Author: Fran Panteli

Coursework For Module: 5CCP211C Introduction to Numerical Modelling

Due Date: 25/01/2021

Lecturers: Dr Cedric Weber & Dr Joe Bhaseen

In [1]:
```
#Importing modules:
import numpy as np
import numpy.random as random

import matplotlib
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
%matplotlib notebook
%matplotlib inline

import time
import json
import copy
from IPython.core.interactiveshell import InteractiveShell
```

5CCP211C Coursework: A Report Using Numerical Modelling Methods To Investigate The Travelling Salesman Problem & Tacoma Bridge Collapse

Abstract:

This report uses numerical modelling methods to investigate the travelling salesman problem (TSP) and 1940 Tacoma Narrows bridge collapse, as examples of optimisation problems which are otherwise seen in sectors such as investing, production and energy. The TSP aims to produce atour of shortest distance through a given number of locations. Numerical methods used to solve this problem for thirty cities on a map of North America include the simulated annealing algorithm and Monte Carlo sampling, finding a path of shortest distance equal to 2233 arbitrary units of distance. Numerical methods used to model the Tacoma Narrows bridge collapse are the McKenna model, Taylor and Cromer methods. This report finds maximal bridge resonance occurs under a wind force amplitude equal to 2N, with wind of an angular velocity equal to 3rad/s - matching its literature value. Extensions to the former problem are discussed in the form of a TSP for London Underground stations and extensions to the latter are discussed in the form of foot-force models for pedestrians driving bridge collapse.

Contents

1. Introduction

1.1 An Introduction to The Importance of Optimisation Problems

Optimisation problems are key to the functioning of modern society in sectors such as finance and investing, production, agriculture and energy. Solutions to these problems are designed to use a resource in the most efficient way possible to accomplish the optimisation goal with regards to constraints. For instance, in investing this is seen in ‘portfolio optimisation.’ This is designed to decide where to invest funds in order to minimise risk and maximise investment return. An example used in production is ‘blending,’ deciding the proportion of different materials used in garments of clothing. This is in order to achieve required qualities while minimising cost. Another example is in agriculture, in which the type and number of crops to produce are decided in a process called ‘crop planning.’ This is to produce the most crop while managing land sustainably. A final example is ‘generator commitment’ used in the energy sector, deciding when to produce energy according to when it will be most profitable [1]. In each case, a decision of how to garner the most productivity from a resource while minimising another constraint is made. The importance of numerical modelling to simulate solutions and outcomes to these problems is key to making these decisions. In this report, both problems we shall explore (the ‘travelling salesman’ and ‘Tacoma bridge’ problems) involve cost reduction in order to maximise profits as such.

1.2 An Introduction to The Travelling Salesman Problem

The first problem we shall explore is the travelling salesman problem, referred to as the 'TSP.' This problem adopts the concept that cities on a map may be represented by nodes, with each having their own set of coordinates. A ‘tour’ is a route between these cities which traverses every node once and returns to the starting node [2]. The problem sees a hypothetical salesperson wishing to obtain the tour of minimum distance between a set of locations such that they may minimise the associated fuel and time costs incurred by travelling a more inefficient tour, which for instance has applications in the delivery of goods via drones. We shall explore this problem applied first to three, then eight, twenty and thirty American cities whose names and coordinates are stored in the variable ‘capitals_list.’ This is by using a form of 'simulated annealing' and Monte Carlo sampling to produce a heuristic ('best estimate') solution of minimum length. After this, other applications of simulated annealing are discussed.

1.3 An Introduction to The Tacoma Bridge Problem

The second example we investigate is the modelling of the historic collapse of the Tacoma Narrows bridge. The bridge was primarily constructed in 1938 as a suspension bridge in order to reduce the number of materials required, such that only two towers and steel ribbons would be used to support the structure. This minimalistic design was estimated to reduce the production cost of the bridge by 3 million dollars while achieving the objective of a functional bridge [7]. However, bridge aerodynamics were not sufficiently considered when it met its demise two years after construction. This was due to two main affects caused by wind known as vortex shedding (causing vertical bridge oscillations) and aeroelastic flutter (causing torsional bridge oscillations). While ‘vortex shedding’ refers to the forming of wind vortices surrounding the bridge due to its steel girder, ‘aeroelastic flutter’ refers to the build in angular momentum of the bridge once it started oscillating torsionally. This collapse lead to modern solutions used in suspension bridges to prevent further bridge collapses such as Stockbridge dampers and using a break in the steel girders to reduce vortex shedding. Here, we use coupled second order differential equations and the Taylor method in order to explore the behaviour of the bridge under different initial conditions of torsion angle and displacement. This is first implemented without and then with wind, and the impact of people walking on the bridge as opposed to wind is discussed as an extension to the problem.

2. Example 1 : The Travelling Salesman

2.1 A Summary of The Approach Used to Solve the Travelling Salesman Problem

A summary of the structure of the approach used for this problem is outlined below:

1. Theory used to solve the problem is discussed

1. Basic parameters and are first established such as the list of capital cities and their coordinates

1. Functions for the problem are defined and explained. The purpose of all defined functions and their arguments for this problem are provided below for reference and ease of use:
   - `coords(path)` - returns a list of the coordinates for the cities in the input path ‘path’
   - `coord(city)` - returns the coordinates for the input city ‘city’
   - `show_path(path)` - returns the input path ‘path’ on a map of North America
   - `temperature(t, alpha, T_o)` - returns the fictitious temperature given the set of initial conditions for time ‘t,’ value of α equal to ‘alpha’ and initial temperature ‘T_o’
   - `random_path_generate(number_of_cities)` – generates a random path with an imput number of cities equal to ‘number_of_cities’

- `pathswap(path)` – returns the input path called ‘path’ with two random cities swapped
- `d(path)` – returns the total distance of the input path called ‘path’
- `anneal(initial_path, T_o, alpha, num_steps, t)` – performs the simulated annealing algorithm and returns a new path with an attempted number of pair-wise exchanges equal to ‘num_steps’ for an initial input path called ‘initial_path,’ initial temperature equal to ‘T_o,’ decay constant ‘alpha,’ and a decay time of ‘t’ seconds
- `behaviour(randp, alpha_time, initial_temparature, final_temparature, temparature_steps, alpha_1, alpha_2, alpha_3, alpha_4)` - function which produces a logarithmic graph of distance obtained against temperature, exploring the behaviour of the algorithm on an input path called ‘randp’ for different initial conditions. These are: initial temperature ‘initial_temparature,’ increasing in increments of ‘temparature_steps’ to a final temperature ‘final_temparature’ for four values of α, ‘alpha_1’ to ‘alpha_4’ with a decay time equal to ‘alpha_time’

1. Initial conditions for ideal decay time, α and initial temperature T_o are established using the function defined as ‘behaviour’ for random paths of length three, eight and thirty cities

1. Results for local minima in the energy landscape are gathered, presented and discussed. This is by running the simulated annealing algorithm eight times but with a significant number of pair-wise exchanges and with the determined initial conditions. While these local minima are discussed a final solution to the TSP is proposed

1. Outlooks and improvements to the implementation of the problem are discussed

2.2 Theory Behind Using Simulated Annealing To Solve the Travelling Salesman Problem

2.2.1 The Need For Pair-Wise Exchange

The need for Monte Carlo sampling in the TSP arises from the rising complexity of the algorithm as more nodes are added. This is such that if every permutation of possible paths were trialled and the best selected, as the number of cities increases to 15, the run time of the algorithm would be 12 hours. This would rise to $1,928$ years should five more cities be added, later evolving into a run time of the order of years which exceeds the number of atoms in the known universe at 61 nodes. Therefore, a random starting tour is generated – from which arcs connecting adjacent cities in the tour are switched should they generate a path of shorter distance. This process is referred to as pair-wise exchange [8].

Should a suggested pair-wise exchange be generated of longer distance than the length of the current tour, an acceptance probability is generated according to a fictitious exponentially decaying temperature profile. This acceptance probability P is compared to a randomly generated probability r, and the suggested pair-wise exchange is accepted or rejected accordingly.

2.2.2 The Purpose of Finely Tuned Initial Conditions

It is notable firstly that both the decay constant α of the temperature profile and the initial temperature T_o are referred to as the ‘initial conditions,’ whose values must be finely initialised should the approach succeed. Therefore, after defining functions for the code, the behaviour of the algorithm for different initial conditions is investigated with the aim of obtaining initial conditions which produce paths of shortest distance. Typically, this involves large values of T_o which are greater than 1000 degrees kelvin, and values of α smaller than but close to one.

The specific temperature profile this report uses to solve the problem is found in equation 2 below, with equation 1 mathematically defining P in terms of this temperature:

$$P_i = \exp\left(\frac{d_{i-1} - d_i}{T}\right) \qquad (1)$$

$$T_i(t_i) = \alpha^{t_i} T_0 \qquad (2)$$

2.2.3 An Energy Landscape

Secondly, a suggested pair-wise exchange of larger distance than the current tour may be accepted should it satisfy $r<P$ at that point despite the aim of the algorithm to produce a tour of shortest distance. This is because associated with each solution is a fictitious temperature and thus a fictitious energy. The ‘energy’ of all possible paths exists in an ‘energy landscape’ consisting of minima and maxima, with the ideal path existing at the global minimum of the landscape. As new paths are adopted, the simulated annealing ‘scans’ the landscape. Should only pair-wise exchanges of shorter distances than the current path be adopted, the annealing becomes stuck in local minima in the landscape. This means that accepting a select number of pair-wise exchanges of larger distance allows the annealing to ‘escape’ local minima, increasing the chance of finding the global minimum and thereby the optimal solution.

In order to obtain the most ideal tour, this simulated annealing is ran multiple times in order to search for local minima in the landscape. This is such that the local minimum of shortest distance and lowest ‘energy’ can be adopted as the heuristic solution. In order to generate a solution as such, we first define the cities and their coordinates we would like to solve the problem for in the next section.

2.3 Establishing Parameters For The Travelling Salesman Problem

The code below loads the list of capital cities and their coordinates stored in the file named "capitals.json":

In [2]:
```
#Define list of capital cities
InteractiveShell.ast_node_interactivity = "all"
map = mpimg.imread("map.png")
with open('capitals.json', 'r') as capitals_file:
    capitals = json.load(capitals_file)
capitals_list = list(capitals.items()) #capitals_list is list of North US capitals & their
 coordinates on the map
capitals_list = [(c[0], tuple(c[1])) for c in capitals_list]
```

Note that a 'path' is defined as a list of cities and their coordinates. For instance, the code below produces the list of cities and the nth city in the list:

In [3]:
```
#Inputs
n = 6

#Call the list
print("The list of capital cities and their coordinates:", capitals_list)

#To print nth city in the list
capitals_list[n]
```

```
The list of capital cities and their coordinates: [('Oklahoma City', (392.8, 356.4)), ('Mont
gomery', (559.6, 404.8)), ('Saint Paul', (451.6, 186.0)), ('Trenton', (698.8, 239.6)), ('Sal
t Lake City', (204.0, 243.2)), ('Columbus', (590.8, 263.2)), ('Austin', (389.2, 448.4)), ('P
hoenix', (179.6, 371.2)), ('Hartford', (719.6, 205.2)), ('Baton Rouge', (489.6, 442.0)), ('S
alem', (80.0, 139.2)), ('Little Rock', (469.2, 367.2)), ('Richmond', (673.2, 293.6)), ('Jack
son', (501.6, 409.6)), ('Des Moines', (447.6, 246.0)), ('Lansing', (563.6, 216.4)), ('Denve
r', (293.6, 274.0)), ('Boise', (159.6, 182.8)), ('Raleigh', (662.0, 328.8)), ('Atlanta', (58
5.6, 376.8)), ('Madison', (500.8, 217.6)), ('Indianapolis', (548.0, 272.8)), ('Nashville',
(546.4, 336.8)), ('Columbia', (632.4, 364.8)), ('Providence', (735.2, 201.2)), ('Boston', (7
38.4, 190.8)), ('Tallahassee', (594.8, 434.8)), ('Sacramento', (68.4, 254.0)), ('Albany', (7
02.0, 193.6)), ('Harrisburg', (670.8, 244.0))]
```

Out[3]:
```
('Austin', (389.2, 448.4))
```

2.4 Defining Functions For The Travelling Salesman Problem

2.4.1 Defining the function 'coords'

The function defined in the cell below removes the names of cities in an in input list, to return an array of the coordinates of the cities in the list. Steps used in the comments of the code defining the function refer to the following:

1. Defines empty iteration lists
2. Extracts tuple containing one city name & coordinate from list of cities in 'path'
3. Extracts x and y city coordinate from tuple
4. Stores x and y coordinates in separate arrays
5. Zips x and y arrays into a combined list of coordinates for cities in 'path' & returns it

In [4]:
```
#Defining the function:
def coords(path):
    #1:
    x_list=[] #for x coordinates
    y_list=[] #for y coordinates
    coords=[] #for (x, y)

    for i in range(len(path)): #iterates throguh path adds coordinates to list
        #2:
        city_tuple=path[i]

        #3:
        x=city_tuple[1][0] #x
        y=city_tuple[1][1] #y

        #4:
        x_list=np.append(x_list, x) #x
        y_list=np.append(y_list, y) #y

        #5:
        coords=zip(x_list, y_list)
    return list(coords)

#Code to test this function:
p = [capitals_list[23],capitals_list[1],capitals_list[19]]
p
coords(p)
```

Out[4]:
```
[('Columbia', (632.4, 364.8)),
 ('Montgomery', (559.6, 404.8)),
 ('Atlanta', (585.6, 376.8))]
```

Out[4]:
```
[(632.4, 364.8), (559.6, 404.8), (585.6, 376.8)]
```

2.4.2 Defining The Function 'coord'

The defined function below takes a city name and its coordinates as its arguement and returns only the coordinates:

In [5]:
```
#Defining the function:
def coord(city): #note the arguement is in the form of capitals_list[n]
    return (city[1][0],city[1][1])

#Code to test this function:
q=capitals_list[23]
q
coord(q)
```

Out[5]: ('Columbia', (632.4, 364.8))

Out[5]: (632.4, 364.8)

2.4.3 Defining The Function 'show_path'

The defined function below returns the input path shown on a map of Noth America provided by the U.S. Geological Survey, National Geospatial Program.

In [6]:
```
#Defining the function:
def show_path(path_, starting_city, w=35, h=15):
    """Plot a TSP path overlaid on a map of the US States & their capitals."""
    path=coords(path_)
    x, y = list(zip(*path))

    _, (x0, y0) = starting_city

    plt.imshow(map)
    plt.plot(x0, y0, 'y*', markersize=15)  # y* = yellow star for starting point
    plt.plot(x + x[:1], y + y[:1])  # include the starting point at the end of path
    plt.axis("off")
    fig = plt.gcf()
    fig.set_size_inches([w, h])

#Code to test the function:
p = [capitals_list[1],capitals_list[2],capitals_list[3]]
show_path(p, p[0])
print("Figure 1: A Test Path For Three Cities")
```

Figure 1: A Test Path For Three Cities

2.4.4 Defining The Function 'temperature'

The temperature profile defined in equation 2 is coded in the cell below, in a function called 'temperature':

In [11]:
```
def temperature(t, alpha, T_o):
    return (alpha**t)*T_o #returns temparature at time t
```

2.4.5 Defining The Function 'random_path_generate'

The code below defines a function which generates a random path between cities traversing each city only once.

The structure in the comments of the code refers to the following:

1. Initialises lists
2. Generates random list of numbers which satisfy the following conditions:
   - 2A: Contains only integers
   - 2B: No integers are repeated
   - 2C: Integers are between 0 and the length of this list of cities
   - 2D: The list contains number_of_cities terms
3. Generates list of random cities
   - Generates one random city
     - 3A: Generates a random city using list from step 2.
     - 3B: Makes tuple from 3A a list
   - Forms a path
     - 3C: Adds city generated in 3B to list of random cities
     - 3D: Makes list of random cities a tuple
     - 3E: Prints result

In [7]:
```
#Code defining the function
def random_path_generate(number_of_cities):
    #1:
    path_start=[]
    random_integer_list =[]

    #2:
    for i in range(len(capitals_list)+1000): #2D
        if len(random_integer_list)<number_of_cities: #2D
            r=random.randint(0,len(capitals_list)-1) #2C
            if r not in random_integer_list: random_integer_list.append(r) #2B, 2A

    #3
    for i in random_integer_list:
        iteration_city_tuple=capitals_list[i] #3A
        iteration_city_list=list(iteration_city_tuple) #3B
        path_start.append(iteration_city_list) #3C
    path_start=tuple(path_start) #3D
    return path_start #3E

#Code to test the function
test=random_path_generate(8)
print(len(test))
print(type(test))
print(test)
```

```
8
<class 'tuple'>
(['Salem', (80.0, 139.2)], ['Baton Rouge', (489.6, 442.0)], ['Boise', (159.6, 182.8)], ['Lit
tle Rock', (469.2, 367.2)], ['Columbia', (632.4, 364.8)], ['Nashville', (546.4, 336.8)], ['M
ontgomery', (559.6, 404.8)], ['Lansing', (563.6, 216.4)])
```

2.4.6 Defining The Function 'pathswap'

The code below defines a function which takes a path as its arguement and returns the same path with one random set of adjacent cities which have switched order. This begins by choosing a random starting city in the path and swapping it with the city left or right of it in the list by way of a random number. In the case that the starting city is the first city in the path and a left swap is implimented, it is swapped with the last city in the list. A similar process is used in the event that last city in the list is chosen as the starting city with a proposed swap with the city to its right.

In [8]:
```
#Defining the function:
def pathswap(path):
    path = list(path)
    n = random.randint(0, len(path)) #index of random starting city
    new_path = path.copy()
    swapping_city_1 = new_path[n]
    left_or_right = np.random.randint(2)
```

```
    ##Move left of random starting city
    if left_or_right==0:
        #Left corner case & swap left
        if n==0:
            swapping_city_2=new_path[len(path)-1]
            new_path[len(path)-1]=swapping_city_1
            new_path[n]=swapping_city_2

        #Not left cornercase & swap left
        else:
            swapping_city_2=new_path[n-1]
            new_path[n-1]=swapping_city_1
            new_path[n]=swapping_city_2


    ##Move right of random starting city
    else:
        #Right corner case & swap right
        if n==len(path)-1:
            swapping_city_2=new_path[0]
            new_path[n]=swapping_city_2
            new_path[0]=swapping_city_1

        #Not right corner case & swap right
        else:
            swapping_city_2=new_path[n+1]
            new_path[n]=swapping_city_2
            new_path[n+1]=swapping_city_1

    return new_path


#Code to test this function:
testing_path=random_path_generate(3)
print(testing_path)
pathswap(testing_path)
```

```
(['Raleigh', (662.0, 328.8)], ['Nashville', (546.4, 336.8)], ['Little Rock', (469.2, 367.
2)])
```

Out[8]:
```
[['Nashville', (546.4, 336.8)],
 ['Raleigh', (662.0, 328.8)],
 ['Little Rock', (469.2, 367.2)]]
```

2.4.7 Defining The Function 'd(path)'

The next funciton we shall define takes a path as its arguement and returns its distance. This iterates through pairs of coordinates in the input path, calculates the distance between them using Pythagoras's theorem and adds this distance to a counter 'd' which is initialised at zero:

In [9]:
```
#Defining the function:
def d(path):
    coords_list=coords(path)
    d=0

    for i in range(len(path)): #iterate through coordinates
        new_coords=coords_list[i]
        old_coords=coords_list[i-1]
        x_d = new_coords[0]- old_coords[0] #x distance
        y_d = new_coords[1]- old_coords[1]#y distance
        d=d+((x_d**2) + (y_d)**2)**0.5

    return d

#Code to test the function:
d_test=random_path_generate(3)
d_test
d(d_test)
```

Out[9]:
```
(['Salt Lake City', (204.0, 243.2)],
 ['Baton Rouge', (489.6, 442.0)],
 ['Little Rock', (469.2, 367.2)])
```

Out[9]:
```
718.2677363510835
```

2.4.8 Defining The Function 'anneal'

The 'anneal' function peforms pair-wise exchange using theory outlined in section 2.1, takes arguements which are detailed in section 2.2 and is defined as follows:

In [12]:
```
#Code defining the function
def anneal(initial_path, T_o = 1000, alpha=0.97, num_steps=200, t=60):

    #Initialise parameters
    times=np.linspace(0, t, num_steps)
    current_path = initial_path
    initial_distance = d(current_path)

    for i in range(num_steps):
        suggested_path = pathswap(current_path) #2 - perform an attempted pair wise exchang
e
```

```
        #distances
        d_old = d(current_path)
        d_new = d(suggested_path)

        #probability
        T = temperature(times[i], alpha, T_o)
        delta_dist = d_old - d_new
        P=np.exp(delta_dist/T)

        #New path is shorter, accept pair-wise exchange
        if d_new < d_old:
            current_path = suggested_path

        #New path is longer
        else:
            r=np.random.uniform()
            if r<P:
                current_path = suggested_path

        final_distance = d(current_path)

    return current_path

#Code to test this function:
print("Before the anneal function is ran:")
randp=random_path_generate(8)
randp
d(randp)
print("After the anneal function is ran:")
anneal(randp, T_o = 1000, alpha=0.97, num_steps=200)
d(anneal(randp, T_o = 1000, alpha=0.97, num_steps=200))
```

```
Before the anneal function is ran:
```

Out[12]:
```
([['Oklahoma City', (392.8, 356.4)],
  ['Tallahassee', (594.8, 434.8)],
  ['Jackson', (501.6, 409.6)],
  ['Austin', (389.2, 448.4)],
  ['Columbus', (590.8, 263.2)],
  ['Sacramento', (68.4, 254.0)],
  ['Richmond', (673.2, 293.6)],
  ['Lansing', (563.6, 216.4)])
```

Out[12]:
```
2189.37157790326
```

```
After the anneal function is ran:
```

Out[12]:
```
[['Oklahoma City', (392.8, 356.4)],
 ['Sacramento', (68.4, 254.0)],
 ['Lansing', (563.6, 216.4)],
 ['Jackson', (501.6, 409.6)],
 ['Columbus', (590.8, 263.2)],
 ['Tallahassee', (594.8, 434.8)],
 ['Richmond', (673.2, 293.6)],
 ['Austin', (389.2, 448.4)]]
```

Out[12]:
```
1798.47017134537
```

2.4.9 Defining The Function 'behaviour'

The code below defines a function called 'behaviour,' in order to gauge the behaviour of the algorithm under different initial conditions for temparature and alpha. When given an input path, this function generates a logarithmic scatter plot of the best distance obtained against temperature and returns the best path from four values of alpha. Data for these values of alpha is plotted on the same scatter plot since it allows generated data to be better compared.

