the 6-city problem (see above for order).

(c) Solve the problem with 20 cities. Show the output and plot your solution.

NOTE: since we "... are welcome to use the equivalent functions in Python" I used the PuLP library from Python to do the solving. See code at bottom of PDF.

```
The solver took 0:00:05.106297 to run for 20 cities.
(time is in hours : minutes : seconds.frac_of_second)
Optimal
```

Figure 15: How long it took Python formulation to solve a random 20-city problem.

```
Path travelled
--------------
  0  to  8
  8  to  5
  5  to  10
  10  to  1
  1  to  15
  15  to  14
  14  to  6
  6  to  19
  19  to  17
  17  to  16
  16  to  11
  11  to  13
  13  to  9
  9  to  3
  3  to  12
  12  to  18
  18  to  4
  4  to  2
  2  to  7
  7  to  0
Total distance traveled is: 72.2049 units
```

Figure 16: The order of the optimal tour.

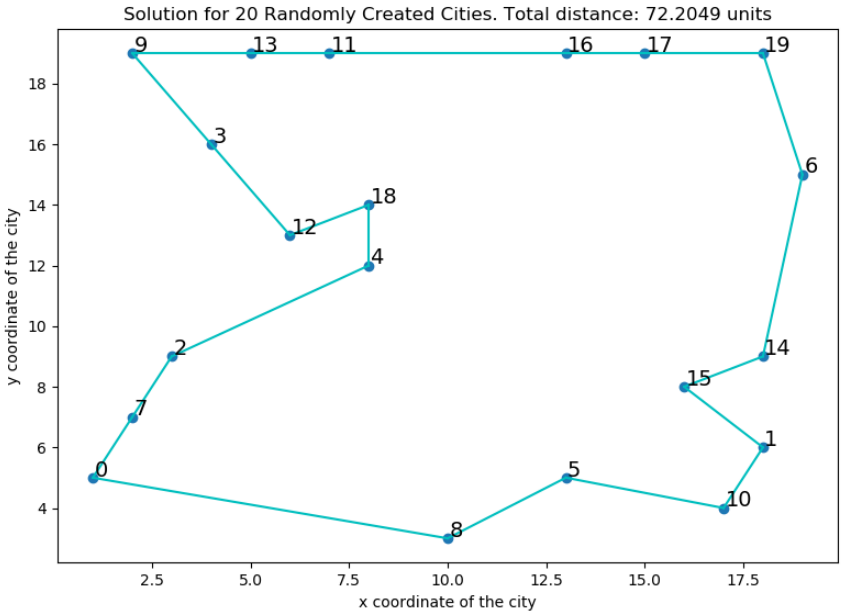

Figure 17: The plot of the optimal routing for a random 20-city problem (see above for order).

(d) Solve the problem with 40 cities. Show the output and plot your solution.

NOTE: since we "... are welcome to use the equivalent functions in Python" I used the PuLP library from Python to do the solving. See code at bottom of PDF.

```
The solver took 0:01:46.691009 to run for 40 cities.
(time is in hours : minutes : seconds.frac_of_second)
Optimal
```

Figure 18: How long it took Python formulation to solve a random 40-city problem.

```
Path travelled
--------------
  0  to  12
  12  to  27
  27  to  26
  26  to  34
  34  to  8
  8  to  37
  37  to  23
  23  to  31
  31  to  7
  7  to  15
  15  to  6
  6  to  36
  36  to  17
  17  to  14
  14  to  2
  2  to  9
  9  to  16
  16  to  10
  10  to  35
  35  to  19
  19  to  25
  25  to  33
  33  to  32
  32  to  39
  39  to  29
  29  to  11
  11  to  4
  4  to  22
  22  to  21
  21  to  5
  5  to  18
  18  to  24
  24  to  28
  28  to  38
  38  to  13
  13  to  20
  20  to  1
  1  to  30
  30  to  3
  3  to  0
Total distance traveled is: 215.7922 units
```

Figure 19: The order of the optimal tour.

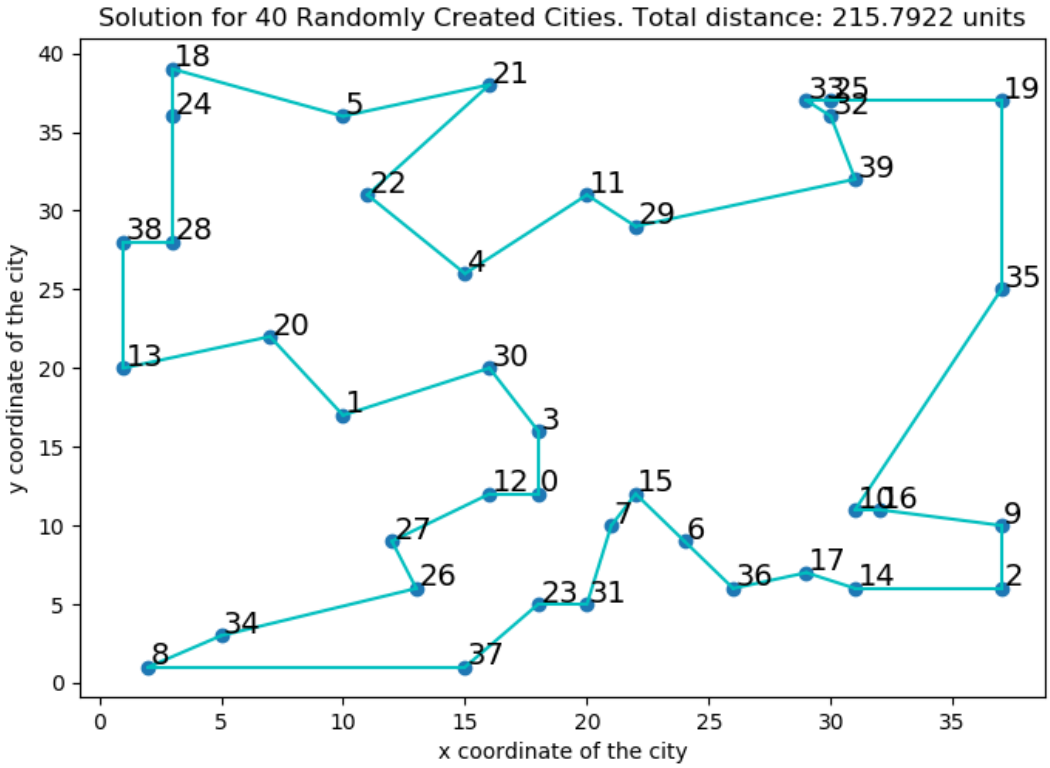

Figure 20: The plot of the optimal routing for a random 40-city problem (see above for order).

(e) Solve the problem with 60 cities. Show the output and plot your solution.

NOTE: since we "... are welcome to use the equivalent functions in Python" I used the Gurobi interface with Python to do the solving (the PuLP package couldn't handle it). See code at bottom of PDF.

```
Root relaxation: objective 6.321593e+02, 97 iterations, 0.00 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0  632.15925    0    6          -  632.15925      -     -    0s
     0     0  650.68299    0    8          -  650.68299      -     -    0s
     0     0  654.43738    0   20          -  654.43738      -     -    0s
     0     0  656.69793    0   20          -  656.69793      -     -    0s
     0     0  666.94670    0    8          -  666.94670      -     -    0s
H    0     0                       677.3868176  666.94670  1.54%     -    0s
H    0     0                       673.7521420  666.94670  1.01%     -    0s
     0     0  667.39655    0   12  673.75214  667.39655  0.94%     -    0s
     0     0  670.27325    0   24  673.75214  670.27325  0.52%     -    0s
     0     0  670.47099    0   22  673.75214  670.47099  0.49%     -    0s
     0     0     cutoff    0       673.75214  673.75214  0.00%     -    0s

Cutting planes:
  Gomory: 4
  MIR: 2
  Zero half: 6

Explored 1 nodes (241 simplex iterations) in 0.33 seconds
Thread count was 8 (of 8 available processors)

Solution count 2: 673.752 677.387

Optimal solution found (tolerance 1.00e-04)
Best objective 6.737521419883e+02, best bound 6.737521419883e+02, gap 0.0000%
The solver took 0:00:00.341254 to run for 60 cities.
(time is in hours : minutes : seconds.frac_of_second)
```

Figure 21: How long it took Gurobi-Python formulation to solve a random 60-city problem.

```
Optimal tour: [0, 31, 22, 33, 51, 11, 21, 55, 8, 18, 56, 16, 28, 15, 2, 25, 12, 42, 24, 14,
41, 30, 1, 6, 13, 19, 37, 26, 32, 5, 48, 43, 27, 35, 50, 17, 39, 58, 57, 47, 10, 46, 3, 54,
20, 34, 44, 59, 9, 29, 36, 23, 4, 40, 53, 38, 45, 49, 7, 52, 0]
```

Figure 22: The order of the optimal tour.