The structure of the code below follows the format of:

1. Initialising quantities (arrays and parameters for iterating)
2. Iterating over arrays to populate them with values
3. Plotting the arrays
4. Printing results and returning the best path obtained

Reiterating section 2.2, some inputs of the function are:

- *randp* - the initial path the annealing is peformed on, which can be created using the random_path_generate function
- *alpha_time* - the ficticious decay time of the temparature profile
- *initial_temparature* - the lower limit of the trialled temparature range
- *final_temparature* - the upper limit of the trialled temparature range
- *temparature_steps* - increments the temperature increases by
- *alpha_i* where $1<i<4$ - the values of the four values of alpha trialled

In [13]:
```
#Defining the function:
def behaviour(randp, alpha_time = 18.140, initial_temparature = 1000,
              final_temparature = 100000000, temparature_steps = 10000000,
              alpha_1 = 0.8, alpha_2 = 0.9, alpha_3 = 0.95, alpha_4 = 0.97,
             axis_font_size = 14, figure_width = 20, figure_height = 20,
```

```
            title = "A Logarithmic Graph of Distance Against Temparature \nFor Different V
alues of Alpha"):

    ###1: Initialising quantities
        #Parameters for iterating:
    initial_time = time.time()
    best_p=randp
    best_dist=d(randp)
    index = 0
    best_temp = initial_temparature

        #Iteration lists:
    x_axis_list_T = np.arange(initial_temparature, final_temparature, temparature_steps) #x
axis
    y_axis_list_1 = np.zeros(len(x_axis_list_T)) #distance for alpha_1
    y_axis_list_2 = np.zeros(len(x_axis_list_T)) #distance for alpha_2
    y_axis_list_3 = np.zeros(len(x_axis_list_T)) #distance for alpha_3
    y_axis_list_4 = np.zeros(len(x_axis_list_T)) #distance for alpha_4



    ###2:Iterating over arrays to populate them with values
    for T in x_axis_list_T:
        #Generates annealed paths for each value of alpha:
        annealed_path_1 = anneal(randp, T_o = T, alpha=alpha_1, num_steps=200, t=alpha_time
)
        annealed_path_2 = anneal(randp, T_o = T, alpha=alpha_2, num_steps=200, t=alpha_time
)
        annealed_path_3 = anneal(randp, T_o = T, alpha=alpha_3, num_steps=200, t=alpha_time
)
        annealed_path_4 = anneal(randp, T_o = T, alpha=alpha_4, num_steps=200, t=alpha_time
)

        #Finds best path:
        d_list = [d(annealed_path_1), d(annealed_path_2), d(annealed_path_3), d(annealed_pa
th_4)]
        if np.min(d_list) < best_dist:
            best_dist = np.min(d_list)
            best_i = list(d_list).index(np.min(d_list))
            best_temp = T
            if best_i == 0:
                best_p = annealed_path_1
                best_alpha = alpha_1
            if best_i == 1:
                best_p = annealed_path_2
                best_alpha = alpha_2
            if best_i == 2:
                best_p = annealed_path_3
                best_alpha = alpha_3
            if best_i == 3:
                best_p = annealed_path_4
                best_alpha = alpha_4

        #Populates arrays for y axis with distances:
        y_axis_list_1[index] = d_list[0]
        y_axis_list_2[index] = d_list[1]
        y_axis_list_3[index] = d_list[2]
        y_axis_list_4[index] = d_list[3]
        index = index + 1


    ###3: Plotting the arrays
        #For alpha_1:
    plt.figure(figsize=(figure_width,figure_height))
    plt.plot(x_axis_list_T, y_axis_list_1, 'co', label="alpha_1 = {0}".format(alpha_1))
    plt.xscale("log")
    plt.yscale("log")

        #For alpha_2:
    plt.plot(x_axis_list_T, y_axis_list_2, 'go', label="alpha_2 = {0}".format(alpha_2))
    plt.xscale("log")
    plt.yscale("log")

        #For alpha_3:
    plt.plot(x_axis_list_T, y_axis_list_3, 'yo', label="alpha_3 = {0}".format(alpha_3))
    plt.xscale("log")
    plt.yscale("log")

        #For alpha_4:
    plt.plot(x_axis_list_T, y_axis_list_4, 'ro', label="alpha_4 = {0}".format(alpha_4))
    plt.xscale("log")
    plt.yscale("log")
    plt.ylabel("Log scale of distance - [arbitrary units]", fontsize=axis_font_size)
    plt.xlabel("Log scale of temperature - [degrees Kelvin]", fontsize=axis_font_size)
    plt.legend(loc="best")
    plt.title(title, fontsize=axis_font_size+2)



    ###4: Printing results and returning best path obtained
    print("- Run time(s):", time.time() - initial_time)
    print("- Initial path is: {0} with distance {1}".format(randp, d(randp)))
    print("- Best path is: {0} with distance {1}".format(best_p, d(best_p)))
    print("- Shortest distance occured at T_o = {0} and alpha = {1}".format(best_temp, best
_alpha))
    return best_p

#Code to test this function:
# test_path = random_path_generate(8)
# behaviour(randp, alpha_time = 18.140, initial_temparature = 1000,
#               final_temparature = 100000000, temparature_steps = 10000000,
#               alpha_1 = 0.8, alpha_2 = 0.9, alpha_3 = 0.95, alpha_4 = 0.97,
```

```
#               axis_font_size = 16, figure_width = 15, figure_height = 13,
#               title = "Test Graph Title For 'behaviour' Function")
```

2.5 Initial Conditions For Simulated Annealing

2.5.1 Initial Conditions For Generating The Best Path Between Eight Cities

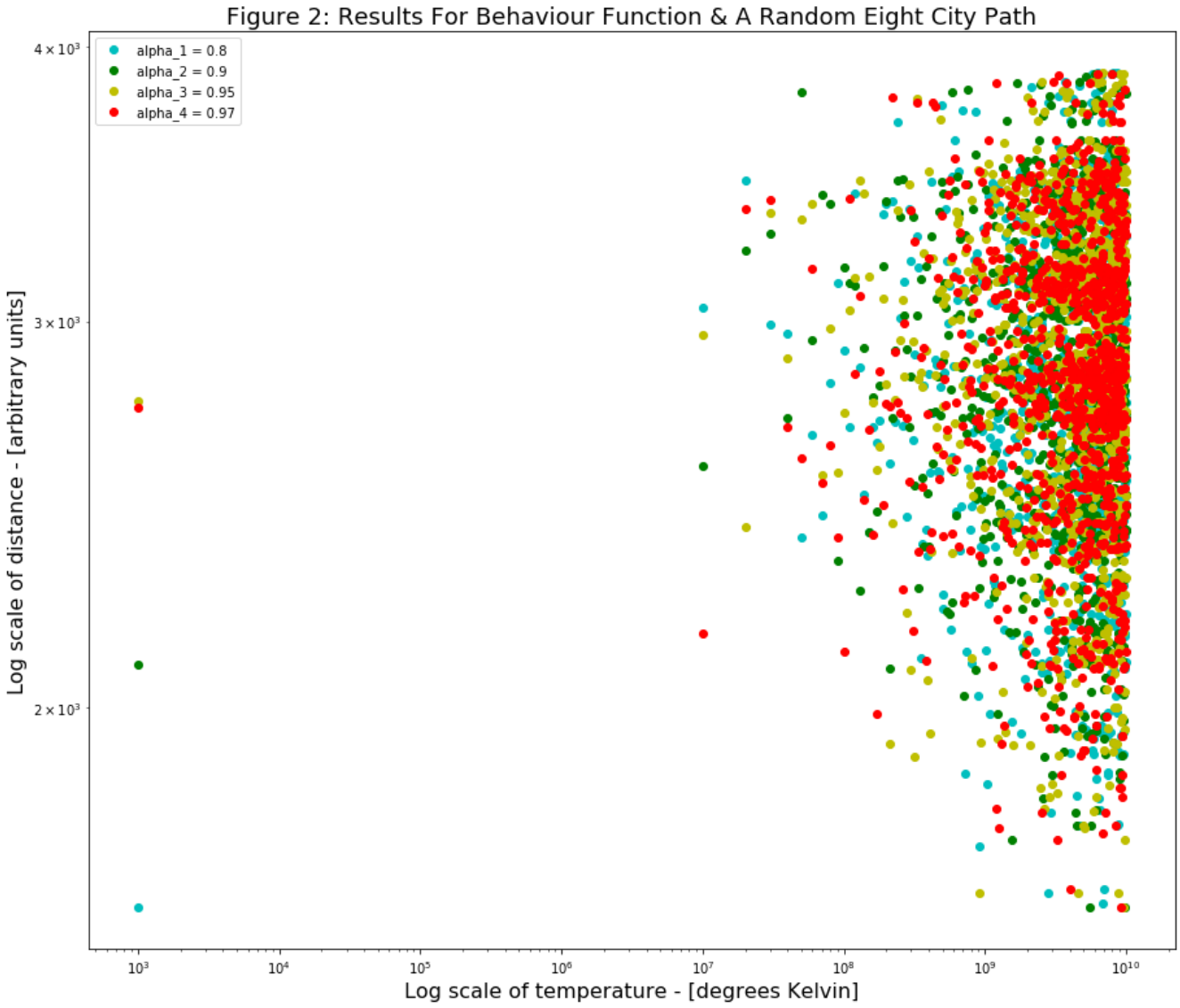
We shall now use the behaviour function in order to explore the behaviour of the algorithm for a path with eight cities, later expanding to paths with twenty and thirty cities. In order to acheive consistent results the same random initial path will be used each time, which is stored in the variable called 'random_eight_city_path' below, and was generated using the random_path_generate function:

In [141]:
```
random_eight_city_path = (['Albany', (702.0, 193.6)],
 ['Austin', (389.2, 448.4)],
 ['Lansing', (563.6, 216.4)],
 ['Salt Lake City', (204.0, 243.2)],
 ['Phoenix', (179.6, 371.2)],
 ['Providence', (735.2, 201.2)],
 ['Salem', (80.0, 139.2)],
 ['Columbia', (632.4, 364.8)])
```

The behaviour function is now ran on this path, which produces the graph below:

In [144]:
```
ideal_eight_city_path = behaviour(random_eight_city_path, alpha_time = 18.140, initial_temp
arature = 1000,
              final_temparature = 10000000000, temparature_steps = 10000000,
              alpha_1 = 0.8, alpha_2 = 0.9, alpha_3 = 0.95, alpha_4 = 0.97,
             axis_font_size = 16, figure_width = 15, figure_height = 13,
             title = "Figure 2: Results For Behaviour Function & A Random Eight City Path")
```

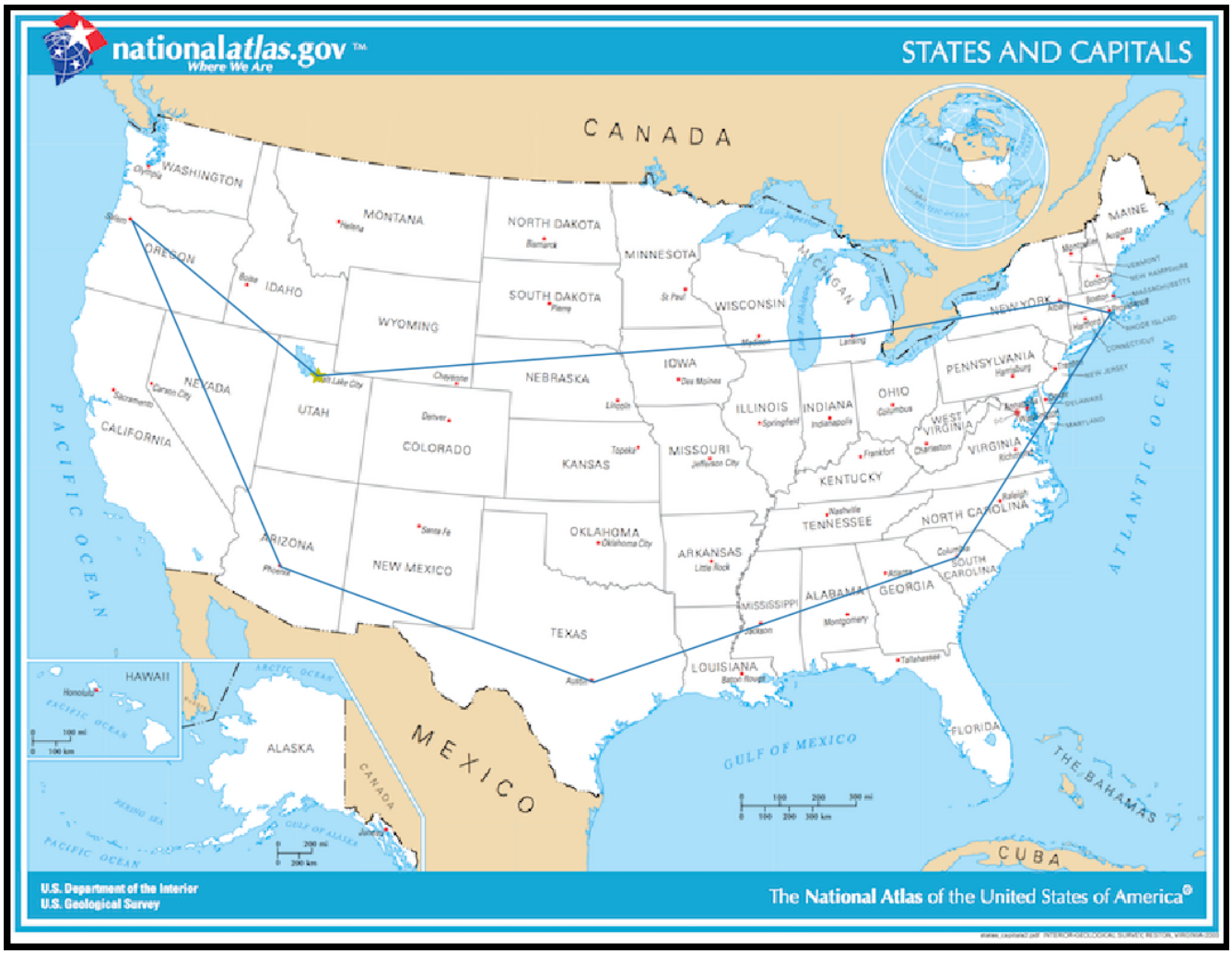
```
- Run time(s): 265.23431491851807
- Initial path is: (['Albany', (702.0, 193.6)], ['Austin', (389.2, 448.4)], ['Lansing', (56
3.6, 216.4)], ['Salt Lake City', (204.0, 243.2)], ['Phoenix', (179.6, 371.2)], ['Providenc
e', (735.2, 201.2)], ['Salem', (80.0, 139.2)], ['Columbia', (632.4, 364.8)]) with distance 3
205.2378916511025
- Best path is: [['Salt Lake City', (204.0, 243.2)], ['Salem', (80.0, 139.2)], ['Phoenix',
(179.6, 371.2)], ['Austin', (389.2, 448.4)], ['Columbia', (632.4, 364.8)], ['Providence', (7
35.2, 201.2)], ['Albany', (702.0, 193.6)], ['Lansing', (563.6, 216.4)]] with distance 1622.9
868004368116
- Shortest distance occured at T o = 1000 and alpha = 0.8
```



When using the show_path function, this produces an ideal path shown in the graph below:

In [147]:
```
show_path(ideal_eight_city_path, ideal_eight_city_path[0])
print("Figure 3: The Path of Shortest Distance Shown In Figure 2")
```

```
Figure 3: The Path of Shortest Distance Shown In Figure 2
```



2.5.2 Initial Conditions For Generating The Best Path Between Twenty Cities

Using a similar process as with eight cities, the code below stores a random path comprised of twenty random cities:

In [148]:
```
random_twenty_city_path = (['Raleigh', (662.0, 328.8)],
 ['Richmond', (673.2, 293.6)],
 ['Boise', (159.6, 182.8)],
 ['Providence', (735.2, 201.2)],
 ['Phoenix', (179.6, 371.2)],
 ['Oklahoma City', (392.8, 356.4)],
 ['Denver', (293.6, 274.0)],
 ['Saint Paul', (451.6, 186.0)],
 ['Hartford', (719.6, 205.2)],
 ['Boston', (738.4, 190.8)],
 ['Sacramento', (68.4, 254.0)],
 ['Jackson', (501.6, 409.6)],
 ['Austin', (389.2, 448.4)],
 ['Columbia', (632.4, 364.8)],
 ['Atlanta', (585.6, 376.8)],
 ['Lansing', (563.6, 216.4)],
 ['Salt Lake City', (204.0, 243.2)],
 ['Indianapolis', (548.0, 272.8)],
 ['Montgomery', (559.6, 404.8)],
 ['Columbus', (590.8, 263.2)])
```

The behaviour function is again used on this path, which produces the graph below:

In [171]:
```
ideal_twenty_city_path = behaviour(random_twenty_city_path, alpha_time = 18.140, initial_te
mparature = 1000000,
              final_temparature = 10000000000000, temparature_steps = 10000000000,
              alpha_1 = 0.8, alpha_2 = 0.9, alpha_3 = 0.95, alpha_4 = 0.97,
             axis_font_size = 16, figure_width = 15, figure_height = 13,
             title = "Figure 4: Results For Behaviour Function & A Random Twenty City Path"
)
```

```
- Run time(s): 607.0399482250214
- Initial path is: (['Raleigh', (662.0, 328.8)], ['Richmond', (673.2, 293.6)], ['Boise', (15
9.6, 182.8)], ['Providence', (735.2, 201.2)], ['Phoenix', (179.6, 371.2)], ['Oklahoma City',
(392.8, 356.4)], ['Denver', (293.6, 274.0)], ['Saint Paul', (451.6, 186.0)], ['Hartford', (7
19.6, 205.2)], ['Boston', (738.4, 190.8)], ['Sacramento', (68.4, 254.0)], ['Jackson', (501.
6, 409.6)], ['Austin', (389.2, 448.4)], ['Columbia', (632.4, 364.8)], ['Atlanta', (585.6, 37
6.8)], ['Lansing', (563.6, 216.4)], ['Salt Lake City', (204.0, 243.2)], ['Indianapolis', (54
8.0, 272.8)], ['Montgomery', (559.6, 404.8)], ['Columbus', (590.8, 263.2)]) with distance 53
34.918346325279
```

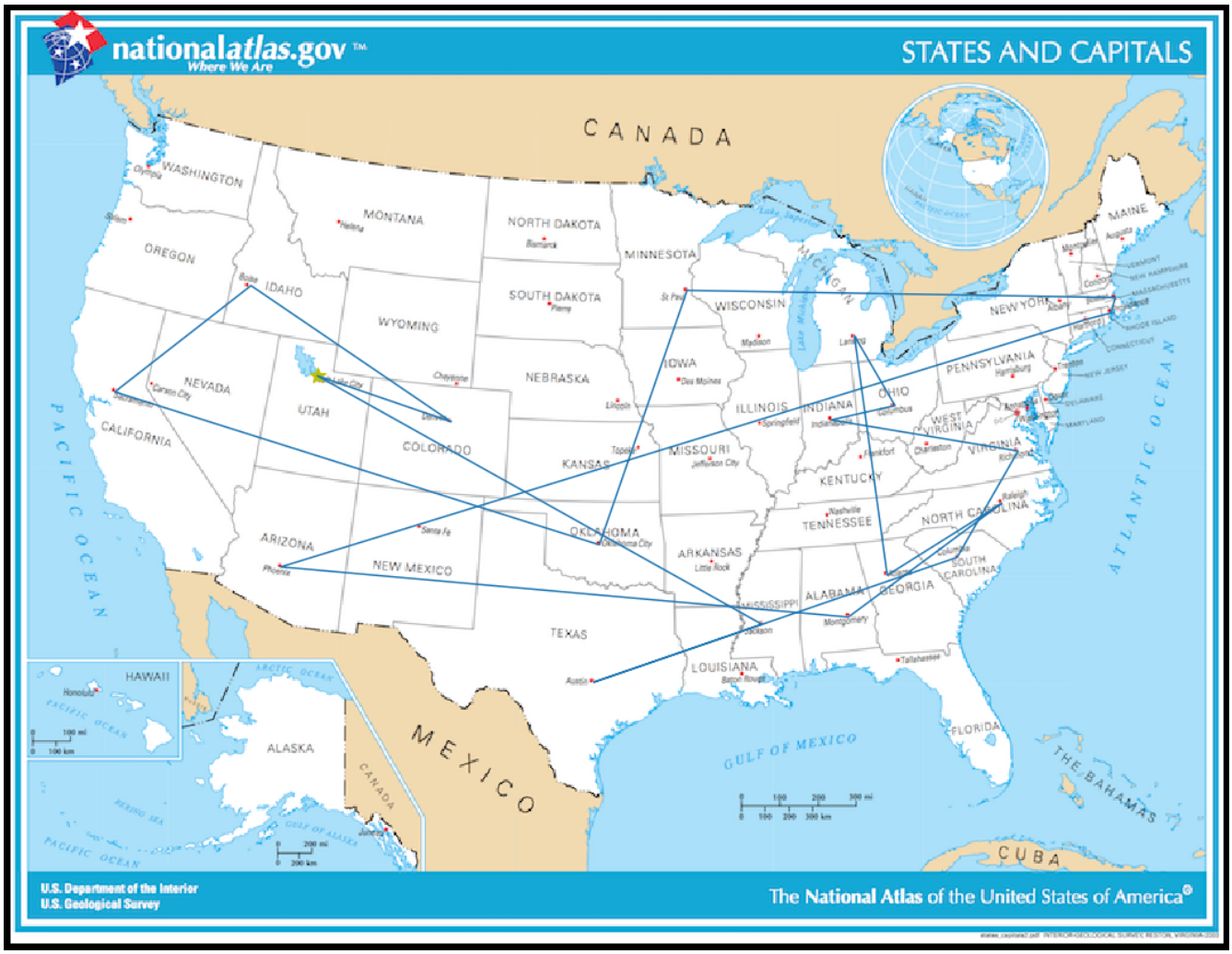
```
- Best path is: [['Raleigh', (662.0, 328.8)], ['Montgomery', (559.6, 404.8)], ['Phoenix', (1
79.6, 371.2)], ['Hartford', (719.6, 205.2)], ['Providence', (735.2, 201.2)], ['Boston', (73
8.4, 190.8)], ['Saint Paul', (451.6, 186.0)], ['Oklahoma City', (392.8, 356.4)], ['Sacrament
o', (68.4, 254.0)], ['Boise', (159.6, 182.8)], ['Denver', (293.6, 274.0)], ['Salt Lake Cit
y', (204.0, 243.2)], ['Jackson', (501.6, 409.6)], ['Austin', (389.2, 448.4)], ['Columbia',
(632.4, 364.8)], ['Richmond', (673.2, 293.6)], ['Indianapolis', (548.0, 272.8)], ['Columbu
s', (590.8, 263.2)], ['Lansing', (563.6, 216.4)], ['Atlanta', (585.6, 376.8)]] with distance
3556.8830670997168
- Shortest distance occured at T o = 120001000000 and alpha = 0.9
```