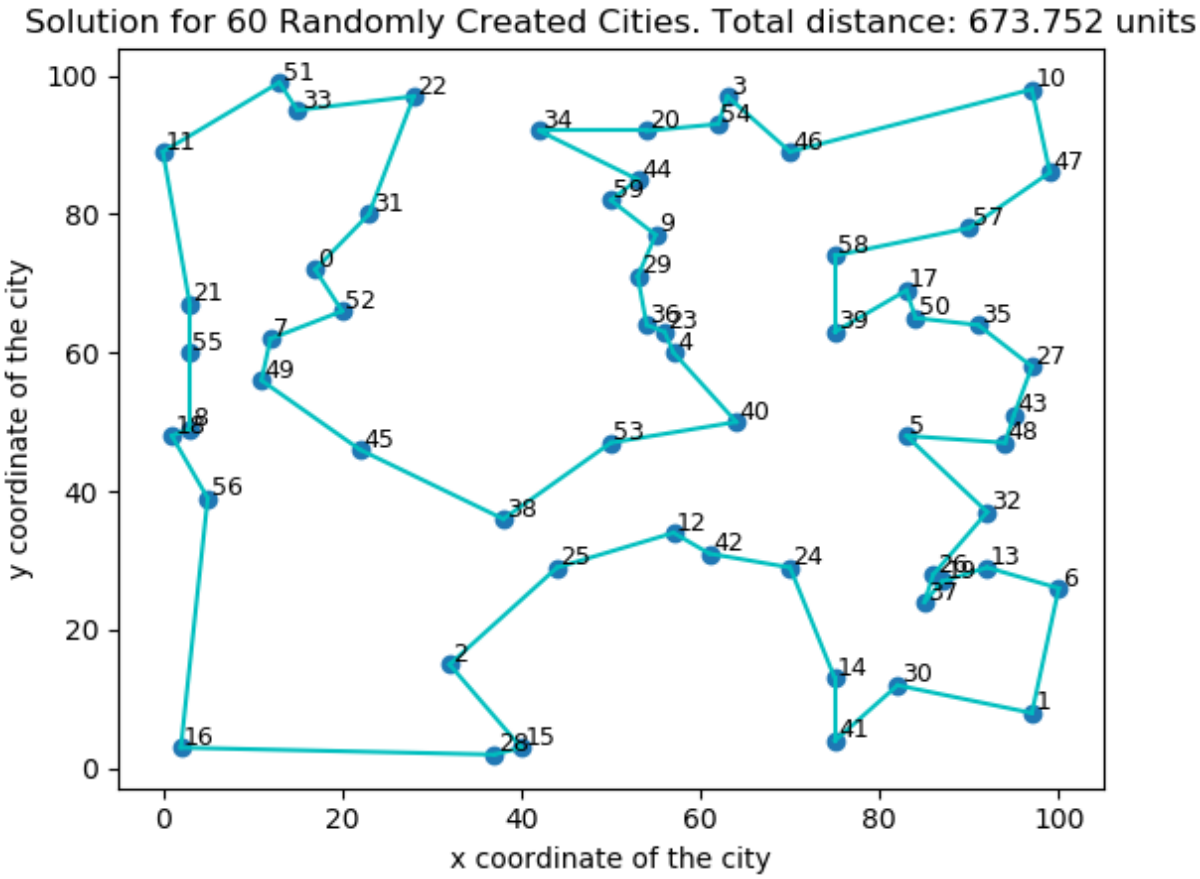

Figure 23: The plot of the optimal routing for a random 60-city problem (see above for order).

(f) If your computer did okay with 60 cities, keep increasing the number of cities until it doesn't.

***** The best that I could do was 400 cities below are the results. See code at bottom of PDF.**

```
The solver took 0:07:57.108607 to run for 400 cities.
(time is in hours : minutes : seconds.frac_of_second)
```

Figure 24: How long it took Gurobi-Python formulation to solve a random 400-city problem. (My laptop was starting to heat up, so I had to stop here even though it only took 7 minutes.)

```
Optimal tour: [0, 90, 73, 268, 52, 88, 63, 358, 150, 298, 125, 282, 191, 141, 373, 114, 31,
87, 399, 311, 68, 368, 212, 307, 22, 203, 172, 340, 218, 318, 379, 164, 34, 386, 71, 356,
20, 389, 302, 44, 127, 59, 9, 149, 359, 325, 69, 309, 245, 83, 314, 301, 260, 385, 54, 3,
258, 316, 181, 188, 178, 367, 377, 272, 336, 329, 46, 148, 199, 271, 123, 303, 104, 343,
129, 361, 246, 58, 190, 194, 352, 144, 39, 204, 155, 288, 396, 5, 332, 121, 280, 228, 306,
50, 17, 237, 394, 57, 333, 310, 277, 321, 380, 95, 330, 10, 393, 143, 47, 226, 84, 342, 74,
227, 166, 72, 113, 35, 265, 120, 179, 322, 27, 43, 48, 255, 270, 131, 32, 106, 229, 238,
215, 354, 103, 266, 219, 128, 370, 387, 300, 213, 80, 390, 220, 371, 40, 79, 124, 165, 153,
256, 107, 4, 23, 36, 65, 275, 283, 281, 29, 137, 295, 163, 210, 77, 383, 291, 66, 173, 223,
242, 158, 319, 211, 391, 305, 105, 384, 348, 85, 67, 346, 160, 110, 221, 109, 334, 53, 81,
308, 392, 285, 175, 152, 193, 99, 135, 38, 366, 294, 362, 278, 96, 355, 177, 243, 25, 397,
374, 328, 345, 12, 42, 174, 61, 350, 142, 206, 369, 224, 236, 395, 335, 102, 365, 231, 171,
122, 24, 64, 78, 89, 98, 37, 273, 19, 26, 207, 13, 6, 136, 262, 185, 159, 279, 197, 1, 252,
168, 269, 70, 126, 92, 344, 162, 381, 30, 339, 297, 323, 14, 251, 41, 86, 289, 357, 398,
287, 284, 94, 115, 130, 189, 201, 353, 117, 347, 296, 326, 320, 315, 241, 232, 364, 138, 2,
97, 196, 375, 264, 176, 244, 259, 240, 15, 28, 198, 156, 91, 167, 312, 248, 116, 276, 257,
205, 293, 169, 341, 134, 222, 108, 93, 216, 360, 16, 200, 376, 388, 299, 202, 100, 214,
147, 132, 267, 157, 170, 60, 261, 363, 230, 225, 304, 161, 111, 286, 101, 327, 112, 247,
338, 192, 250, 234, 133, 56, 254, 154, 18, 8, 145, 183, 184, 182, 186, 263, 274, 317, 146,
45, 233, 239, 249, 75, 119, 351, 209, 49, 195, 7, 382, 76, 55, 331, 337, 21, 82, 378, 139,
140, 253, 11, 290, 118, 62, 217, 313, 235, 51, 372, 33, 208, 151, 349, 187, 324, 292, 180,
0]
```

Figure 25: The order of the optimal tour.