Plotting this on the map shows the path of shortest distance produced:

In [172]:
```
show_path(ideal_twenty_city_path, ideal_eight_city_path[0])
d(ideal_twenty_city_path)
print("Figure 5: The Path of Shortest Distance Shown In Figure 4")
```

Out[172]:
```
3556.8830670997168
```

```
Figure 5: The Path of Shortest Distance Shown In Figure 4
```



2.5.3 Initial Conditions For Generating The Best Path Between Thirty Cities

The cell below stores a random path containing all thirty cities in a variable called 'random_thirty_city_path':

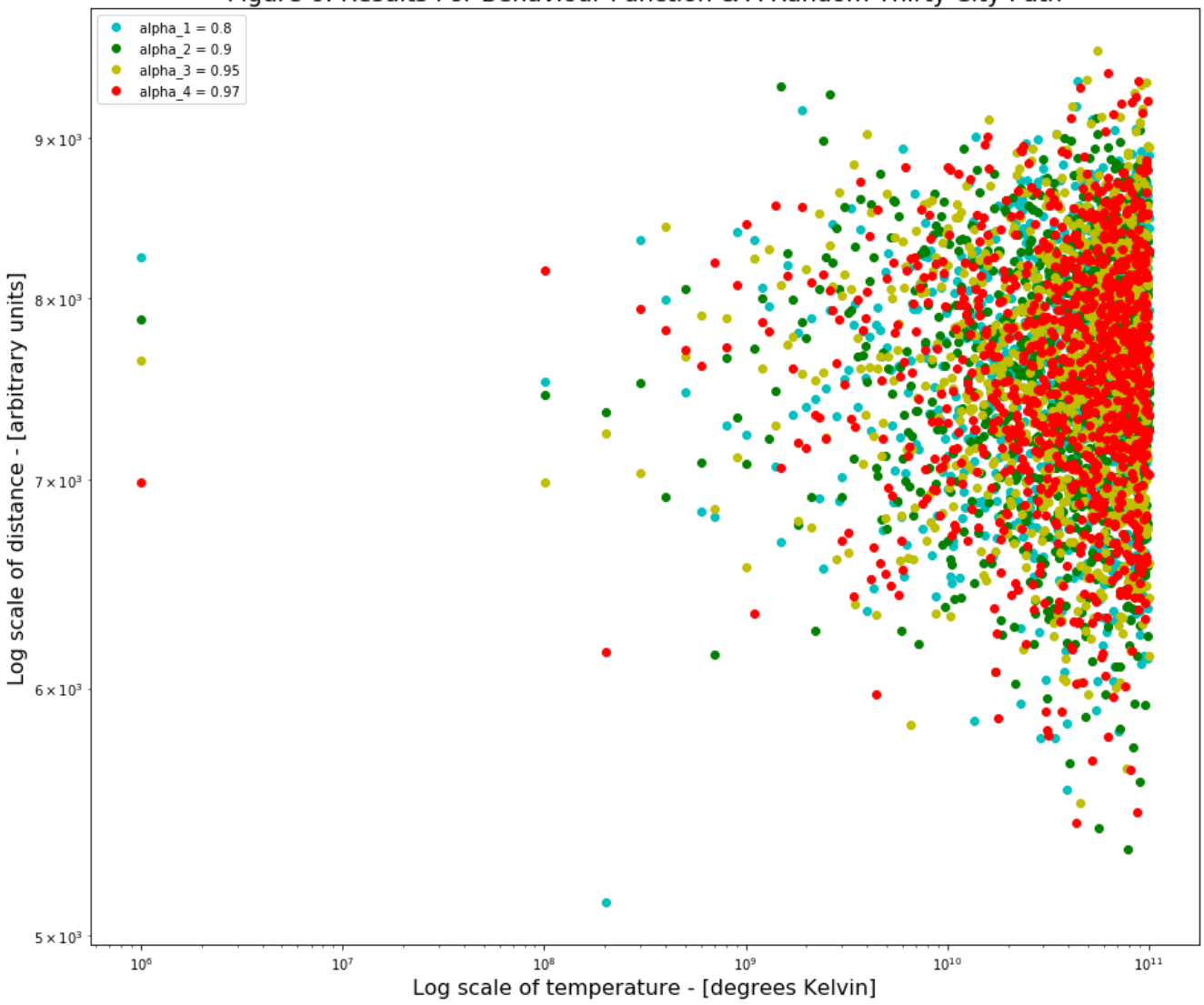
In [152]:
```
random_thirty_city_path = (['Trenton', (698.8, 239.6)],
 ['Lansing', (563.6, 216.4)],
 ['Montgomery', (559.6, 404.8)],
 ['Baton Rouge', (489.6, 442.0)],
 ['Sacramento', (68.4, 254.0)],
 ['Indianapolis', (548.0, 272.8)],
 ['Salem', (80.0, 139.2)],
 ['Tallahassee', (594.8, 434.8)],
 ['Phoenix', (179.6, 371.2)],
 ['Boise', (159.6, 182.8)],
 ['Austin', (389.2, 448.4)],
 ['Columbus', (590.8, 263.2)],
 ['Albany', (702.0, 193.6)],
 ['Columbia', (632.4, 364.8)],
 ['Boston', (738.4, 190.8)],
 ['Des Moines', (447.6, 246.0)],
 ['Atlanta', (585.6, 376.8)],
 ['Salt Lake City', (204.0, 243.2)],
 ['Raleigh', (662.0, 328.8)],
 ['Providence', (735.2, 201.2)],
 ['Denver', (293.6, 274.0)],
 ['Saint Paul', (451.6, 186.0)],
 ['Little Rock', (469.2, 367.2)],
 ['Hartford', (719.6, 205.2)],
 ['Richmond', (673.2, 293.6)],
 ['Nashville', (546.4, 336.8)],
 ['Jackson', (501.6, 409.6)],
 ['Madison', (500.8, 217.6)],
 ['Oklahoma City', (392.8, 356.4)])
```

Running the behaviour function on the path produces the graph below:

In [160]:
```
ideal_thirty_city_path = behaviour(random_thirty_city_path, alpha_time = 18.140, initial_te
mparature = 1000000,
                final_temparature = 100000000000, temparature_steps = 100000000,
                alpha_1 = 0.8, alpha_2 = 0.9, alpha_3 = 0.95, alpha_4 = 0.97,
               axis_font_size = 16, figure_width = 15, figure_height = 13,
               title = "Figure 6: Results For Behaviour Function & A Random Thirty City Path"
)
```

```
- Run time(s): 840.2732951641083
- Initial path is: (['Trenton', (698.8, 239.6)], ['Lansing', (563.6, 216.4)], ['Montgomery',
(559.6, 404.8)], ['Baton Rouge', (489.6, 442.0)], ['Sacramento', (68.4, 254.0)], ['Indianapo
lis', (548.0, 272.8)], ['Salem', (80.0, 139.2)], ['Tallahassee', (594.8, 434.8)], ['Phoeni
x', (179.6, 371.2)], ['Boise', (159.6, 182.8)], ['Austin', (389.2, 448.4)], ['Columbus', (59
0.8, 263.2)], ['Albany', (702.0, 193.6)], ['Columbia', (632.4, 364.8)], ['Boston', (738.4, 1
90.8)], ['Des Moines', (447.6, 246.0)], ['Atlanta', (585.6, 376.8)], ['Salt Lake City', (20
4.0, 243.2)], ['Raleigh', (662.0, 328.8)], ['Providence', (735.2, 201.2)], ['Denver', (293.
6, 274.0)], ['Saint Paul', (451.6, 186.0)], ['Little Rock', (469.2, 367.2)], ['Hartford', (7
19.6, 205.2)], ['Richmond', (673.2, 293.6)], ['Nashville', (546.4, 336.8)], ['Jackson', (50
1.6, 409.6)], ['Madison', (500.8, 217.6)], ['Oklahoma City', (392.8, 356.4)]) with distance
7807.371674075221
- Best path is: [['Oklahoma City', (392.8, 356.4)], ['Baton Rouge', (489.6, 442.0)], ['Littl
e Rock', (469.2, 367.2)], ['Tallahassee', (594.8, 434.8)], ['Indianapolis', (548.0, 272.8)],
['Montgomery', (559.6, 404.8)], ['Austin', (389.2, 448.4)], ['Sacramento', (68.4, 254.0)],
['Phoenix', (179.6, 371.2)], ['Salem', (80.0, 139.2)], ['Boise', (159.6, 182.8)], ['Salt Lak
e City', (204.0, 243.2)], ['Columbia', (632.4, 364.8)], ['Albany', (702.0, 193.6)], ['Bosto
n', (738.4, 190.8)], ['Des Moines', (447.6, 246.0)], ['Columbus', (590.8, 263.2)], ['Hartfor
d', (719.6, 205.2)], ['Atlanta', (585.6, 376.8)], ['Raleigh', (662.0, 328.8)], ['Providenc
e', (735.2, 201.2)], ['Denver', (293.6, 274.0)], ['Saint Paul', (451.6, 186.0)], ['Nashvill
e', (546.4, 336.8)], ['Jackson', (501.6, 409.6)], ['Richmond', (673.2, 293.6)], ['Trenton',
(698.8, 239.6)], ['Madison', (500.8, 217.6)], ['Lansing', (563.6, 216.4)]] with distance 512
6.225576385508
- Shortest distance occured at T o = 201000000 and alpha = 0.8
```

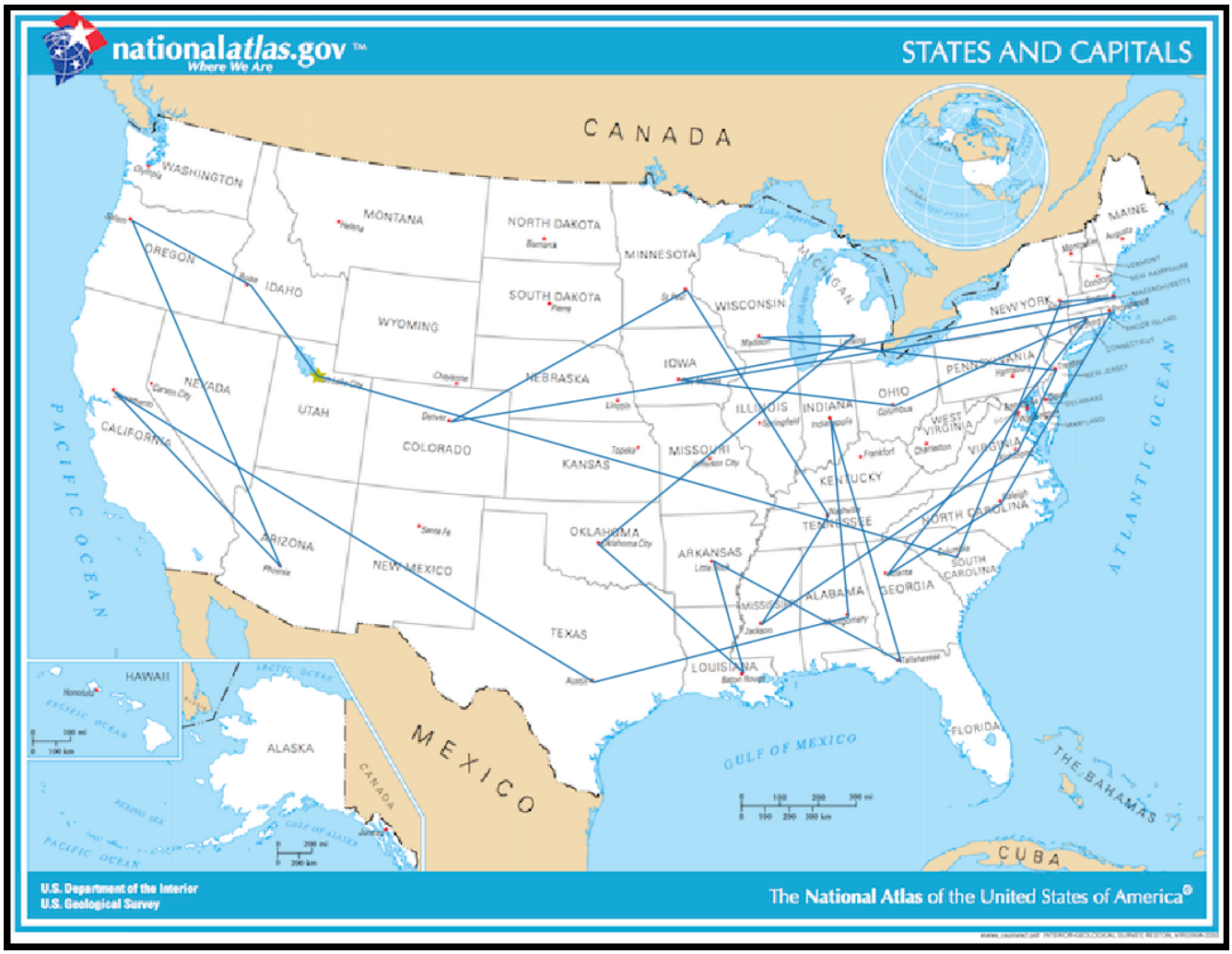
Figure 6: Results For Behaviour Function & A Random Thirty City Path



The path of shortest distance this produces is given on the map below:

In [168]:
```
show_path(ideal_thirty_city_path, ideal_eight_city_path[0])
print("Figure 7: The Path of Shortest Distance Shown In Figure 6")
```

```
Figure 7: The Path of Shortest Distance Shown In Figure 6
```



2.5.4 Trialling Initial Conditions For Decay Time

Since the temparature profile uses time as an arguement, either the time.time() module can be used or a simulated decay time. This report uses a simulated decay time since this provides more control over initial conditions. In order to find the ideal decay time, the code below plots a graph of distance obtained against decay time.

In [181]:
```
#Inputs
trial_count=10 #number of times annealing is trialled
random_path = random_thirty_city_path#number of cities tour traverses
temp = 1000
t_min = 1
t_max = 40
t_steps = 1
al = 0.82


#Run time
start_time = time.time()

#Create arrays
best_distances_t = np.zeros(trial_count)
t_list = np.arange(t_min, t_max, t_steps)
t_y_axis = []

#Iterate over arrays
for t in t_list:
    for i in range(trial_count): #trial count is starting from 0
        annealed_path_t = anneal(random_path, T_o = temp, alpha=al, num_steps=200, t=t)
        best_distances_t[i] = d(annealed_path_t)

#Extract best path
        best_distance_t = np.min(best_distances_t)
    t_y_axis = np.append(t_y_axis, best_distance_t)

#Return values
print("Run time(s):", time.time() - start_time)
plt.figure(figsize=(15,6))
plt.plot(t_list, t_y_axis, 'bo')
plt.ylabel("Path Distance - [arbitrary units of distance]")
plt.xlabel("Simulated decay time - [seconds]")
plt.title("Figure 8: Graph of Distance Obtained Against Simulated Decay Time for Thirty Cit
ies", fontsize=18)
plt.show()
```

```
Run time(s): 82.5514018535614
```

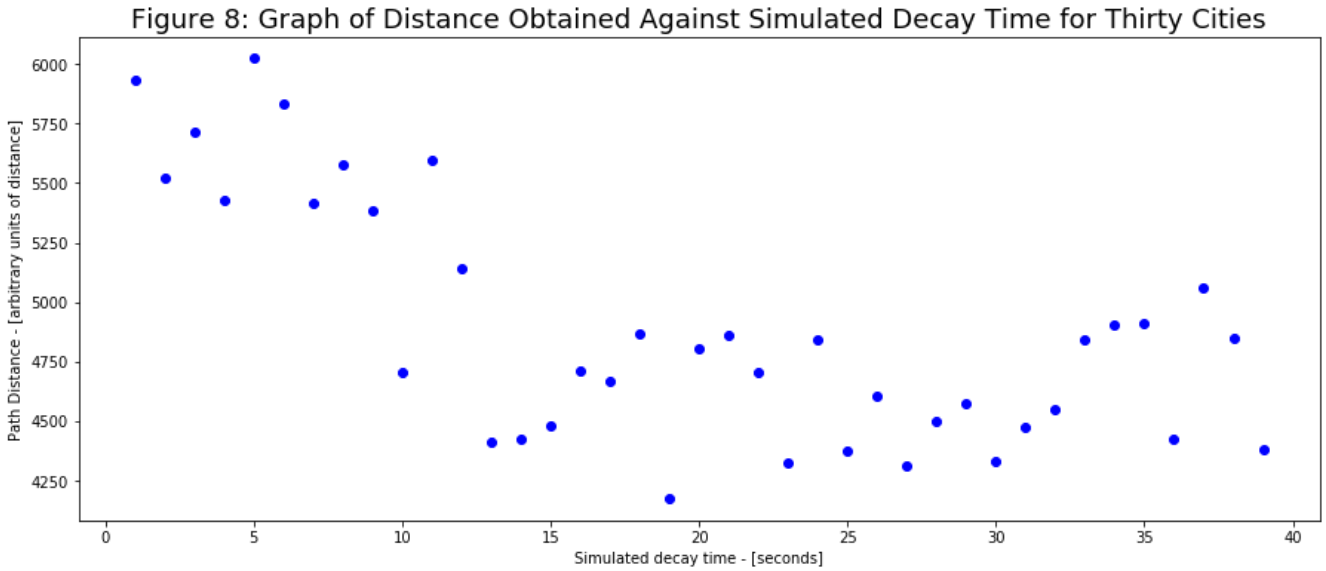
Out[181]: <Figure size 1080x432 with 0 Axes>

Out[181]: [<matplotlib.lines.Line2D at 0x7fc3fe6f38d0>]

Out[181]: Text(0, 0.5, 'Path Distance - [arbitrary units of distance]')

Out[181]: Text(0.5, 0, 'Simulated decay time - [seconds]')

Out[181]: Text(0.5, 1.0, 'Figure 8: Graph of Distance Obtained Against Simulated Decay Time for Thirty Cities')



2.5.5 Obtained Initial Conditions

Through trialling initial conditions further, it has been determined that when searching for local minima initial conditions that are to be used are: a simulated decay time of 18.140s, α equal to 0.812289 and initial temperature of 1000 degrees kelvin.

2.6 Local Minima & Searching

We now repeat the simulated anealing multiple times under the determined initial conditions in the cell below. The aim of this is to generate many local minima in the simulated annealing energy landscape, selecting the path of smallest distance as the global minimum. After this, the cell plots the path found of shortest distance using the show_path function.