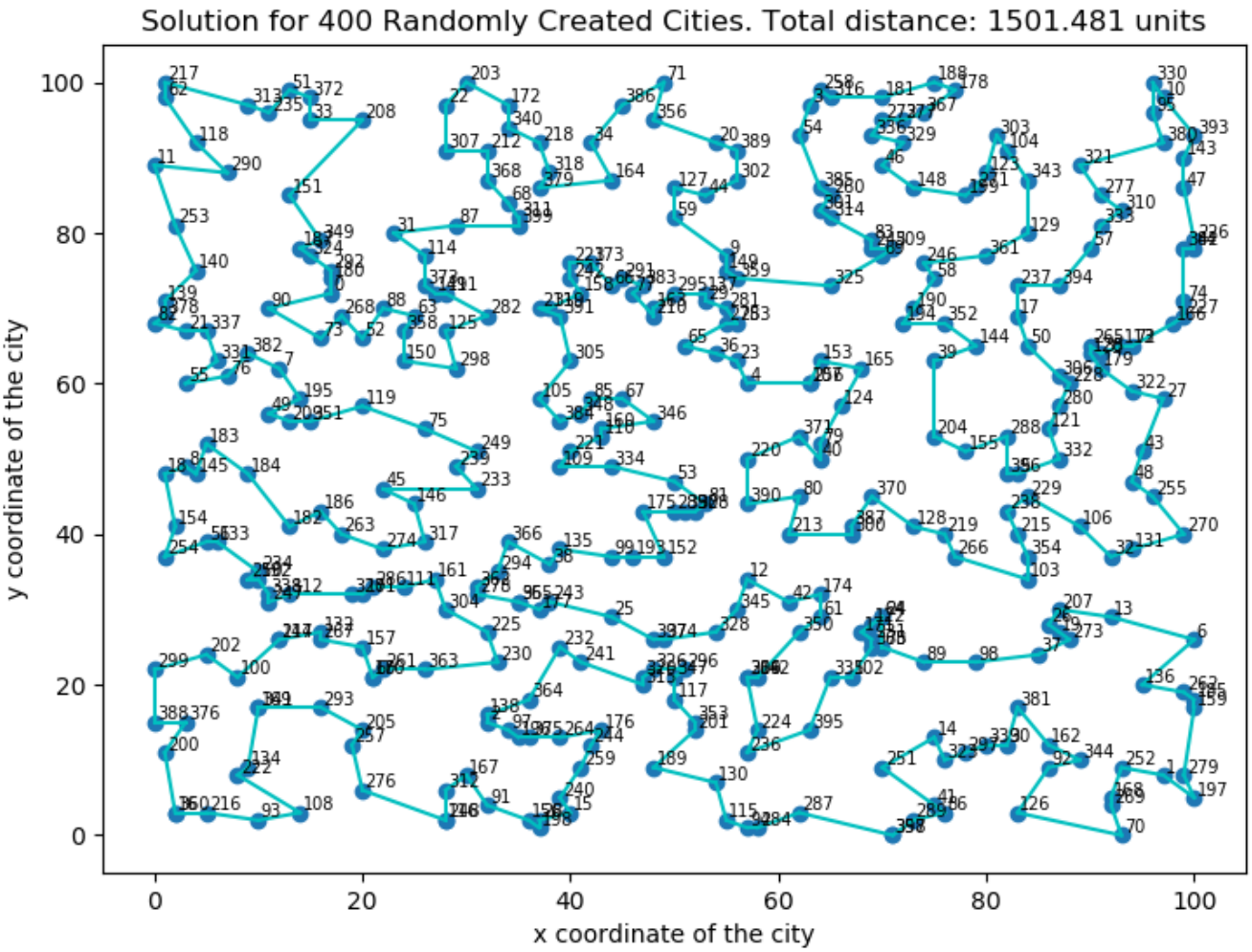

Figure 26: The plot of the optimal routing for a random 400-city problem (see above for order).

```
'''
File name      : hw5_prob2_TSP.py
Author         : Jared Hansen
Date created   : 10/22/2019
Python version : 3.7.3

DESCRIPTION: The purpose of this script is to solve the TSP (Traveling
             Salesman Problem) using integer programming. I have elected to use
             the pulp package in Python to do so (at least for the 6, 20, and
             40 city problems. It couldn't handle the 60 city problem.)

'''




#=============================================================================
#=============================================================================
#==== IMPORT STATEMENTS
#=============================================================================
#=============================================================================
import copy
import datetime as dt
#from gurobipy import *
import math
import matplotlib.pyplot as plt
import numpy as np
import random
import pylab as pl
from matplotlib import collections as mc
from pulp import *
from datetime import datetime




#=============================================================================
#=============================================================================
#==== FUNCTION IMPLEMENTATIONS
#=============================================================================
#=============================================================================

def distance(pt_i, pt_j):
    ''' Function for calculating Euclidean distance between two points.
    '''
    dx2 = (pt_i[0] - pt_j[0])*(pt_i[0] - pt_j[0])
    dy2 = (pt_i[1] - pt_j[1])*(pt_i[1] - pt_j[1])
    return(math.sqrt(dx2 + dy2))

def subtour_remove(tsp_prob):
    ''' This function encodes the logic from Pataki's paper, equations 2.3 for
    removing sub-tours from our route.

    Parameters
    ----------
    tsp : pulp.pulp.LpProblem
        Our definition of the TSP problem as an LP using the PuLP library.

    Returns
    -------
    n/a : simply adds more constraints (eliminates subtours) to the tsp LP.
    '''
    # Use two nested loops to go over all pairs of cities. We will end up
    # ignoring tuples (city_i,city_j) where i == j.
    for i in cities:
```

```
        for j in cities:
            if((i != j) and                 # can't be same city
               ((i != '0') and (j != '0')) and  # can't be the origin city
               ((i,j) in bnry)):            # city combo that has 1 (connected)
                tsp_prob += (order[i] - order[j] <= n*(1-bnry[(i,j)]) - 1)

def city_visited(ind):
    ''' This function takes the index of a city visited, adds it to the
    "visited" list and removes it from the "remaining" list.
    '''
    visited.append(remaining.pop(remaining.index(ind)))




#=============================================================================
#=============================================================================
#==== PROCEDURAL CODE
#=============================================================================
#=============================================================================

# How many cities are we dealing with?
n = 20
# Create labeled cities
cities = []
for i in range(n):
    # Append chars '0', '1', '2', ..., 'n' to the cities list
    cities.append(str(i))
# Set seed for script, determine the coordinates for each city
random.seed(1776)
cities_x = np.random.randint(low=1, high=n, size=n)
cities_y = np.random.randint(low=1, high=n, size=n)


'''
#-----------------------------------------------------------------------------
# This section is for doing part B where we are given the coordinates of
# the cities. UNCOMMENT TO RUN FOR PART B.
#-----------------------------------------------------------------------------
cities_x = np.array([0,1,2,10,11,12])
cities_y = np.array([0,1,0.1,-0.1,1,0])
n = len(cities_x)
# Create labeled cities
cities = []
for i in range(n):
    # Append chars '0', '1', '2', ..., 'n' to the cities list