The structure of the code in the cell below is as follows:

1. Defining input values for:
   - trial_count - the number of times the annealing is simulated
   - initial_path - the initial path the annealing is trialled on
   - num_steps - the number swaps proposed in each annealing
   - simulated_time - the simulated decay time of the temparature profile
   - alpha - the value of α used in the temparature profile
   - start_temperature - the value of T_o used in the temparature profile

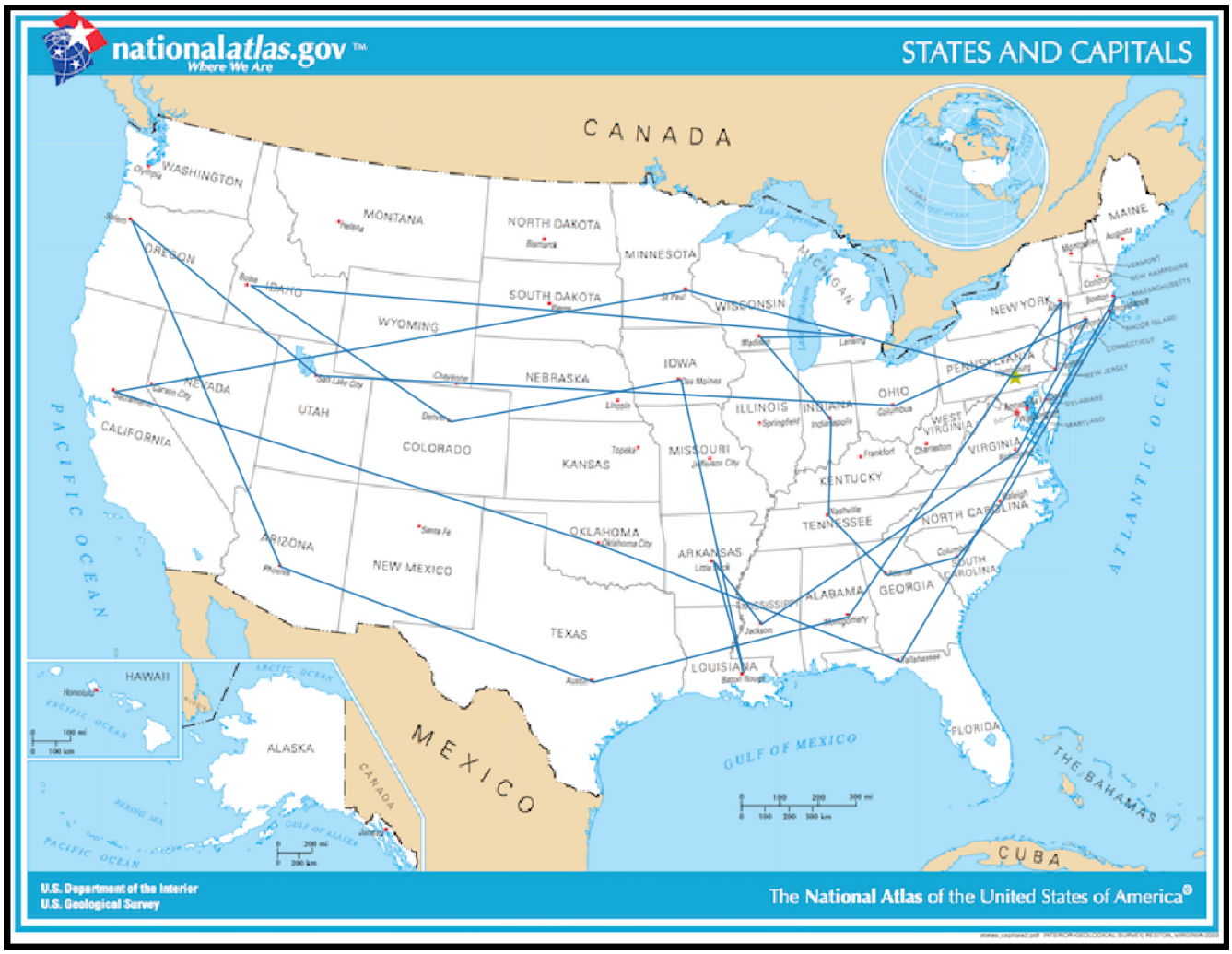
1. Initialising the run time of the code
2. Producing arrays to store values
3. Iterateing over lists to store values
4. Extracting the path with the shortest distance and the value of the shortest distance path
5. Ploting the path produced with the shortest distance and printing results

The initial conditions entered into the cell below are the suggested initial conditions for 'trial_count' and 'num_steps.' However, it was found after experimentation that although using larger values of num_steps and trial_count produced longer run times, the paths produced were of shorter distance. The next section explores obtained results using these larger values of num_steps and trial_count - with less paths, but with each path having a significantly shorter distance.

In [170]:
```
##1: Input values
trial_count = 20 #number of times annealing is trialled
initial_path = capitals_list #or "randp=random_path_generate(x)" if require a random path w
ith x cities
num_steps = 200 #the number of switches proposed in each annealing
simulated_time = 18.140
```

```
alpha = 0.812289
start_temperature = 1000

#2: Run time
initial_time = time.time()

##3: Make arrays to store values
best_distances = np.zeros(trial_count)
best_paths = list(np.empty([trial_count, len(initial_path)-1]))


##4: Iterate over lists to store values
for i in range(trial_count): #trial count is starting from 0
    annealed_path = anneal(initial_path, T_o = start_temperature, alpha=alpha, num_steps=nu
m_steps, t=simulated_time)
    best_distances[i] = d(annealed_path) #updated list of smallest distances
    best_paths[i] = annealed_path #adds the annealed path to the list

##5: Extract path with shortest distance & value of the shortest distance path
best_distance = np.min(best_distances) #best distance
index_of_best_path = list(best_distances).index(best_distance)
best_path = best_paths[index_of_best_path] #extract best path
overall_solution = best_path
overall_solution

##6: Plot best_path, print results
show_path(best_path, best_path[0], w=35, h=15)
print("Run time(s):", time.time() - initial_time) #print run time
print("Distance of the shortest path obtained is", best_distance)
print("Figure 9: An Example of a Path Produced By 'Local Minima & Searching' Code")
```

Out[170]:
```
[('Harrisburg', (670.8, 244.0)),
 ('Trenton', (698.8, 239.6)),
 ('Albany', (702.0, 193.6)),
 ('Montgomery', (559.6, 404.8)),
 ('Austin', (389.2, 448.4)),
 ('Phoenix', (179.6, 371.2)),
 ('Salem', (80.0, 139.2)),
 ('Salt Lake City', (204.0, 243.2)),
 ('Columbus', (590.8, 263.2)),
 ('Hartford', (719.6, 205.2)),
 ('Richmond', (673.2, 293.6)),
 ('Jackson', (501.6, 409.6)),
 ('Little Rock', (469.2, 367.2)),
 ('Baton Rouge', (489.6, 442.0)),
 ('Des Moines', (447.6, 246.0)),
 ('Denver', (293.6, 274.0)),
 ('Boise', (159.6, 182.8)),
 ('Lansing', (563.6, 216.4)),
 ('Madison', (500.8, 217.6)),
 ('Indianapolis', (548.0, 272.8)),
 ('Nashville', (546.4, 336.8)),
 ('Atlanta', (585.6, 376.8)),
 ('Columbia', (632.4, 364.8)),
 ('Raleigh', (662.0, 328.8)),
 ('Providence', (735.2, 201.2)),
 ('Boston', (738.4, 190.8)),
 ('Tallahassee', (594.8, 434.8)),
 ('Oklahoma City', (392.8, 356.4)),
 ('Sacramento', (68.4, 254.0)),
 ('Saint Paul', (451.6, 186.0))]

Run time(s): 4.271213054656982
Distance of the shortest path obtained is 4997.922540581289
Figure 9: An Example of a Path Produced By 'Local Minima & Searching' Code
```



2.7 Travelling Salesperson Problem Results

In order to find the global minimum in the landscape of solutions, the 'local minima and searching' code in the cell above was ran eight times, proposing different solutions to the problem. This was such that the path of shortest distance would be adopted as the final proposed solution to the problem.

The code was ran eight times with the following input parameters:

- trial_count = 100
- initial_path = capitals_list
- num_steps = 100,000
- simulated_time = 18.140
- alpha = 0.812289
- start_temperature = 1000

A summary of the path names, run times and distances of the eight paths this gave is shown in Table 1 below. Here, distance is rounded to the nearest whole number and the run time is rounded to three significant figures:

Path Name	Run Time / Hours	Distance / Units
Path_1	7.51	2367
Path_2	7.52	2383
Path_3	7.49	2361
Path_4	7.43	2495
Path_5	3.42	2514
Path_6	3.38	2318
Path_7	3.41	2233
Path_8	3.42	2513

Table 1 : A table of results for the travelling salesman problem on thirty cities

For simplicity, the cell below only stores the path obtained of shortest distance (Path 7). However, paths 1-8 summarised in Table 1 are stored in the appendices (section 5) for reference purposes.

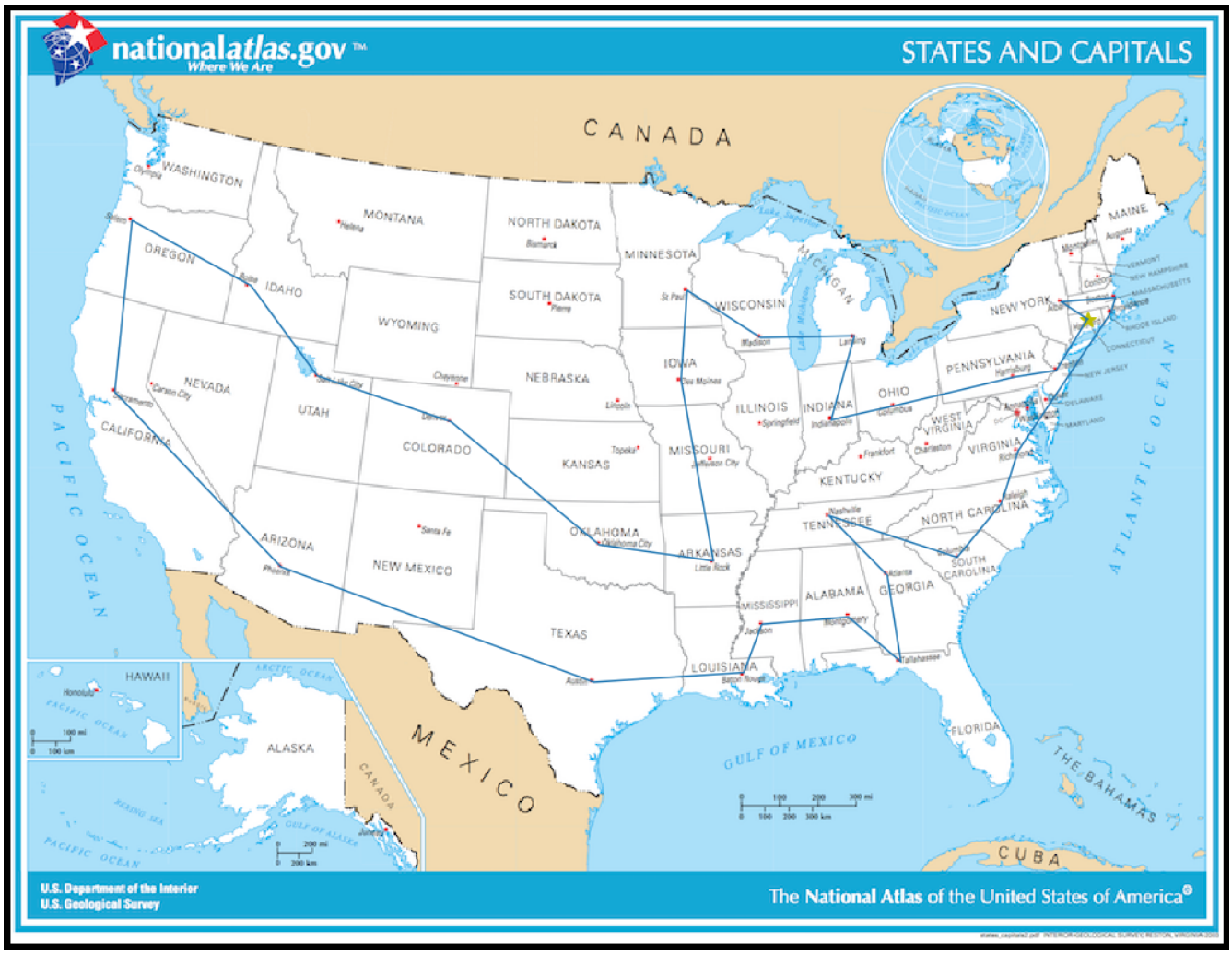
In [24]:
```
Final_path = Path_7
Path_7 = [('Hartford', (719.6, 205.2)),('Albany', (702.0, 193.6)),('Boston', (738.4, 190.8
)),
          ('Providence', (735.2, 201.2)),('Richmond', (673.2, 293.6)),('Raleigh', (662.0, 3
28.8)),
          ('Columbia', (632.4, 364.8)),('Nashville', (546.4, 336.8)),('Atlanta', (585.6, 37
6.8)),
          ('Tallahassee', (594.8, 434.8)),('Montgomery', (559.6, 404.8)),('Jackson', (501.6
, 409.6)),
          ('Baton Rouge', (489.6, 442.0)),('Austin', (389.2, 448.4)),('Phoenix', (179.6, 37
1.2)),
          ('Sacramento', (68.4, 254.0)),('Salem', (80.0, 139.2)),('Boise', (159.6, 182.8)),
          ('Salt Lake City', (204.0, 243.2)),('Denver', (293.6, 274.0)),('Oklahoma City',(3
92.8, 356.4)),
          ('Little Rock', (469.2, 367.2)),('Des Moines', (447.6, 246.0)),('Saint Paul', (45
1.6, 186.0)),
          ('Madison', (500.8, 217.6)),('Lansing', (563.6, 216.4)),('Indianapolis', (548.0,
272.8)),
          ('Columbus', (590.8, 263.2)),('Harrisburg', (670.8, 244.0)),('Trenton', (698.8, 2
39.6))]
```

2.8 A Proposed Final Solution To The Travelling Salesperson Problem

Path_7 has the shortest distance of all obtained estimates and is therefore the final proposed solution to this travelling salesperson problem. Information regarding this path is summarised in the cell below, including a visualisation of the path and its distance:

In [23]:
```
show_path(Final_path, Final_path[0])
print("Therefore, the optimal solution obtained is Path_7 with a distance of {0}.".format(r
ound(d(Final_path))))
#The exact distance of this path is given by d(Final_path)
print("\nThis path is given by the following list:\n\n", Final_path)
print("Figure 10: A Proposed Final Solution To The Travelling Salesman Problem For 30 Citie
s")
```

```
Therefore, the optimal solution obtained is Path_7 with a distance of 2233.0.

This path is given by the following list:

 [('Hartford', (719.6, 205.2)), ('Albany', (702.0, 193.6)), ('Boston', (738.4, 190.8)), ('Pr
ovidence', (735.2, 201.2)), ('Richmond', (673.2, 293.6)), ('Raleigh', (662.0, 328.8)), ('Col
umbia', (632.4, 364.8)), ('Nashville', (546.4, 336.8)), ('Atlanta', (585.6, 376.8)), ('Talla
hassee', (594.8, 434.8)), ('Montgomery', (559.6, 404.8)), ('Jackson', (501.6, 409.6)), ('Bat
on Rouge', (489.6, 442.0)), ('Austin', (389.2, 448.4)), ('Phoenix', (179.6, 371.2)), ('Sacra
mento', (68.4, 254.0)), ('Salem', (80.0, 139.2)), ('Boise', (159.6, 182.8)), ('Salt Lake Cit
y', (204.0, 243.2)), ('Denver', (293.6, 274.0)), ('Oklahoma City', (392.8, 356.4)), ('Little
Rock', (469.2, 367.2)), ('Des Moines', (447.6, 246.0)), ('Saint Paul', (451.6, 186.0)), ('Ma
dison', (500.8, 217.6)), ('Lansing', (563.6, 216.4)), ('Indianapolis', (548.0, 272.8)), ('Co
lumbus', (590.8, 263.2)), ('Harrisburg', (670.8, 244.0)), ('Trenton', (698.8, 239.6))]
Figure 10: A Proposed Final Solution To The Travelling Salesman Problem For 30 Cities
```



2.9 A Discussion of Results For The Travelling Salesperson Problem

2.9.1 Run Times

It is notable that the results had large run times, with a mean time of 5.45 hours per path generated using the input parameters. While it was suggested to find the best path out of twenty generated options, it was found that running the algorithm less times but with a larger number of time steps (of the order of $100,000$) produced less paths in a given time frame, but with each path having a significantly shorter distance and larger run time. Since one criticism of the annealing is its reliance on initial conditions, one solution for shorter run times is to more carefully 'fine tune' the initial conditions for T_o and α.

Since one criticism of the algorithm is its dependance on initial conditions being finely determined, one solution would be an algorithm which generates ideal initial conditions for annealing. This would be ran prior to searching for local minima, therefore reducing the run time of the algorithm. An alternative method could also be used, such as obtaining a minimum spanning tree from a table of least distances and finding a lower bound [2]. Another more efficient algorithm could also be used such as Dijikstra or Prim's algorithm.

Table 1 also shows the inonsistency of run times given that each path was trialled with the same initial conditions. While paths 1 - 4 have a mean run time of 7.49 hours, paths 5 - 8 have a mean run time of 3.41 hours. This is because it was found that in order to run more calculations in a given time frame, coppies of the code could be pasted into different jupyter notebooks and ran at the same time. With the access to processing power, it was tested that the maxiumum amount of coppies of the code that could be ran simultaneously with these initial conditions was 4. This meant that paths 1 - 4 and 5 - 8 would be simultaneously generated on two separate occasions. In the former case, other tasks were being carried out on the computer while the simulations ran and therefore produced longer run times. While in the latter case, there were no other tasks being carried out on the computer while paths were generated, resulting in shorter run times. One solution to this inonsistancy in run times would be to have computers which are used solely for processing results. Another solution is to use computers with more processing power in order to process more results at the same time.

2.9.2 A Heuristic Soluition

While improvements to the algorithm involve decreases in run time and more accurate solutions, it is evident that paths obtained for the travelilng salesperson are heuristic. This means that the solutions obtained are best estimates, although they may be ideal [3]. In some cases, it is also notable that there may be more than one ideal solution. This brings into question the validity of the proposed final solution, since it is only a 'best estimate' and cannot be determined as ideal or not. In

the next section, we discuss how the anneal function would be modified to generate a tour of shortest length for stations on the London Underground as an extension to the problem.

2.10 An Extension Discussing the Modification of the 'Anneal' Function to Solve The Tube Challenge Problem

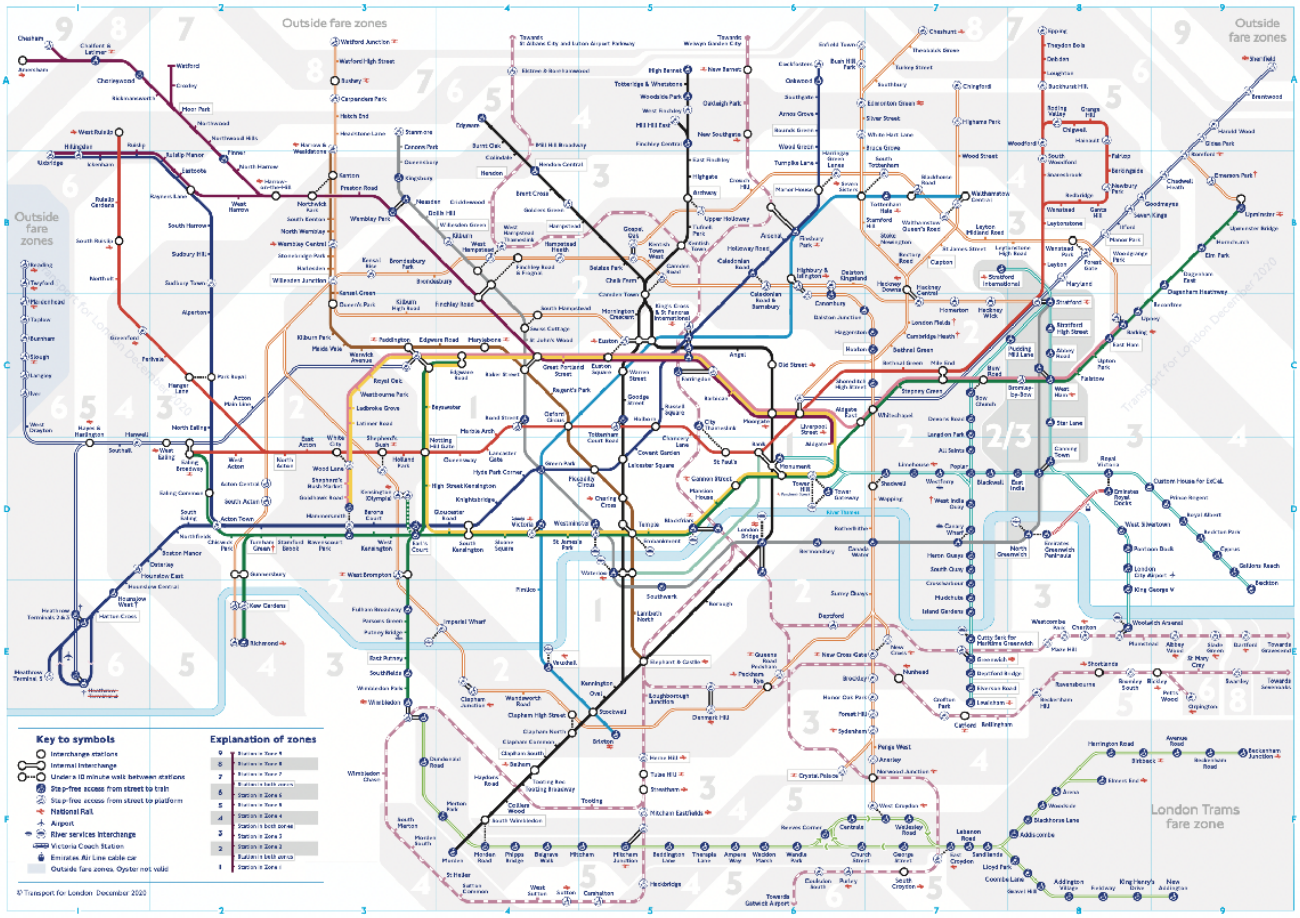
2.10.1 An Introduction To The Tube Challenge Problem

Tourism in London was estimated to generate £15.73 billion for the British economy in the 2019 annum [4] and London Underground's famed network of 270 tube stations shown on the map in Figure 11 [5]. This begs the question of what the shortest tour a tourist wishing to visit all 270 underground stations would be. A similar problem in literature surrounding the topic has been hailed the 'tube challenge' problem and has physically been attempted by teams for the Guinness Book of World Records since 1959 [6]. In order to use the anneal function to solve this problem, multiple simplifications must be made to the rules competitors use for the tube challenge problem, which we discuss in this section as an extension to the TSP.

In [7]:
```
plt.figure(figsize=(35,15))
plt.imshow(plt.imread('Tube Map.png'))
plt.title("Figure 11: A Map of The 270 London Underground Stations",fontsize=27.5)
plt.axis('off')
```

Out[7]:
```
(-0.5, 2245.5, 1589.5, -0.5)
```

Figure 11: A Map of The 270 London Underground Stations



2.10.2 Modifying Rules For The Tube Challenge Problem For Use With Simulated Annealing

The first rule to the tube challenge which must be modified in order to use the anneal function to solve this problem is the prohibition of competitors to travel overground between close tube stations by foot or bus. This means that the tube must be used for all journeys made. Furthermore, overground stations are to be excluded from the problem [6].