    cities.append(str(i))
#-----------------------------------------------------------------------------
'''

# Define a dictionary of distance between each pair of points
pt_distances = {}
for i in range(n):
    for j in range(n):
        # Only store distances for cities that aren't the same (e.g. don't
        # store distance between city0 and city0 or city1 and city1, etc.)
        if(i != j):
            # The key for the dict is the tuple of city_i and city_j indices.
            # The values in the dict are the distance btwn city_i and city_j.
            pt_distances[(cities[i],cities[j])] = distance([cities_x[i], cities_y[j]],
                                                           [cities_x[j], cities_y[i]])

# Plot the points that we've generated, with cities labeled by number
fig, tsp_img = plt.subplots()
tsp_img.scatter(cities_x, cities_y)
for i, cityNum in enumerate(cities):
```

```
    tsp_img.annotate(cityNum, xy=(cities_x[i], cities_y[i]), xytext=(1,1),
                     textcoords='offset points',
                     fontsize=14)

# Define the problem, similar to using Problem-Based approach in Matlab
tsp_prob = LpProblem("TSP", LpMinimize)
# Define binary variables denoting if city_j is visited after city_i in tour
bnry = LpVariable.dicts('bnry', pt_distances, 0, 1, LpBinary)
# Define the objective function to be minimized.
# Our objective is to minimize the total distance traveled in order to visit
# each city exactly once, and end up back at the city in which we started.
obj = lpSum([bnry[(i,j)]*pt_distances[(i,j)] for (i,j) in pt_distances])
# Add our newly-defined objective function to the problem we've defined above
tsp_prob += obj

# Define our "assignment constraints" as given in the Pataki paper.
# Add them to the tsp_prob as we go.
for c in cities:
    # Every city is arrived at exactly once
    tsp_prob += (lpSum([bnry[(i,c)] for i in cities if (i,c) in bnry]) == 1)
    # Every city is departed from exactly once
    tsp_prob += (lpSum([bnry[(c,i)] for i in cities if (c,i) in bnry]) == 1)

# To get rid of subtours we have to store the order of cities visited.
order = LpVariable.dicts('order', cities, 0, (n-1), LpInteger)
# Call the function for removing subtours
subtour_remove(tsp_prob)

# Start timing the solver
start_time = dt.datetime.now()
# Call the "solve" method on our defined problem to get the solution
tsp_prob.solve()
# Stop timing the solver
timed = str(dt.datetime.now() - start_time)
print("The solver took " + timed + " to run for " + str(n) + " cities.")
print("(time is in hours : minutes : seconds.frac_of_second)")
# It returns a "1" to the console, so we know that it was successful.
# Furthermore, we can call the LpStatus method to see that 1 == "Optimal"
print(LpStatus[tsp_prob.status])

#---------------------------------------------------------------------------
# Determine the order in which we vist the cites
#---------------------------------------------------------------------------
# Make a deep copy of the list "cities" (otherwise it's basically just a
# reference to "cities" so we'd modify the original object).
remaining = copy.deepcopy(cities)
# Create an integer variable to hold the index of the current city
current = '0'
# Initialize a list with 0 since this is the origin city by definition. This
# list will keep track of the order of our visits
visited = []
city_visited(current)
# Loop until we've visited all of the cities (e.g. removed all of them from
# the "remaining" list)
while(len(remaining) > 0):
    for next_city in remaining:
        # If bnry for the tuple (current,next_city) is 1 this means that we
        # visit the next_city from the current one. So we can add current to
        # our visited list, remove it from remaining, and set current to be
        # the next city.
        if(bnry[(current,next_city)].varValue == 1):
            city_visited(next_city)
            current = next_city
            # Once we've found a new visit and modified lists correctly,
            # break out of the for loop.
            break
# Since we go back to the origin city at the end, add this to the visited list
```

```
# now that we've stored the rest of the tour
visited.append('0')

# Store the distances between each city that we visited
visited_dists = [pt_distances[(visited[i-1], visited[i])] for i in
                        range(1,len(visited))]

print('\nPath travelled')
print('--------------')
# Display the order in which the cities were visited
for i in range(len(visited_dists)):
    print(' ', visited[i], ' to ', visited[i+1])
# How far was the total distance traveled?
total_dist = round(sum(visited_dists),4)
print('Total distance traveled is:', total_dist, 'units')

# Plot the solution over top of the points
fig, tsp_img = plt.subplots()
tsp_img.scatter(cities_x, cities_y)
for i, cityNum in enumerate(cities):
    tsp_img.annotate(cityNum, xy=(cities_x[i], cities_y[i]), xytext=(1,1),
                     textcoords='offset points',
                     fontsize=12)
for i in range(len(visited)-1):
    plt.plot([cities_x[cities.index(visited[i])],cities_x[cities.index(visited[i+1])]],
                      [cities_y[cities.index(visited[i])],cities_y[cities.index(visited[i+1])]], 'c')
plt.title('Solution for ' + str(n) + ' Randomly Created Cities. Total distance: ' + str(total_dist) + ' units')
#plt.title('Solution for ' + str(n) + ' part-B Cities. Total distance: ' + str(total_dist) + ' units')
plt.xlabel('x coordinate of the city')
plt.ylabel('y coordinate of the city')
plt.show()
```

```
'''
File name      : gurobi_tsp.py
Author         : Jared Hansen
Date created   : 10/23/2019
Python version : 3.7.3

DESCRIPTION: The purpose of this script is to solve the TSP (Traveling
             Salesman Problem) using integer programming.
             For sake of disclosure, I should note that I began this code by
             using the script provided by Gurobi here
             https://www.gurobi.com/documentation/8.1/examples/tsp_py.html
             I have elected to use the Gurobi interface with Python to do the
             60-city problem since the pulp package was only computationally
             efficient enough to do the 6-, 20-, and 40-city problems.

'''




#=============================================================================
#=============================================================================
#==== IMPORT STATEMENTS
#=============================================================================
#=============================================================================
import datetime as dt
import itertools
import math
import matplotlib.pyplot as plt
import numpy as np
import random
import sys
from datetime import datetime
from gurobipy import *




#=============================================================================
#=============================================================================
#==== FUNCTION IMPLEMENTATIONS
#=============================================================================
#=============================================================================

def subtourelim(model, where):
    ''' This function is used to eliminate sub-tours from a given route.

    Parameters
    ----------
    model : gurobipy.Model
        Stored model formulation (e.g. constraints, objective, dec. vars.)
    where : int

    Returns
    -------
    n/a : updates the gurobipy.Model by removing subtours (adding constraints)
    '''
    if where == GRB.Callback.MIPSOL:
        # make a list of edges selected in the solution
        vals = model.cbGetSolution(model._vars)
        selected = tuplelist((i,j) for i,j in model._vars.keys() if vals[i,j] > 0.5)
        # find the shortest cycle in the selected edge list
        tour = subtour(selected)
        if len(tour) < n:
            # add subtour elimination constraint for every pair of cities in tour
```

```
            model.cbLazy(quicksum(model._vars[i,j]
                                  for i,j in itertools.combinations(tour, 2))