Secondly, in the tube challenge teams aim to produce and carry out tours through all tube stations in the minimum time possible, with the 2013 record held at 16 hours and 20 minutes. However, in our simulated annealing problem the aim is to produce the tour of minimum distance rather than time. Weightings for solutions to the tube challenge are generated by using the 'TFL journey planner,' to generate the time taken to travel between stations. The stochastic nature of this method and the variation of journey times, for instance due to the unpredictable nature of delays and inconsistent speeds of tubes will lead us to use distance as weightings rather than time.

In order to generate a map with coordinates of tube stations, a geographic map would be used rather than the map shown in Figure 11. This is because of the distorted relative distances between the stations the tube map shows. The show_path function used for the American cities would then be modified for use on this geographical map for the tube stations. Coordinates for the tube stations would then be generated from this map, and distances be verified or modified to be consistent with those published by TFL in a 2017 Freedom of Information Act [10].

Another difference between this scenario and our exploration of the shortest tour for the American cities is that in the latter scenario, one city could be connected to any other city. However, in this problem one station may only ever connect to its nearest neighbours. Although the nearest neighbour algorithm would be suitable here, it is often used to generate upper bounds for solutions to problems as such [2], therefore implying that annealing would be a more ideal approach. This problem also involves approximately ten times the number of nodes. However, this nearest neighbour constraint would reduce the number of permutations that would have to be trailed and so the algorithm run time. In literature surrounding the topic it is suggested to group these connected nodes into groups or 'clusters' in what is known as the 'Generalised Travelling Salesman Problem,' or GTSP [11].

2.10.3 Validating Solutions & The Future of The Tube Challenge Problem

Since record holders for the tube challenge do not publish their ideal solutions, the solution obtained to the problem for annealing could be compared with those from other algorithms such as Dijkstra or Prim's algorithms [2] in order to judge its effectiveness.

Further advances and improvements to the algorithms in this field will aid in reducing competitor's times for the tube challenge in future. Extensions to this problem also exist in the possibility of using the New York subway system and other underground networks, for instance. It is also inevitable that new stations will continue to be added to London's network.

3. Example 2 : The Tacoma Bridge

3.1 The Structure of This Section

This section shall first discuss the theory regarding differential equations needed to model the Tacoma bridge collapse. Following this, the theory is coded into a function called 'tacoma_with_wind,' which generates three arrays for: the vertical displacement, torsion angle of the bridge and time. The behaviour of this algorithm using different initial conditions is discussed by plotting these arrays. Note that this function can generate these arrays for conditions with and without wind, by using a wind force amplitude A equal or not equal to zero. Ideas for extending the example and 'foot-force models' in literature are then discussed.

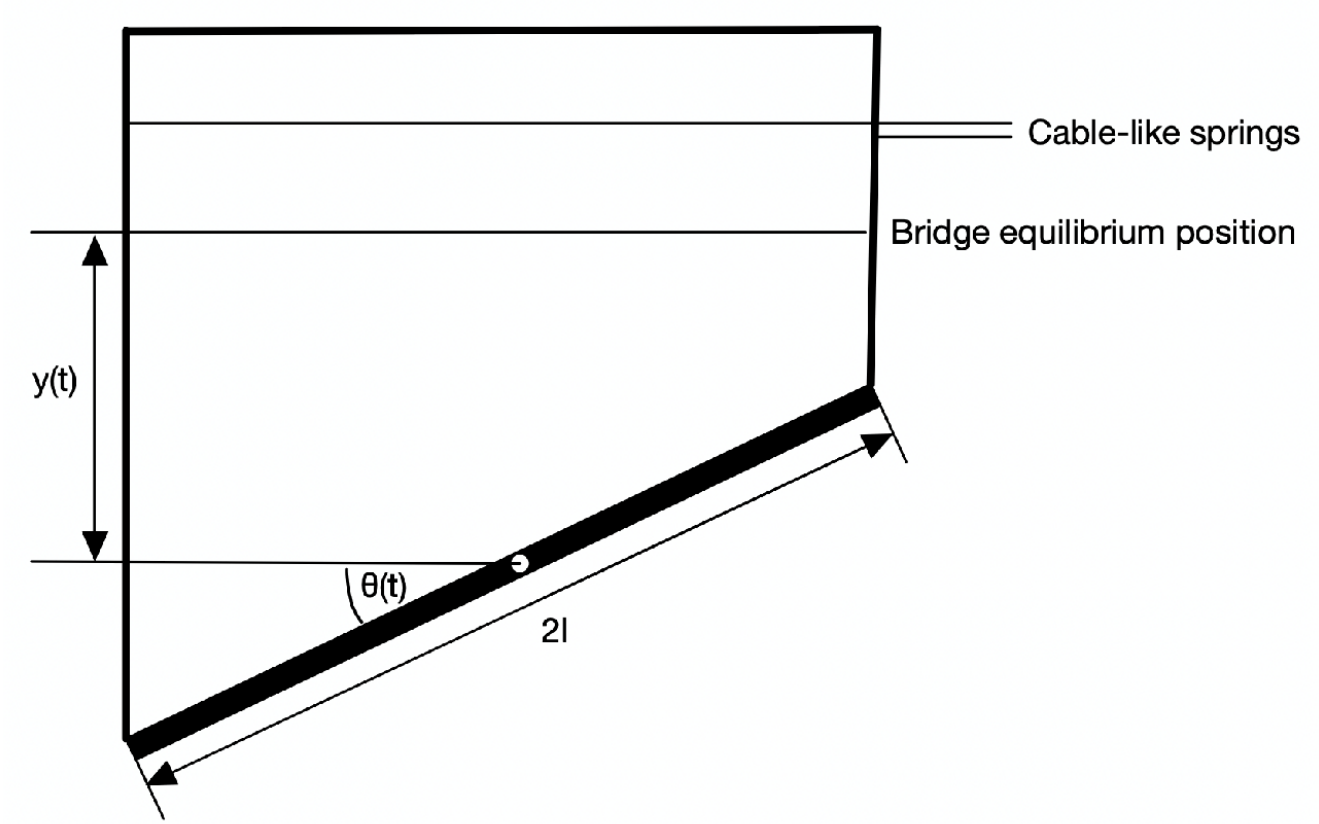
3.2 Theory Behind Modelling The Tacoma Bridge Collapse

In order to model the behaviour of the Tacoma bridge under different initial conditions, we model the torsion angle θ and the vertical displacement from its equilibrium position y of a segment of bridge of width l. This segment is assumed to be supported by two cable-like springs, which are modelled using Hooke's

In [5]:
```
plt.figure(figsize=(35,15))
plt.imshow(plt.imread('Tacoma Diagram.png'))
plt.title("Figure 12: A Diagram Showing a Modelled Segment of The Tacoma bridge",fontsize=2
7.5)
plt.axis('off')
```

Out[5]:
```
(-0.5, 1667.5, 1063.5, -0.5)
```

Figure 12: A Diagram Showing a Modelled Segment of The Tacoma bridge



Since values of y are more negative than positive, the Hooke's law model is modified into an exponential form. In this case, the value of the force constant K is reduced by a constant factor equal to a in what is referred to as the 'McKenna model.' This model uses the conservation of angular momentum and Newton's second law in order to derive equations of motion for $y''(t)$ and $\theta''(t)$, which are shown in Equations 3 and 4. This notation refers to the second differential of y and θ with respect to and as functions of time. Here, d refers to the dissipative constant which produces friction for the segment of bridge and m refers to the mass of the bridge segment. The initial conditions for these values and Equations 3 and 4 are as follows:

$$\begin{aligned}(y, \theta) &= (0, 0)\\ (y', \theta') &= (z, \gamma) = (0, 0)\end{aligned}$$

$$y''(t) = -d[y'] - \tfrac{K}{ma}[exp(a[y - l\sin(\theta)]) + exp(a[y + l\sin(\theta)]) - 2] \quad (3)$$

$$\theta''(t) = -d[\theta'] + \tfrac{3\cos(\theta)}{l}\tfrac{K}{ma}[exp(a[y - l\sin(\theta)]) - exp(a[y + l\sin(\theta)])] \quad (4)$$

These second order coupled differential equations can be decoupled into four first order differential equations in order to solve them numerically for $y(t)$ and $\theta(t)$. The decoupling of Equations 3 and 4 is achieved by using substitutions shown in Equations 5 and 7, to produce Equations 6 and 8 below. This produces the definition of z and γ as the respective vertical and angular velocities of the bridge segment:

$$y' = z \quad (5)$$

$$z' = -d[z] - \frac{K}{ma}[exp(a[y - l\sin(\theta)]) + exp(a[y + l\sin(\theta)]) - 2] \quad (6)$$

$$\theta' = \gamma \quad (7)$$

$$\gamma' = -d[\gamma] + \frac{3\cos(\theta)}{l}\frac{K}{ma}[exp(a[y - l\sin(\theta)]) - exp(a[y + l\sin(\theta)])] \quad (8)$$

The next modification we shall make before using the Taylor and Cromer methods to solve these equations is by adding a driving wind force in the form of Equation 9 below to modify the right hand side of Equation 6. Here, A is the wind force amplitude and ω is the angular velocity of the wind force driving the segment of bridge:

$$f_{\text{wind}} = A\sin(\omega t) \quad (9)$$

In order to solve these equations, we propose two methods. Firstly when the Taylor method is used, the recurrence relations shown from Equations 10 to 13 are implemented. Whereas when the Cromer method is used, the recurrence relations shown by Equations 10 and 11 are implemented for z and γ, and Equations 14 and 15 are used for $\theta(t)$ and $y(t)$:

$$z_{n+1} = z_n + dt[z'] \quad (10)$$

$$\gamma_{n+1} = \gamma_n + dt[\gamma'] \quad (11)$$

$$y_{n+1} = y_n + dt[z_n] \quad (12)$$

$$\theta_{n+1} = \theta_n + dt[\gamma_n] \quad (13)$$

For Cromer's method:

$$y_{n+1} = y_n + dt[z_{n+1}] \quad (14)$$
$$\theta_{n+1} = \theta_n + dt[\gamma_{n+1}] \quad (15)$$

As we will later explore, while both these methods produce similar results for y and θ for small values of time, their differences become more apparent for larger values of time. Both these recurrence relations are coded into a function called 'tacoma_with_wind,' which we define in the following section.

3.3 Defining The 'tacoma_with_wind' Function

In [3]:
```
#Simulation times:
tstart=0 #start time simulation
tend=800 #end time simulation

#Physical parameters:
d = 0.01 #Dissipative (friction) coefficient [s^-1]
a = 0.1 #Power in the exponential [m^-1]
M = 2500 #Bridge mass [kg]
K = 1000 #Hooke's constant [N]
l = 12 #Length of bridge [m]
```

The code in the cell below implements the theory in the previous section to define a function called tacoma_with_wind.

The function takes the arguements:

- *dt* - the time step used in the bridge simulation
- *Cromer* - a boolean input which is 'False' if the Taylor method is used and 'True' if the Cromer method is used
- *y0* - the initial vertical displacement of the bridge
- *z0* - the initial vertical velocity of the bridge
- *theta0* - the initial angle that the bridge is diaplaced by
- *gamma0* - the initial angular velocity of the bridge

Which outputs the three arrays:

- Array 0: list of time vlaues for the bridge
- Array 1: list of values of θ (the torsion angle of the bridge)
- Array 2: list of values of y (vertical displacement of the bridge)

The structure of the code is as follows:

1. Generating and initialising arrays to store results, for A) time, B) y and z and C) theta and gamma.
2. Iterating over these arrays to integrate the model, A) by defining exponentials e_1 and e_2 used in the McKenna model. This is followed by iterating over the arrays for B) z, C) gamma and D) y and theta depending on whether Cromer or Taylor's method is used.

In [4]:
```
#Code to define the funciton:
def tacoma_with_wind(dt=0.1, Cromer=False, y0=0, z0=0, theta0=0.1, gamma0=0, A=0, omega=3):
    ##1. Generating & initialising arrays to store results
        #1A: For time
    times=np.arange(tstart, tend+dt, dt)
    number_of_time_steps=len(times)

        #1B: For y, z
    y=np.zeros(number_of_time_steps)
    y[0]=y0

    z=np.zeros(number_of_time_steps)
    z[0]=z0

        #1C: For theta, gamma
    theta=np.zeros(number_of_time_steps)
    theta[0]=theta0

    gamma=np.zeros(number_of_time_steps)
    gamma[0]=gamma0


    ## 2. Iterate over times to integrate the model:
    for n in range(number_of_time_steps-1):
        #2A: Code exponentials
        e_1=np.exp(a*(y[n]-l*np.sin(theta[n])))
        e_2=np.exp(a*(y[n]+l*np.sin(theta[n])))


        #2B: Iterate for z
        z_coeff=(-1*K)/(M*a)
        z_dot=(-1.0*d*z[n]) +(z_coeff*(e_1 +e_2 -2)) + (A/M)*np.sin(omega*(n*dt)) #the last
term in this
```

```
                                                                                                    #line add
s wind
        z[n+1]=z[n] + dt*(z_dot) #Taylor's method equation for z


        #2C: Iterate for gamma
        gamma_coeff=(3*K*np.cos(theta[n]))/(l*M*a)
        gamma_dot=(-1*d*gamma[n]) +gamma_coeff*(e_1 -e_2)
        gamma[n+1]=gamma[n] + dt*(gamma_dot) #Taylor's method equation for gamma



        #2D: For y and theta depending on whether Cromer or Taylor's method is used:
        if Cromer==True:
            y[n+1]=y[n] + dt*z[n+1]
            theta[n+1]=theta[n] + dt*gamma[n+1]

        if Cromer==False:
            y[n+1]=y[n] + dt*z[n]
            theta[n+1]=theta[n] + dt*gamma[n]
    return times, theta, y


#Code to test the function:
d = 0.01
tacoma with wind(dt=0.1, Cromer=False, y0=0, z0=0, theta0=0.1, gamma0=0, A=0, omega=3)
```

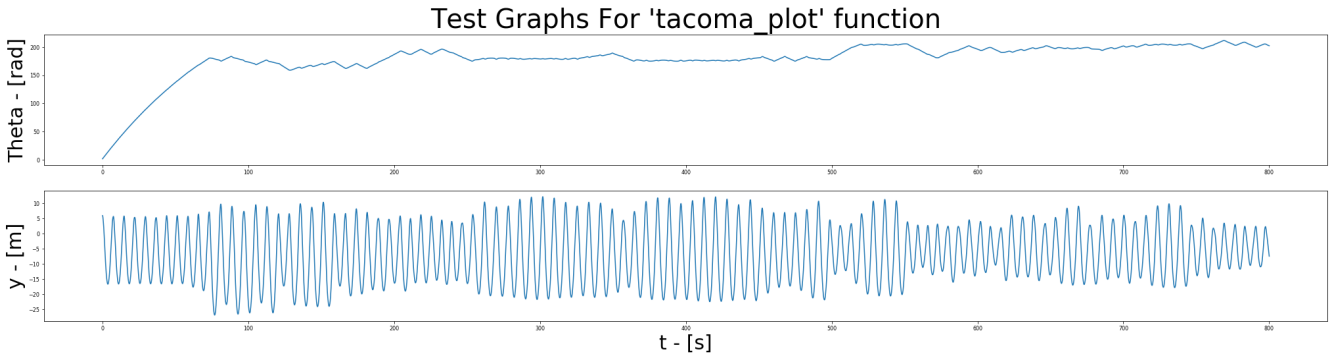
Out[4]:
```
(array([0.000e+00, 1.000e-01, 2.000e-01, ..., 7.998e+02, 7.999e+02,
        8.000e+02]),
 array([1.00000000e-01, 1.00000000e-01, 9.76102613e-02, ...,
        6.13006145e+08, 6.13123843e+08, 6.13241422e+08]),
 array([ 0.00000000e+00,  0.00000000e+00, -5.74769493e-04, ...,
        -7.89491450e+09, -7.89643023e+09, -7.89794444e+09]))
```

3.4 Results & Exploring Bridge Behaviour Under Different Initial Conditions

3.4.1 Exploring Bridge Behaviour In The Absence of Wind

This section repeatedly generates time series of θ and y for the Tacoma bridge under different initial conditions in order to explore its behaviour with the 'tacoma_with_wind' function. To avoid unnecessarily repeating code producing these time series, a simple function called 'tacoma_plot' is first defined in the cell below. This produces graphs for the time series of θ and y under given initial conditions and is defined as follows:

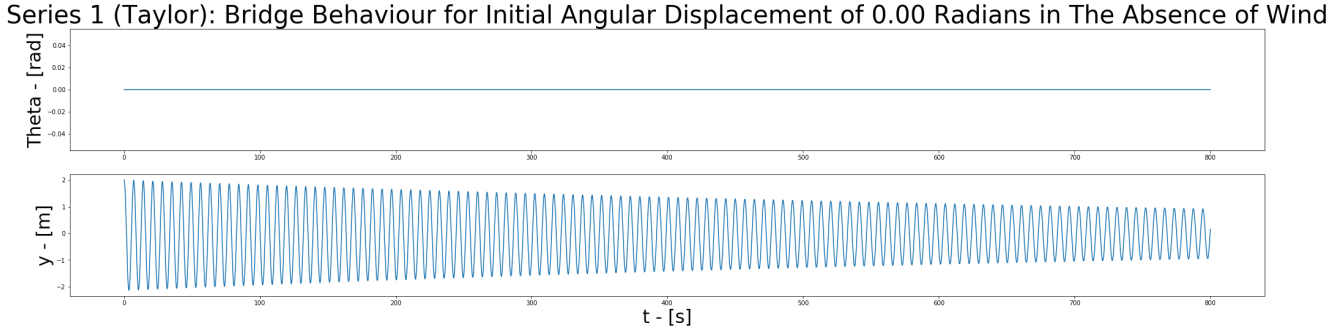
In [217]:
```
#defining the function:
def tacoma_plot(time_step=0.01,Cromer_or_Taylor=False,y_initial=0,
                z_initial=0,theta_initial=0.1,gamma_initial=0, wind_force_amplitude=0, omega
_wind=0,
                graph_title = "Graphs of Vertical Displacement & Theta vs Time For The Tacom
a Bridge",
                figure_width= 10,figure_height=10, y_label_font_size=11,
                 x_label_font_size=11, title_font_size = 12):


    times,theta,y = tacoma_with_wind(dt=time_step, Cromer=Cromer_or_Taylor,
                        y0=y_initial, z0=z_initial, theta0=theta_initial,
                                    gamma0=gamma_initial, A=wind_force_amplitude, omega=om
ega_wind)

    #time series for theta
    plt.figure(figsize=(figure_width,figure_height))
    plt.subplot(2,1,1)
    plt.title(graph_title, fontsize=title_font_size)
    plt.plot(times, theta)
    plt.ylabel('Theta - [rad]', fontsize=y_label_font_size)


    #time series for y
    plt.subplot(2,1,2)
    plt.plot(times, y)
    plt.xlabel('t - [s]', fontsize=x_label_font_size)
    plt.ylabel('y - [m]', fontsize=y_label_font_size)
    plt.figure(figsize=(figure_width,figure_height))
    plt.show()

#code to test the function using arbitrary initial conditions:
d = 0.01
tacoma_plot(time_step=0.01,Cromer_or_Taylor=False,y_initial=6,
                z_initial=0,theta_initial=3.14/2,gamma_initial=3, wind_force_amplitude=1, om
ega_wind=3,
                graph_title = "Test Graphs For 'tacoma_plot' function",
                figure_width= 35,figure_height=8, y_label_font_size=30,
                 x_label_font_size=30, title_font_size = 40)
```



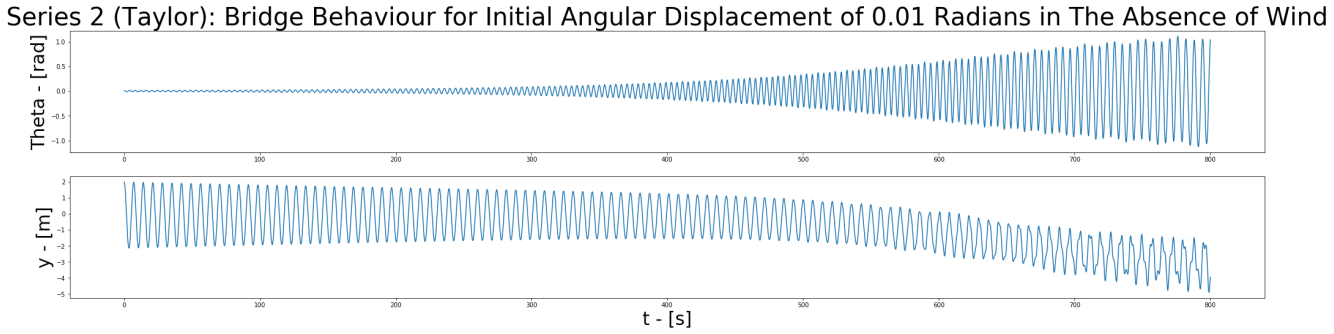

```
<Figure size 2520x576 with 0 Axes>
```

The behaviour of the bridge in the absence of wind can be explored using this function by using a wind force amplitude A equal to 0 in the function. Here, we use the Taylor method to generate different time series for y and θ under different initial angular displacements of the bridge in the absence of wind and an initial value of y equal to 2 metres. This is for $\theta = 0, 0.01$ and 0.1 radians:

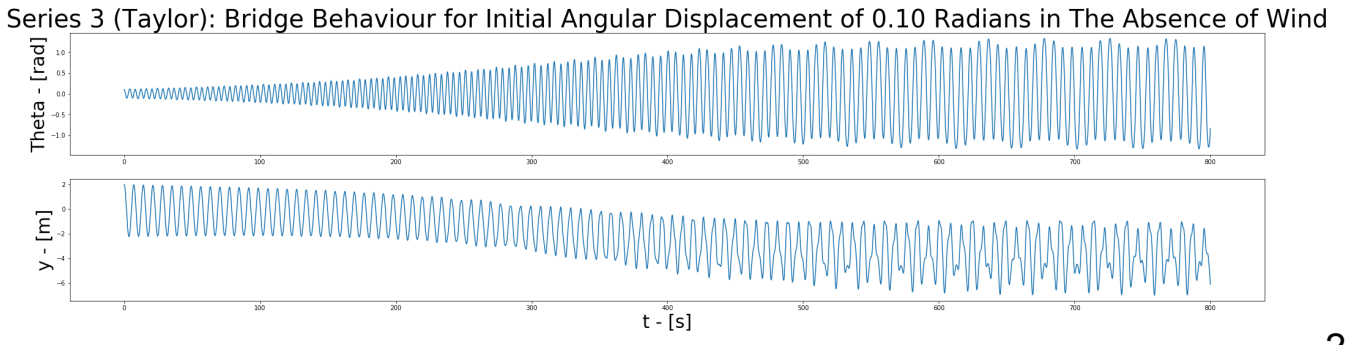
In [59]:
```
Cromer_or_Taylor = False #false for Taylor true for Cromer

#plot graphs
d = 0.01
tacoma_plot(0.01,Cromer_or_Taylor,2,0,0,0, 0,0,
    "Series 1 (Taylor): Bridge Behaviour for Initial Angular Displacement of 0.00 Radians i
n The Absence of Wind",
     35,8,30, 30, 40) #theta = 0 rad
tacoma_plot(0.01,Cromer_or_Taylor,2,0,0.01,0, 0,0,
    "Series 2 (Taylor): Bridge Behaviour for Initial Angular Displacement of 0.01 Radians i
n The Absence of Wind",
     35,8,30, 30, 40)  #theta = 0.01 rad
tacoma_plot(0.01,Cromer_or_Taylor,2,0,0.1,0, 0,0,
    "Series 3 (Taylor): Bridge Behaviour for Initial Angular Displacement of 0.10 Radians i
n The Absence of Wind",
     35,8,30, 30, 40) #theta = 0.1 rad
```




```
<Figure size 2520x576 with 0 Axes>
```



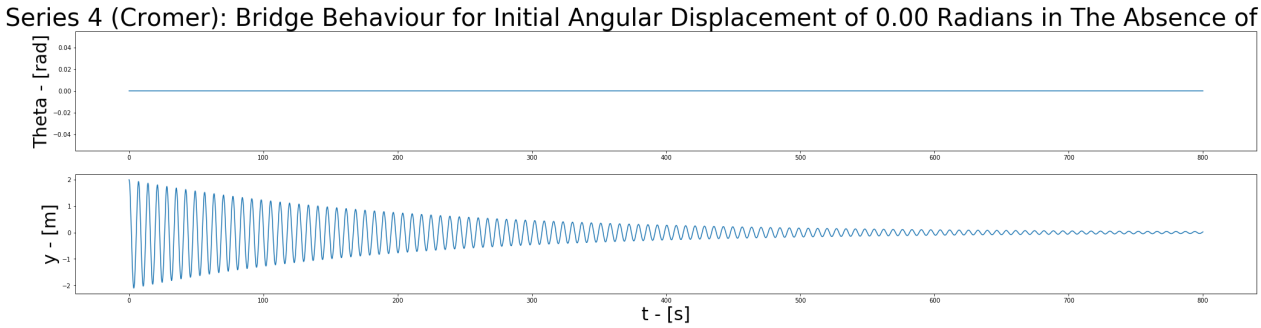

```
<Figure size 2520x576 with 0 Axes>
```



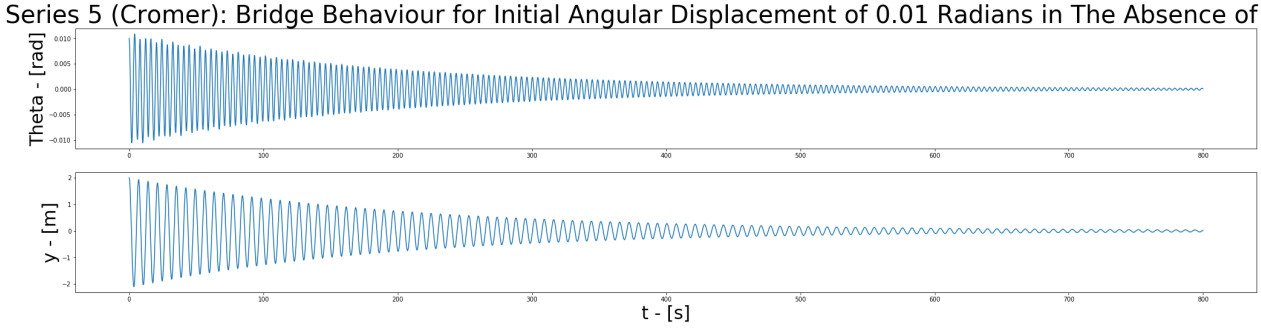

```
<Figure size 2520x576 with 0 Axes>
```

An identical process is repeated in the cell below but instead via the Cromer method:

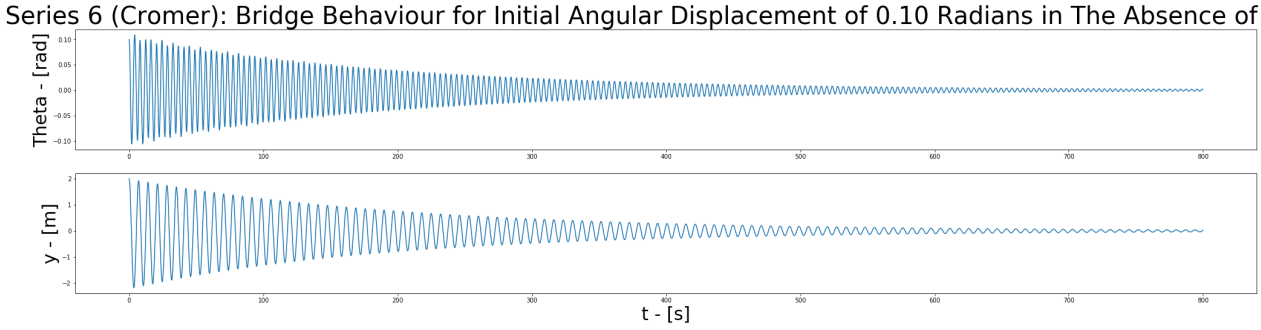
In [61]:
```
Cromer_or_Taylor = True #false for Taylor true for Cromer

#plot graphs
d = 0.01
tacoma_plot(0.01,Cromer_or_Taylor,2,0,0,0, 0,0,
    "Series 4 (Cromer): Bridge Behaviour for Initial Angular Displacement of 0.00 Radians i
n The Absence of Wind",
     35,8,30, 30, 40) #theta = 0 rad
tacoma_plot(0.01,Cromer_or_Taylor,2,0,0.01,0, 0,0,
    "Series 5 (Cromer): Bridge Behaviour for Initial Angular Displacement of 0.01 Radians i
n The Absence of Wind",
     35,8,30, 30, 40)  #theta = 0.01 rad
tacoma_plot(0.01,Cromer_or_Taylor,2,0,0.1,0, 0,0,
    "Series 6 (Cromer): Bridge Behaviour for Initial Angular Displacement of 0.10 Radians i
n The Absence of Wind",
     35,8,30, 30, 40) #theta = 0.1 rad
```




```
<Figure size 2520x576 with 0 Axes>
```




```
<Figure size 2520x576 with 0 Axes>
```




```
<Figure size 2520x576 with 0 Axes>
```

It is notable that for small values of time both Cromer and Taylor methods produce similar time series for y and θ, validating the model. However, for large values of time a disparity in the time series produced by both methods emerges. The model is further validated as both methods produce straight line graphs for the time series of θ in an absence of wind and initial angular displacement of zero radians in Series 1 and 4. Next, we explore the behaviour of the bridge segment in the presence of wind.

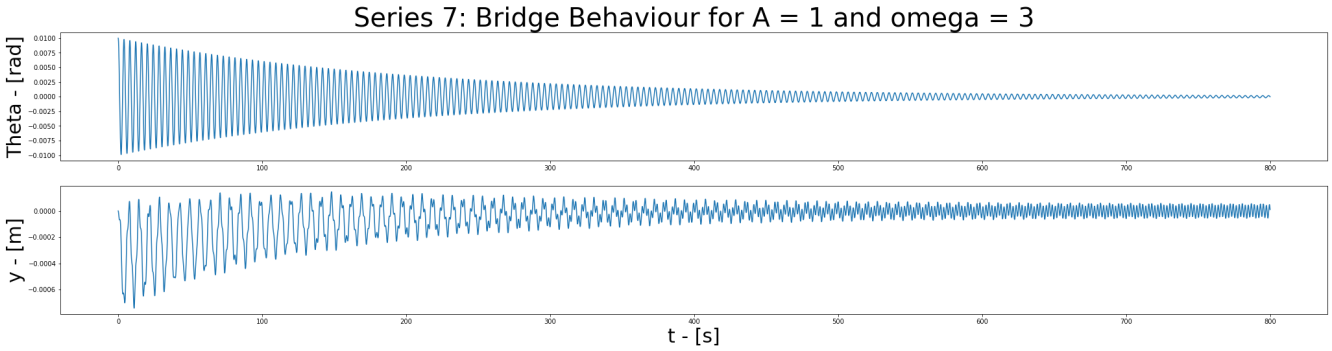
3.4.2 Exploring Bridge Behaviour In The Presence of Wind

The code in the cell below produces time series for $\omega = 3$ in the presence of wind, first for $A = 1$ then followed by $A = 2$, for an initial vertical displacement equal to 0 metres:

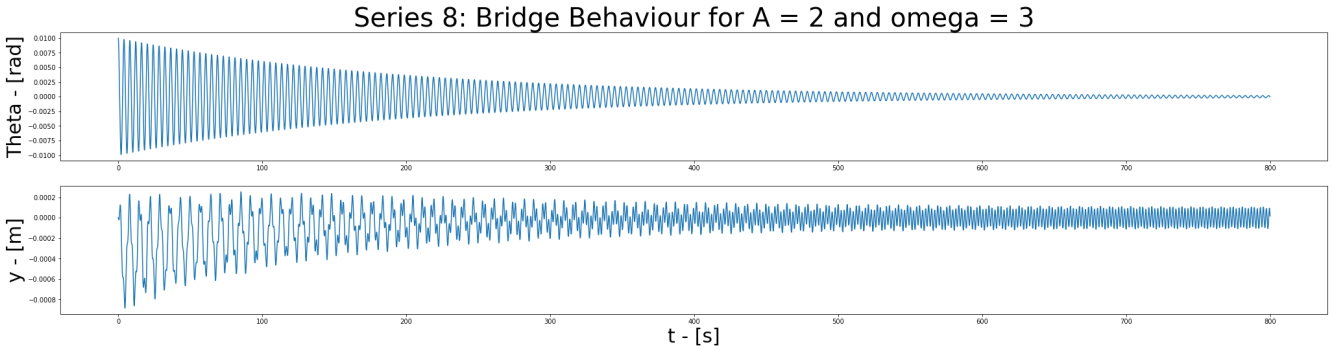

```
Cromer_or_Taylor = True #false for Taylor true for Cromer
```

In [62]:
```
#Taylor or Cromer
if Cromer_or_Taylor==True:
    print("This is using the Cromer method:")
if Cromer_or_Taylor==False:
    print("This is using the Taylor method:")

#plot graphs
d = 0.01
tacoma_plot(0.01,Cromer_or_Taylor,0,0,0.01,0, 1, 3, "Series 7: Bridge Behaviour for A = 1 a
nd omega = 3",35,8,30,
            x_label_font_size=30, title_font_size = 40) #A = 1
tacoma_plot(0.01,Cromer_or_Taylor,0,0,0.01,0, 2,3, "Series 8: Bridge Behaviour for A = 2 an
d omega = 3",
            35,8,30, 30,40) #A = 2
```

```
This is using the cromer method:
```




```
<Figure size 2520x576 with 0 Axes>
```




```
<Figure size 2520x576 with 0 Axes>
```

While both series 7 and 8 produce similar profiles of graph for θ, the time series for y they produce are of different amplitudes. This is because the highest value on the y axis for the y time series in Series 7 is 0.0000m, whereas this is 0.0002m for Series 8. This is similar for the lowest value on the y axis of the y time series for Series 7 which is -0.0006m, a value which is -0.0008m in Series 8. This suggests that the wind force conditions used in Series 8 produce the most resonance of the bridge, generating a larger amplitude of y.

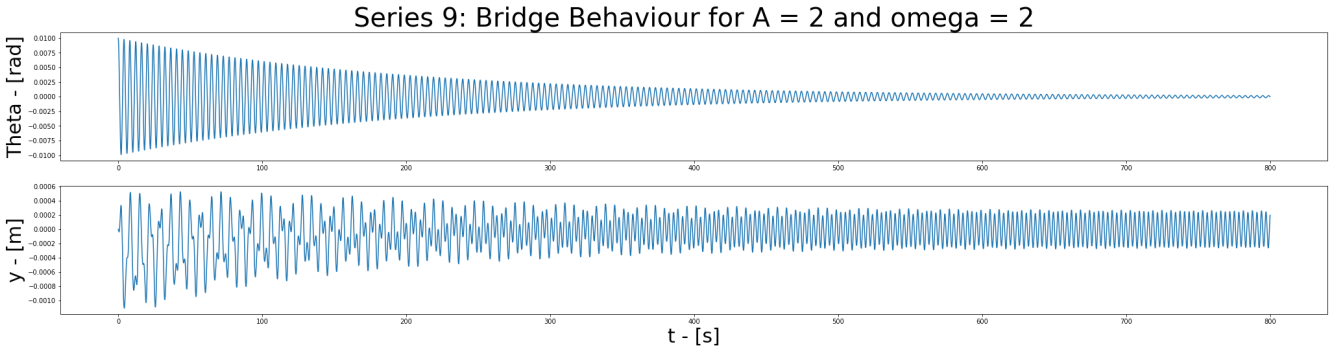
It is also notable that as t becomes larger, the y time series for this graph continues to display more resonance. This suggests that the collapse of the Tacoma bridge was more likely to occur under the initial conditions for wind used in Series 8 as opposed to those used in Series 7, with a value of $A = 2$. The time series for θ are likely similar in both Series 7 and 8 because the initial value of θ used was small and the wind force added acts in the vertical (y) direction.

Using this value of $A = 2$, we shall now explore time series for a range of different values of ω, in order to judge which value also produces the most resonance. This is done in the cell below for four different values of ω: $\omega = 2, 2.5, 2.8$ and 3.3:

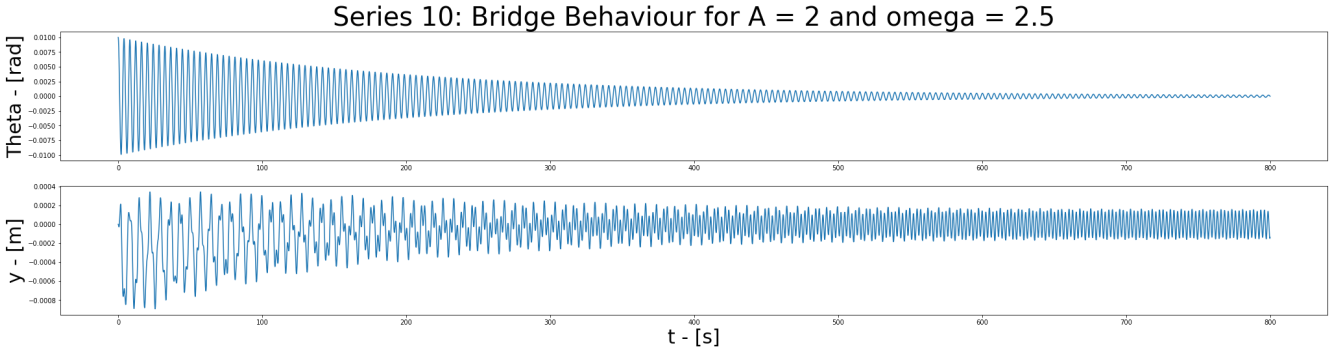
In [73]:
```
Cromer_or_Taylor = True #false for Taylor true for Cromer

#Taylor or Cromer
if Cromer_or_Taylor==True:
    print("This is using the Cromer method:")
if Cromer_or_Taylor==False:
    print("This is using the Taylor method:")
```

```
#plot graphs:
d = 0.01
tacoma_plot(0.01,Cromer_or_Taylor,0,0,0.01,0,2,2, "Series 9: Bridge Behaviour for A = 2 and
omega = 2",
            35,8,30, 30,40) #omega = 2
tacoma_plot(0.01,Cromer_or_Taylor,0,0,0.01,0,2,2.5, "Series 10: Bridge Behaviour for A = 2
 and omega = 2.5",
            35,8,30, 30,40) #omega = 2.5
tacoma_plot(0.01,Cromer_or_Taylor,0,0,0.01,0,2,2.8, "Series 11: Bridge Behaviour for A = 2
 and omega = 2.8",
            35,8,30, 30,40) #omega = 2.8
tacoma_plot(0.01,Cromer_or_Taylor,0,0,0.01,0,2,3.3, "Series 12: Bridge Behaviour for A = 2
 and omega = 3.3",
            35,8,30, 30,40) #omega = 3.3
```

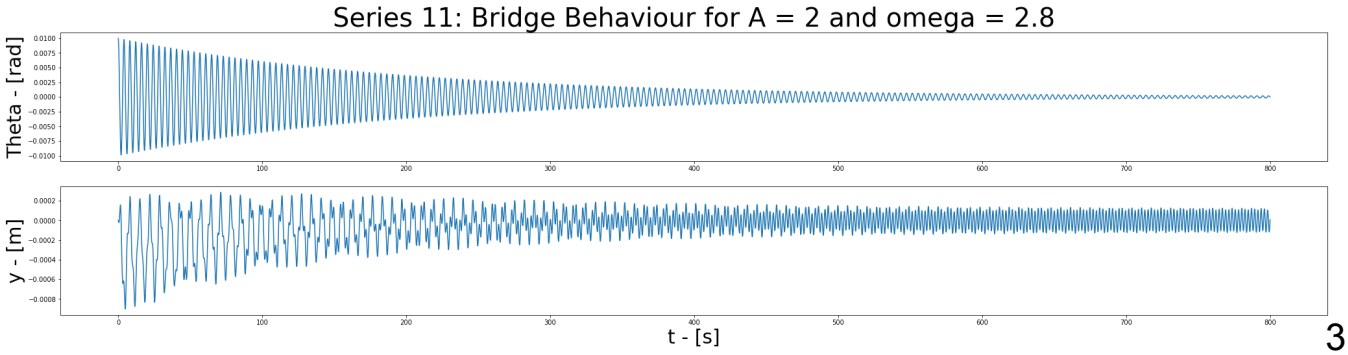
This is using the cromer method:



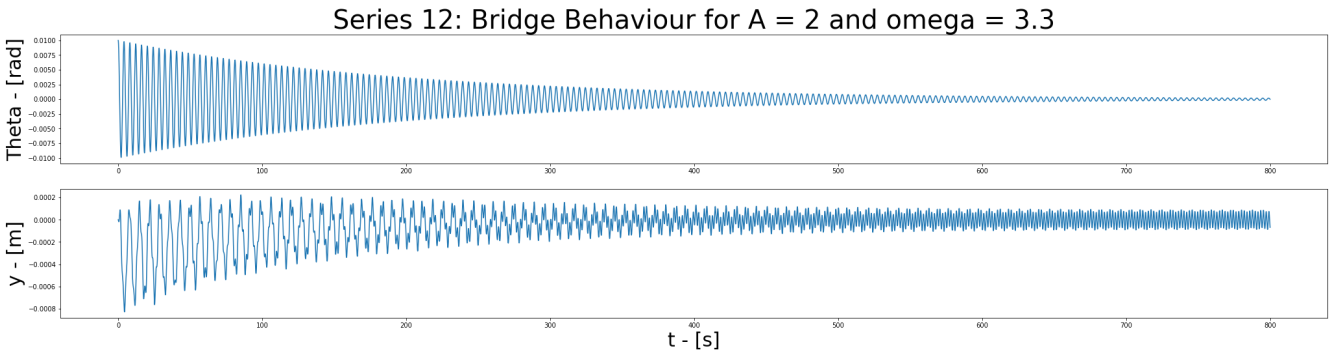
<Figure size 2520x576 with 0 Axes>



<Figure size 2520x576 with 0 Axes>



```
<Figure size 2520x576 with 0 Axes>
```



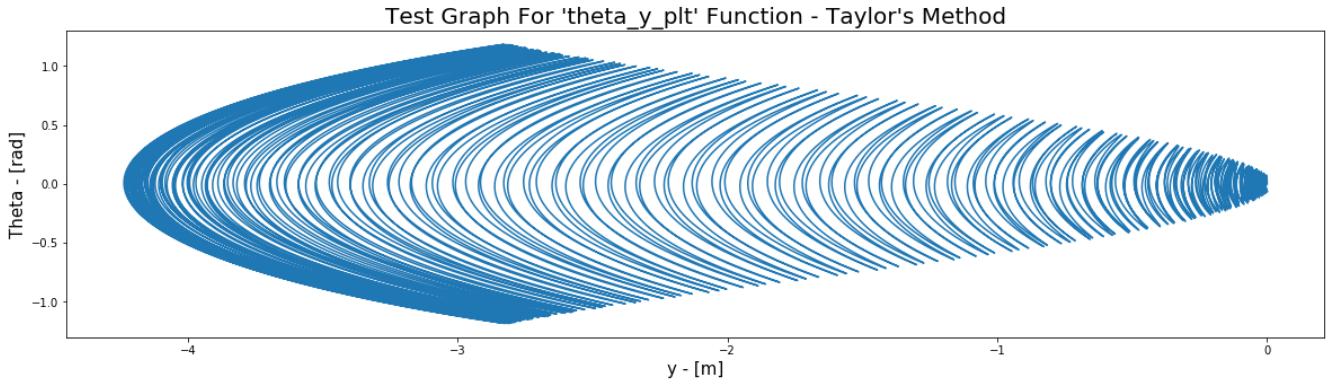

```
<Figure size 2520x576 with 0 Axes>
```

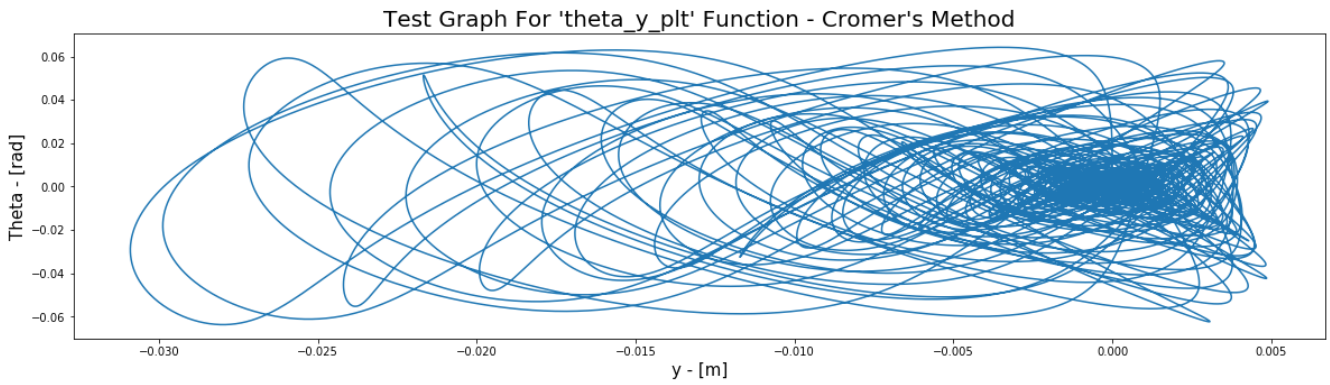
This indeed suggests that the value of ω which produces the largest resonance in the values trialled is close to 2.8 radians per second. This is because the y time series for this value of ω shown in Series 11 is the least chaotic. The increasing values of ω showin in Series 9-10 up to this value have increasing resonance up to the y time seres shown in Series 12 which is increasingly chaotic. This denotes that the value of ω which produces maximal resonance for the bridge segment is in the region of 2.8 radians per second to 3.3 radians per second. In the next section, as opposed to producing time series for y and θ, we produce plots of θ against y. Since this is done multiple tmies we first define a function called 'theta_y_plt.'