                         <= len(tour)-1)

def subtour(edges):
    ''' This function accepts a tuplelist (data type that is specific to
    GurobiPy) and finds the shortest sub-tour for the given cities.

    Parameters
    ----------
    edges : gurobipy.tuplelist
        Collection of cities (nodes) in the problem.

    Returns
    -------
    cycle : list
        List of the cities (nodes) that results in the shortest subtour.
    '''
    unvisited = list(range(n))
    cycle = range(n+1) # initial length has 1 more city
    while unvisited: # true if list is non-empty
        thiscycle = []
        neighbors = unvisited
        while neighbors:
            current = neighbors[0]
            thiscycle.append(current)
            unvisited.remove(current)
            neighbors = [j for i,j in edges.select(current,'*') if j in unvisited]
        if len(cycle) > len(thiscycle):
            cycle = thiscycle
    return cycle




#===============================================================================
#===============================================================================
#==== PROCEDURAL CODE
#===============================================================================
#===============================================================================

# How many cities are we dealing with?
n = 400
# Create labeled cities
cities = []
for i in range(n):
    # Append chars '0', '1', '2', ..., 'n' to the cities list
    cities.append(str(i))

# Create coordinates for each city
random.seed(1)
points = [(random.randint(0,100),random.randint(0,100)) for i in range(n)]
cities_x = np.array(points)[:,0]
cities_y = np.array(points)[:,1]

# Plot the points that we've generated, with cities labeled by number
fig, tsp_img = plt.subplots()
tsp_img.scatter(cities_x, cities_y)
for i, cityNum in enumerate(cities):
    tsp_img.annotate(cityNum, xy=(cities_x[i], cities_y[i]), xytext=(1,1),
                     textcoords='offset points',
                     fontsize=9)

# Dictionary of Euclidean distance between each pair of points
dist = {(i,j) :
    math.sqrt(sum((points[i][k]-points[j][k])**2 for k in range(2)))
```

```
    for i in range(n) for j in range(i)}
# Instantiate an instance of a model
m = Model()
# Create variables
vars = m.addVars(dist.keys(), obj=dist, vtype=GRB.BINARY, name='e')
for i,j in vars.keys():
    vars[j,i] = vars[i,j] # edge in opposite direction
# Add "assignment constraints" (each city gets arrived at once, departed from
# once).
m.addConstrs(vars.sum(i,'*') == 2 for i in range(n))

# Optimize model, time how long it takes to run
start_time = dt.datetime.now()
m._vars = vars
m.Params.lazyConstraints = 1
m.optimize(subtourelim)   # this eliminates sub-tours from our route
vals = m.getAttr('x', vars)
selected = tuplelist((i,j) for i,j in vals.keys() if vals[i,j] > 0.5)
timed = str(dt.datetime.now() - start_time)
print("The solver took " + timed + " to run for " + str(n) + " cities.")
print("(time is in hours : minutes : seconds.frac_of_second)")

# Determine the order in which we visit the cities
tour = subtour(selected)
assert len(tour) == n
tour.append(0)
# Output the results of the problem
print('')
print('Optimal tour: %s' % str(tour))
print('Optimal cost: %g' % m.objVal)
print('')

# Plot the solution over top of the points
fig, tsp_img = plt.subplots()
tsp_img.scatter(cities_x, cities_y)
for i, cityNum in enumerate(cities):
    tsp_img.annotate(cityNum, xy=(cities_x[i], cities_y[i]), xytext=(1,1),
                     textcoords='offset points',
                     fontsize=7)
for i in range(len(tour)-1):
    plt.plot([cities_x[cities.index(str(tour[i]))],cities_x[cities.index(str(tour[i+1]))]],
                      [cities_y[cities.index(str(tour[i]))],cities_y[cities.index(str(tour[i+1]))]], 'c')
plt.title('Solution for ' + str(n) + ' Randomly Created Cities. Total distance: ' + str(round(m.objVal, 3)) + ' units')
plt.xlabel('x coordinate of the city')
plt.ylabel('y coordinate of the city')
plt.show()
```