3.4.3 Time Trajectories for Torsion Angle with Vertical Displacement for The Tacoma Bridge

In order to produce a series of graphs of $\theta(t)$ on the vertical (y) axis against $y(t)$ on the horizontal (x) axis, below we define a function which has a similar purpose as the 'tacoma_plot' function:

In [11]:
```
#define the function:
def theta_y_plt(time_step=0.01,Cromer_or_Taylor=False,y_initial=0,
                z_initial=0,theta_initial=0.1,gamma_initial=0, wind_force_amplitude=0, omega
_wind=0,
                graph_title = "Graph of Torsion Angle Against Vertical Displacement For The
 Tacoma Bridge",
                figure_width= 10,figure_height=10, y_label_font_size=11,
                 x_label_font_size=11, title_font_size = 12):

    times,theta,y = tacoma_with_wind(dt=time_step, Cromer=Cromer_or_Taylor,
                        y0=y_initial, z0=z_initial, theta0=theta_initial,
                                    gamma0=gamma_initial, A=wind_force_amplitude, omega=om
ega_wind)

    #plot graph
    plt.figure(figsize=(figure_width,figure_height))
    plt.title(graph_title, fontsize=title_font_size)
    plt.plot(y, theta)
    plt.ylabel('Theta - [rad]', fontsize=y_label_font_size)
    plt.xlabel('y - [m]', fontsize=x_label_font_size)
    plt.show()

#code to test the function using arbitrary initial conditions:
d = 0.01
theta_y_plt(time_step=0.01,Cromer_or_Taylor=False,y_initial=0,
                z_initial=0,theta_initial=0,gamma_initial=0.1, wind_force_amplitude=0, omega
_wind=0,
                graph_title = "Test Graph For 'theta_y_plt' Function - Taylor's Method",
                figure_width= 20,figure_height=5, y_label_font_size=15,
                 x_label_font_size=15, title_font_size = 20)

theta_y_plt(time_step=0.01,Cromer_or_Taylor=True,y_initial=0,
                z_initial=0,theta_initial=0,gamma_initial=0.1, wind_force_amplitude=0, omega
_wind=0,
                graph_title = "Test Graph For 'theta_y_plt' Function - Cromer's Method",
                figure_width= 20,figure_height=5, y_label_font_size=15,
                 x_label_font_size=15, title_font_size = 20)
```

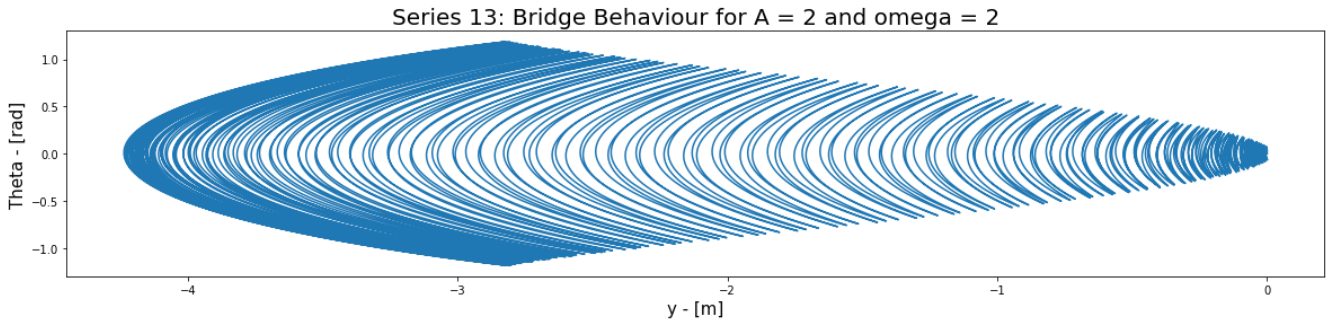


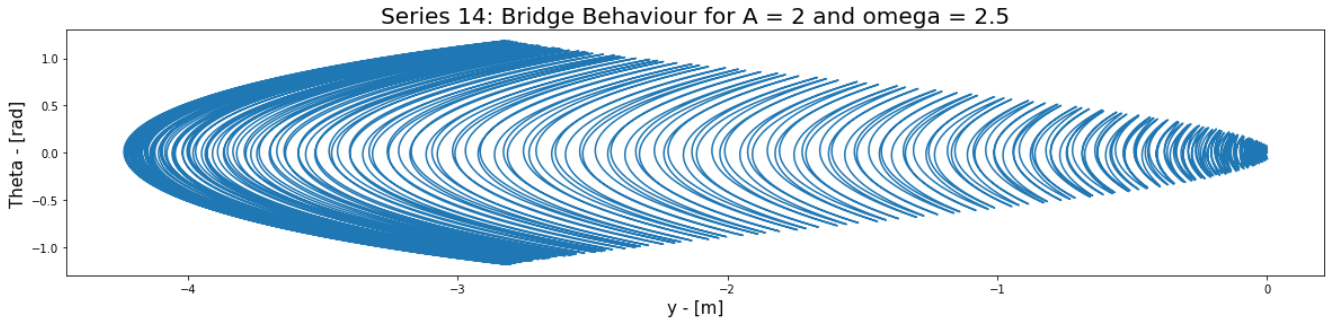


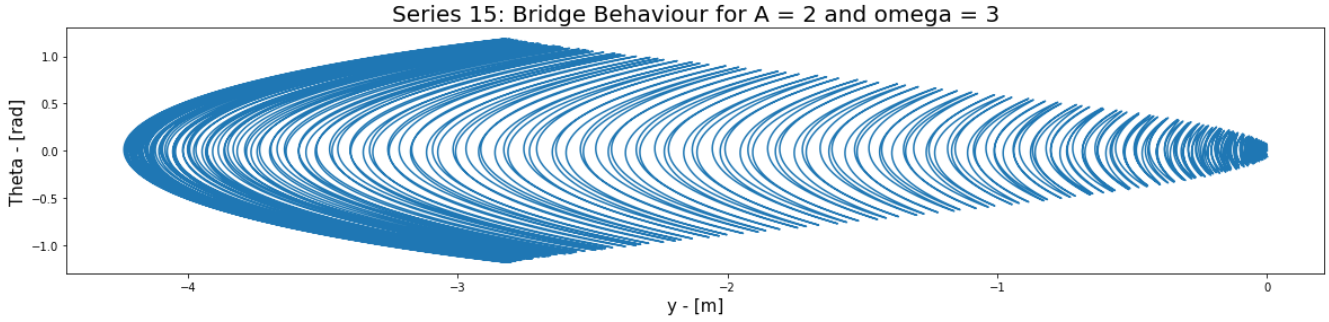
We now use this function with Taylor's method to generate five graphs for $A = 2$ and $\omega\space[2,4]$:

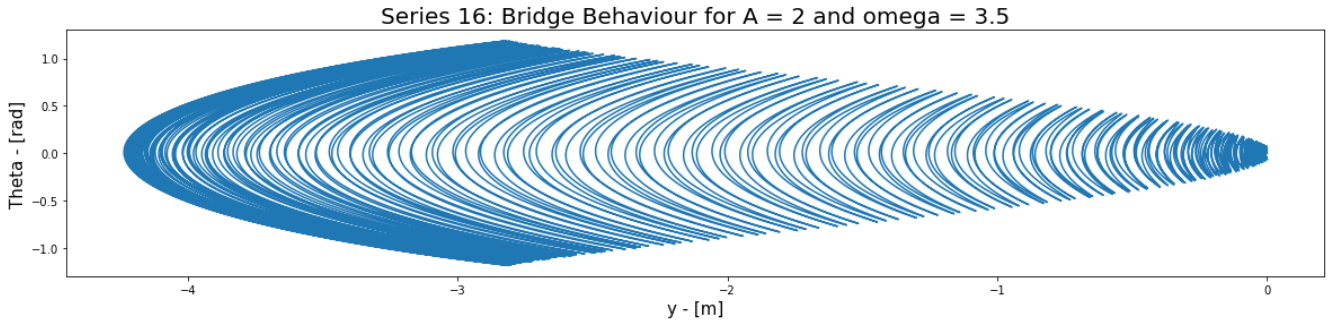
In [14]:
```
#inputs
Cromer_or_Taylor = False #false for Taylor true for Cromer
omega_1 = 2
omega_2 = 2.5
omega_3 = 3
omega_4 = 3.5
omega_5 = 4

#Taylor or Cromer
if Cromer_or_Taylor==True:
    print("This is using the Cromer method:")
if Cromer_or_Taylor==False:
    print("This is using the Taylor method:")

#plot graphs
d = 0.01
theta_y_plt(0.01,Cromer_or_Taylor,0,0,0,0.1, 2, omega_1,
                "Series 13: Bridge Behaviour for A = 2 and omega = 2",20,4, 15, 15,20) #grap
h 1
theta_y_plt(0.01,Cromer_or_Taylor,0,0,0,0.1, 2, omega_2,
                "Series 14: Bridge Behaviour for A = 2 and omega = 2.5",20,4, 15, 15,20) #gr
aph 2
theta_y_plt(0.01,Cromer_or_Taylor,0,0,0,0.1, 2, omega_3,
                "Series 15: Bridge Behaviour for A = 2 and omega = 3",20,4, 15, 15,20) #grap
h 3
theta_y_plt(0.01,Cromer_or_Taylor,0,0,0,0.1, 2, omega_4,
                "Series 16: Bridge Behaviour for A = 2 and omega = 3.5",20,4, 15, 15,20) #gr
aph 4
theta_y_plt(0.01,Cromer_or_Taylor,0,0,0,0.1, 2, omega_5,
                "Series 17: Bridge Behaviour for A = 2 and omega = 4",20,4, 15, 15,20) #grap
h 5
```

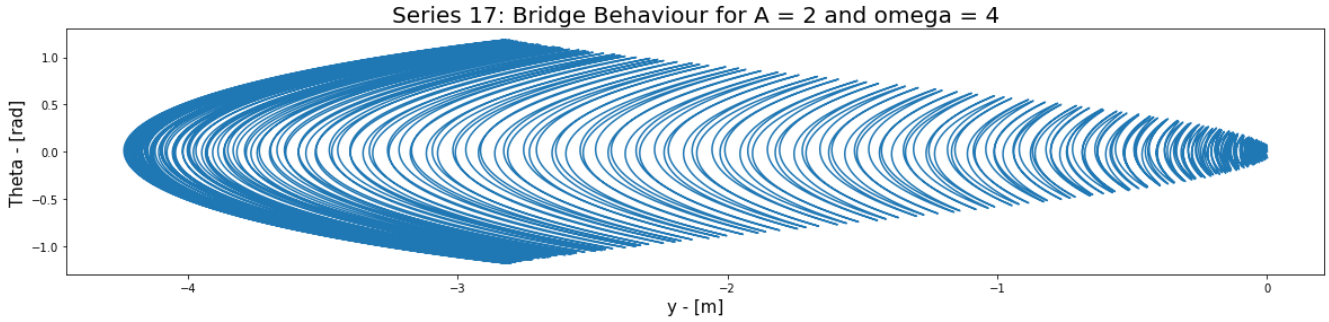
```
This is using the Taylor method:
```







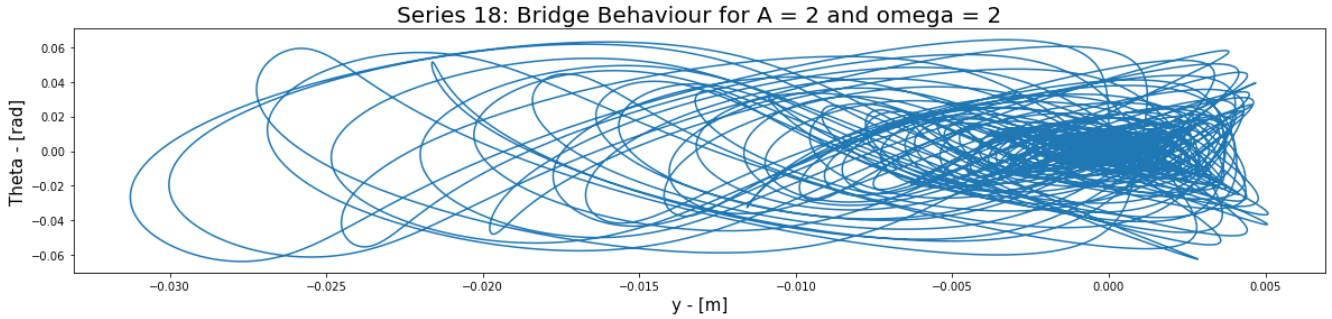


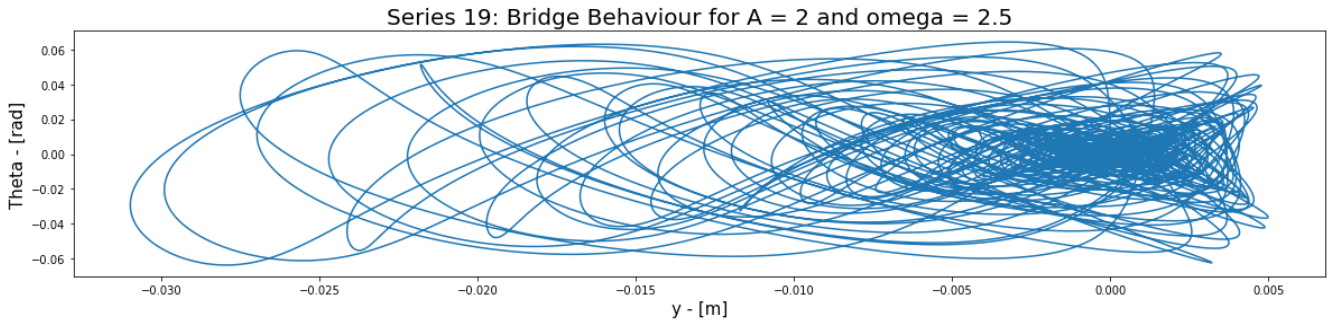


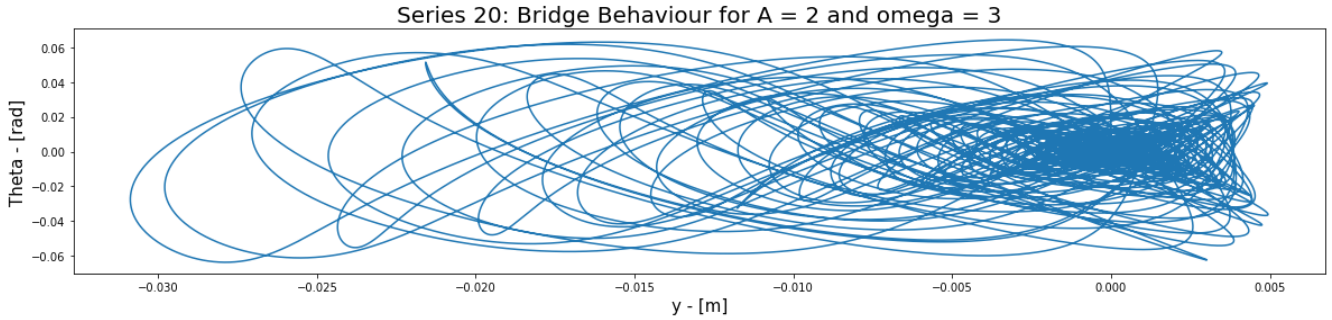
The code below is identical to that of the previous cell but instead uses the Cromer method:

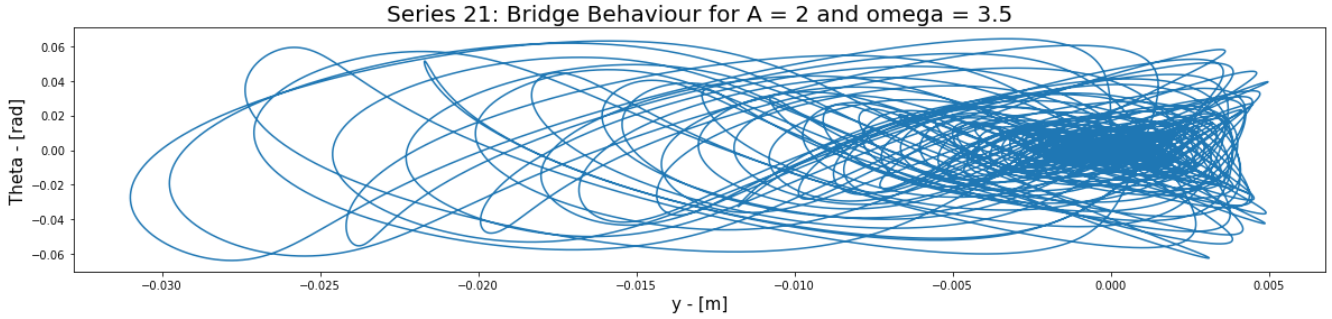
In [65]:
```
#inputs
Cromer_or_Taylor = True #false for Taylor true for Cromer
omega_1 = 2
omega_2 = 2.5
omega_3 = 3
omega_4 = 3.5
omega_5 = 4

#Taylor or Cromer
if Cromer_or_Taylor==True:
    print("This is using the Cromer method:")
if Cromer_or_Taylor==False:
    print("This is using the Taylor method:")

#plot graphs
d = 0.01
theta_y_plt(0.01,Cromer_or_Taylor,0,0,0,0.1, 2, omega_1,
                "Series 18: Bridge Behaviour for A = 2 and omega = 2",20,4, 15, 15,20) #grap
h 1
theta_y_plt(0.01,Cromer_or_Taylor,0,0,0,0.1, 2, omega_2,
                "Series 19: Bridge Behaviour for A = 2 and omega = 2.5",20,4, 15, 15,20) #gr
aph 2
theta_y_plt(0.01,Cromer_or_Taylor,0,0,0,0.1, 2, omega_3,
                "Series 20: Bridge Behaviour for A = 2 and omega = 3",20,4, 15, 15,20) #grap
h 3
theta_y_plt(0.01,Cromer_or_Taylor,0,0,0,0.1, 2, omega_4,
                "Series 21: Bridge Behaviour for A = 2 and omega = 3.5",20,4, 15, 15,20) #gr
aph 4
theta_y_plt(0.01,Cromer_or_Taylor,0,0,0,0.1, 2, omega_5,
                "Series 22: Bridge Behaviour for A = 2 and omega = 4",20,4, 15, 15,20) #grap
h 5
```

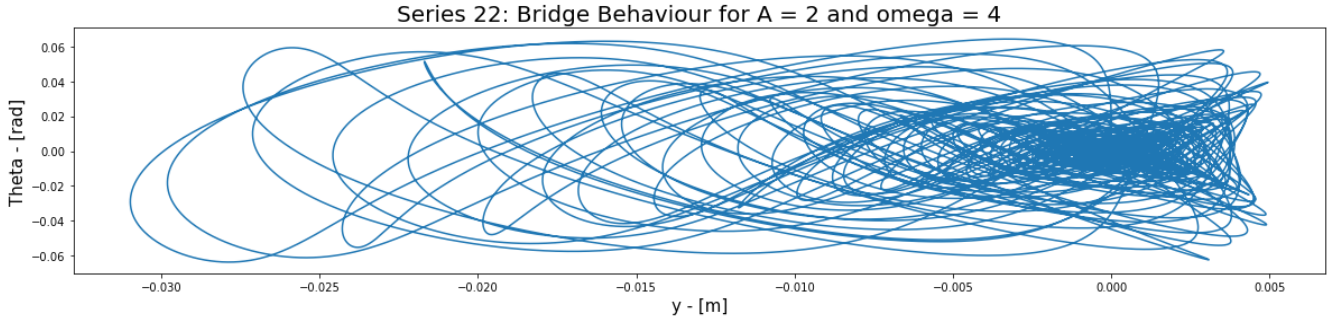
This is using the cromer method:









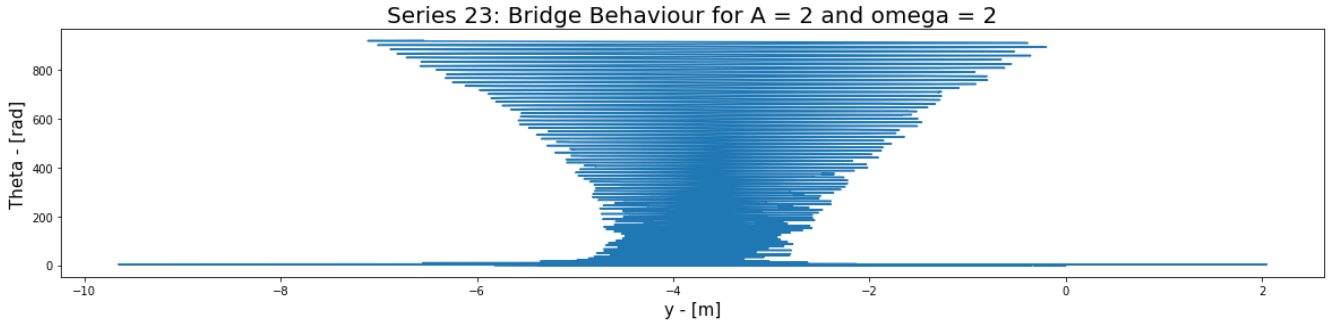


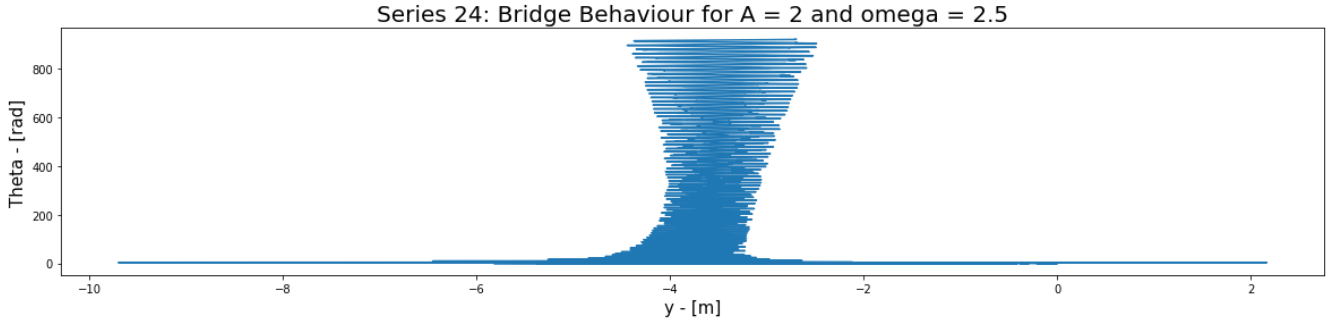
Although these series seem very similar, they each differ slightly upon closer inspection. After experimentation, the differences between them are more easily seen by using Taylor's method and a dissipative constant equal to zero ($d = 0$). This is the case in which the bridge does no work against friction, producing the plots below:

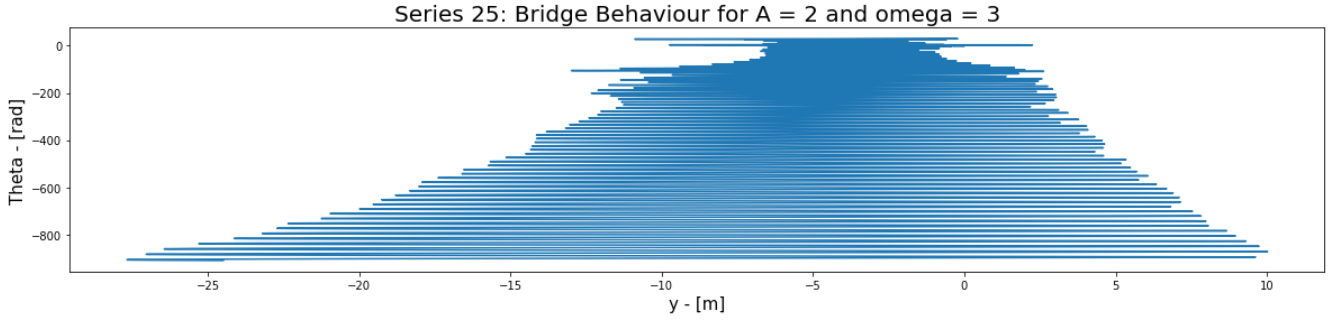
In [75]:
```
#inputs
Cromer_or_Taylor = False #false for Taylor true for Cromer
omega_1 = 2
omega_2 = 2.5
omega_3 = 3
omega_4 = 3.5
omega_5 = 4
```

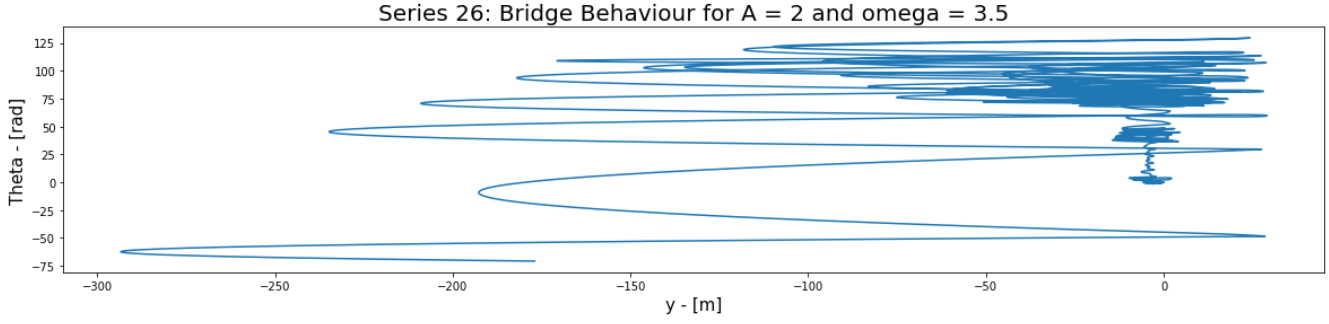
```
#Taylor or Cromer
if Cromer_or_Taylor==True:
    print("This is using the Cromer method and a dissipative constant equal to 0:")
if Cromer_or_Taylor==False:
    print("This is using the Taylor method and a dissipative constant equal to 0:")

#plot graphs
d = 0
theta_y_plt(0.01,Cromer_or_Taylor,0,0,0,0.1, 2, omega_1,
            "Series 23: Bridge Behaviour for A = 2 and omega = 2",20,4, 15, 15,20) #grap
h 1
theta_y_plt(0.01,Cromer_or_Taylor,0,0,0,0.1, 2, omega_2,
            "Series 24: Bridge Behaviour for A = 2 and omega = 2.5",20,4, 15, 15,20) #gr
aph 2
theta_y_plt(0.01,Cromer_or_Taylor,0,0,0,0.1, 2, omega_3,
            "Series 25: Bridge Behaviour for A = 2 and omega = 3",20,4, 15, 15,20) #grap
h 3
theta_y_plt(0.01,Cromer_or_Taylor,0,0,0,0.1, 2, omega_4,
            "Series 26: Bridge Behaviour for A = 2 and omega = 3.5",20,4, 15, 15,20) #gr
aph 4
theta_y_plt(0.01,Cromer_or_Taylor,0,0,0,0.1, 2, omega_5,
            "Series 27: Bridge Behaviour for A = 2 and omega = 4",20,4, 15, 15,20) #grap
h 5
```

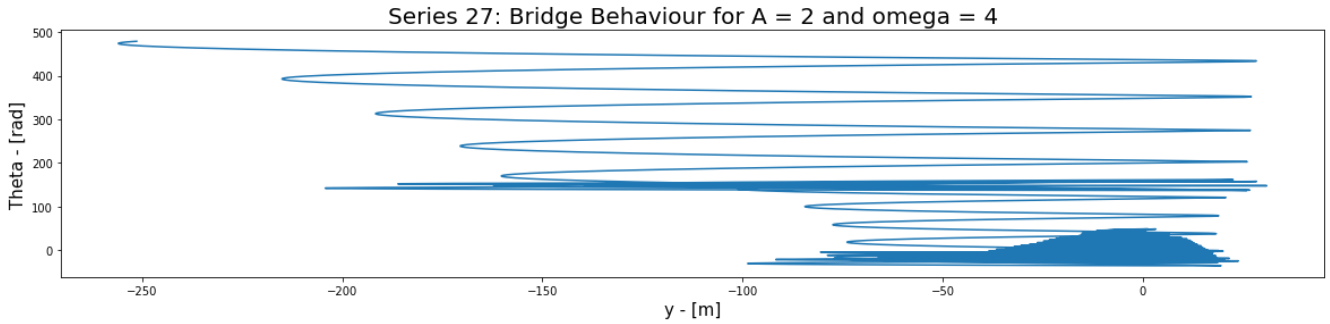
```
This is using the taylor method and a dissipative constant equal to 0:
```









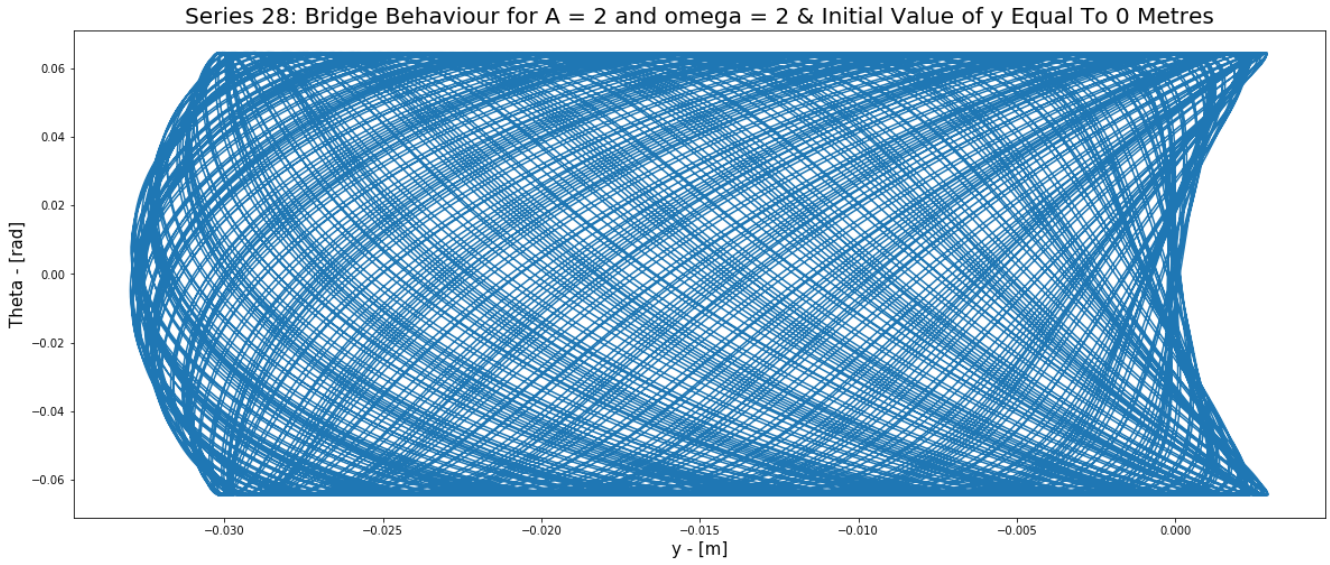


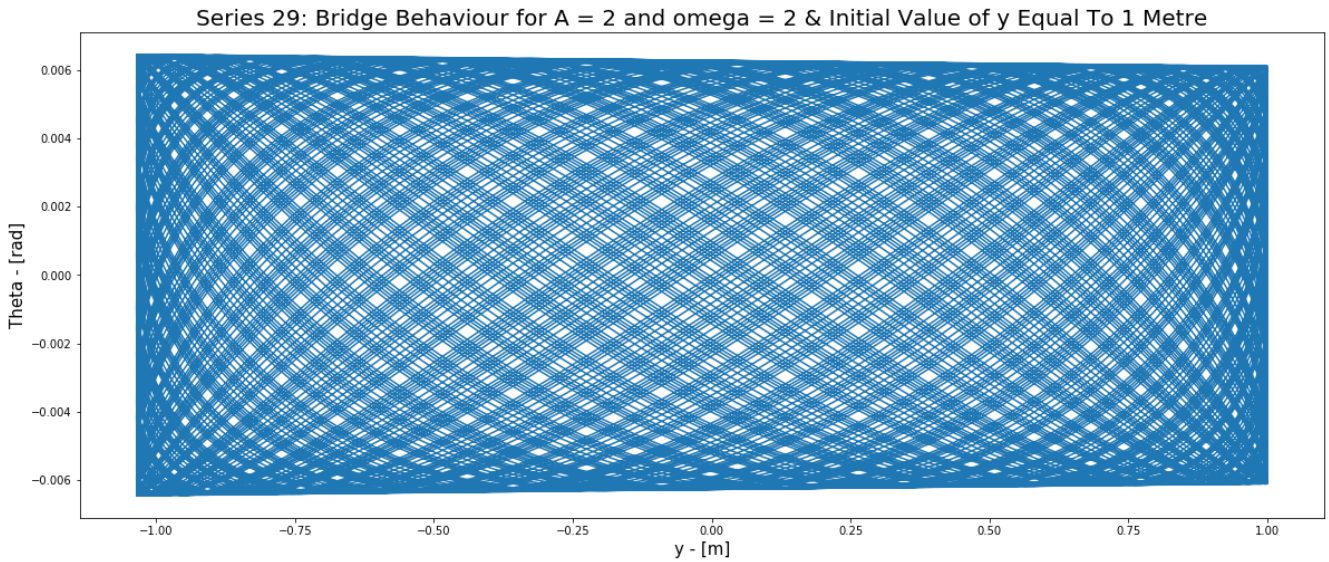
While Series 23-25 are more ordered, Series 27-28 are more chaotic. Curiously, the Series which is ordered and produces the largest amplitude for y (Series 25) has a value of $\omega = 3$. This matches values of ω for the Tacoma bridge while it collapsed which can be obtained by watching footage of the bridge oscillating in the moments before its demise. This suggests that the angular velocity of the wind which caused the bridge to collapse did indeed match the natrual (resonant) frequency of the bridge, although modelling the wind force using a simple sine wave may be an oversimplification. In order to improve this model, different profiles for the wind force may therefore be implimented. In the next section, we discuss ideas for extending the problem further.

For interest and completeness, in the case where $d = 0$ the Cromer method produces mesh shapes as such:

In [103]:
```
d = 0
print("This is using the Cromer method and a dissipative constant equal to 0:")
theta_y_plt(0.01,True,0,0,0,0.1, 2, omega_1,
            "Series 28: Bridge Behaviour for A = 2 and omega = 2 & Initial Value of y Equal
To 0 Metres",
            20,8, 15, 15,20)

theta_y_plt(0.01,True,1,0,0,0.01, 2, omega_1,
            "Series 29: Bridge Behaviour for A = 2 and omega = 2 & Initial Value of y Equal
To 1 Metre",
            20,8, 15, 15,20)
```

```
This is using the cromer method and a dissipative constant equal to 0:
```





3.5 Extending The Tacoma Bridge Problem

3.5.1 Ideas Extending The Tacoma Bridge Problem

The first idea we mention to extend the Tacoma bridge problem is finding the critical wind speed for the bridge to collapse and comparing it to the literature value of the wind speed on the day the bridge collapsed. A relationship between f_{wind} and the wind speed would be obtained experimentally, for instance by placing a model of the bridge in a wind tunnel and gathering data.

Secondly, the Millennium bridge has been hailed as the 'wobbly bridge' due to the impacts of pedestrians driving it to oscillate. Furthermore, soldiers are commonly instructed to break strike while marching across bridges in order to avoid oscillations as such. It is notable that in footage of the Tacoma bridge collapsing there is only one car on the bridge while it oscillates. Should the wind force in the numerical model of the Tacoma bridge be replaced by the force of pedestrians walking in synchrony across the bridge, a comparison may be made of the number of pedestrians required in order to garner an equivalent affect from the wind. Here, a form of pedestrian-wind speed ratio would be made.

A final idea for extending the problem is using models for bridge simulations which are not based on the assumption that sections of the bridge can be modelled as springs, comparing results from these models to those from the McKenna model. The behaviour of the bridge under different conditions for gravitational strength and damping coefficients can be explored to simulate oscillations of the bridge on different planets or in different climates, for instance. A modified McKenna model could also be implemented to simulate using springs to earthquake-proof buildings in different wind conditions and compared to if a pendulum were used in order to judge which would be most effective.

As an extension to the problem, the next part of this section discusses the literature surrounding numerically modelling pedestrians as they drive bridge oscillations in models referred to as ‘foot-force’ models.

3.5.2 A Discussion of Foot- Force Models in Literature

3.5.2.1 Bridge Failure Due to Crowd Synchrony

In this section, ‘bridge failure’ refers to the inability of a bridge to perform its function as being a safe structure for pedestrians to walk across, rather than its total demise. As discussed, the Millennium bridge has been hailed the ‘wobbly bridge’ due to wobbling caused by pedestrians. Other examples of bridges which observe this effect are Japan’s Maple Valley Great Suspension Bridge and Bristol’s Clifton suspension bridge. As with the Tacoma bridge problem, a solution commonly used in these cases to produce safer structures is mass dampers, which alter the resonant frequency of the bridge and thereby reducing wobbling.

The phenomenon of pedestrians walking in phase across a bridge and causing it to wobble is known in literature as ‘pedestrian phase locking’ [15]. The synchrony of these pedestrians as such exists in a branch of Physics called cooperative dynamics. In this case, this means similar rules can be used to define the generic pedestrian walking across the bridge in what is known as a ‘foot-force’ model. For instance, by each pedestrian having the same mass and the same biomechanical model.

Bridge oscillations are produced when the frequency pedestrians walk at match the natural (resonant) frequency of the bridge and there are a sufficient number of pedestrians for bridge resonance to occur. In this section, we later explore an equation for this critical crowd size. While these conditions are satisfied, a maximum rate of energy transfer between the pedestrians and the bridge is achieved - resulting in a maximum amplitude for bridge oscillations and causing bridge failure to occur. Once bridge oscillations occur, pedestrians are also forced to walk in more synchrony, which in turn increases the rate of this energy transfer. Bridge failure under different initial conditions for crowd synchrony can be modelled using ‘foot-force’ models, examples of which we discuss below.

3.5.2.2 Examples of Foot-Force Models

Foot-force models are used to model the impact of crowd synchrony on bridges. While bridges are modelled using pendulum models, pedestrians are commonly modelled using either an 'inverted-pendulum' model or Van der Pol type model [15]. The pendulum model used to model bridges simplifies the bridge into a horizontal platform attached to horizontal pairs of springs on either side, in a 'mass-spring-damper' system [15].

The first model used to emulate pedestrians is a Van der Pol type model, which is a series of differential equations governing pedestrian motion with trigonometric solutions. The contributions of each pedestrian to the model can be summed, producing discrete second order differential equations when pedestrians are phase locked.

The second pedestrian model is the inverted pendulum model. Here, the term 'inverted pendulum' refers to a metronome-like structure, supported by a left and right leg. One second order differential equation of motion is obtained for each leg of the inverted pendulum and solved numerically. In comparison to the Van der Pol type model, this model uses a closed conservative system with no damping, which results in hyperbolic (as opposed to trigonometric) solutions for each leg. In this method the frequency of oscillation of the inverted pendulum is also assumed constant. Solving these models numerically for the bridges and pedestrians leads to engineers being able to simulate the critical number of pedestrians required to be walking across the bridge in order for it to oscillate, which we explore in the next section.

3.5.2.3 Critical Crowd Size for Bridge Oscillations

The critical number of pedestrians walking in synchrony for bridge resonance is denoted by N_c. This is equated to a product of parameters in equation 18, which represent the following quantities [16]:

- K - the spring constant used to model the bridge in the pendulum model
- M - the bridge mass
- B - the damping coefficient of the bridge
- $\Omega_{0}=\sqrt{K/M}$ - the natrual (resonant) frequency of the bridge
- G - the magmitude of the horizontal force each pedestrian exherts on the bridge
- C - a measure of the responsiveness of pedestrians to bridge oscillations
- $P(\Omega)$ - the distribution of stride frequencies each pedestrian walks with

$$N_c = \frac{4B}{\pi\sqrt{4MK}}\left(\frac{K}{GCP(\Omega_0)}\right) \quad (18)$$

When used on the Millennium bridge, this equation produces an approximate critical crowd size of 165 people. On footage of the bridge when it first opened and was oscillating it is notable that the crowd sizes on the bridge are well in excess of this number. A 2017 research paper published by Research Gate used numerical methods to produce the graph reproduced in Figure 13 for the Millennium bridge [16], which supports this value of N_c approximately equal to 165 pedestrians. It is notable that after this threshold crowd size is exceeded, the amplitude of bridge oscillations increases linearly with an increasing number of pedestrians walking in synchrony.

Adding damping to the Millennium bridge increased its value of N_c and altered its resonant frequency to the point beyond which would physically occur, and thereby eliminated the majority of its wobbling. In a similar way, this begs the question of if the Tacoma Narrows bridge had been initially constructed using denser materials or more damping whether its oscillations and so collapse would have been prevented. Alternatively, if its value of N_c was small enough to see crowd sizes which could have led to it to wobble in the event that it was a pedestrian only bridge like the Millennium bridge. This shows the importance of using numerical modelling to simulate the behaviour of proposed bridge designs for the purpose of public safety and the prevention of bridge failure.

Next, we ask four key questions regarding the future of research for foot-force models and begin to conclude the report.

In [32]:
```
plt.figure(figsize=(35,15))
plt.imshow(plt.imread('Millenium Bridge.png'))
plt.title("Figure 13: A Diagram Showing Amplitude of Horizontal Oscillations Against Number of Pedestrians for The Millennium Bridge",fontsize=27.5)
plt.axis('off')
```

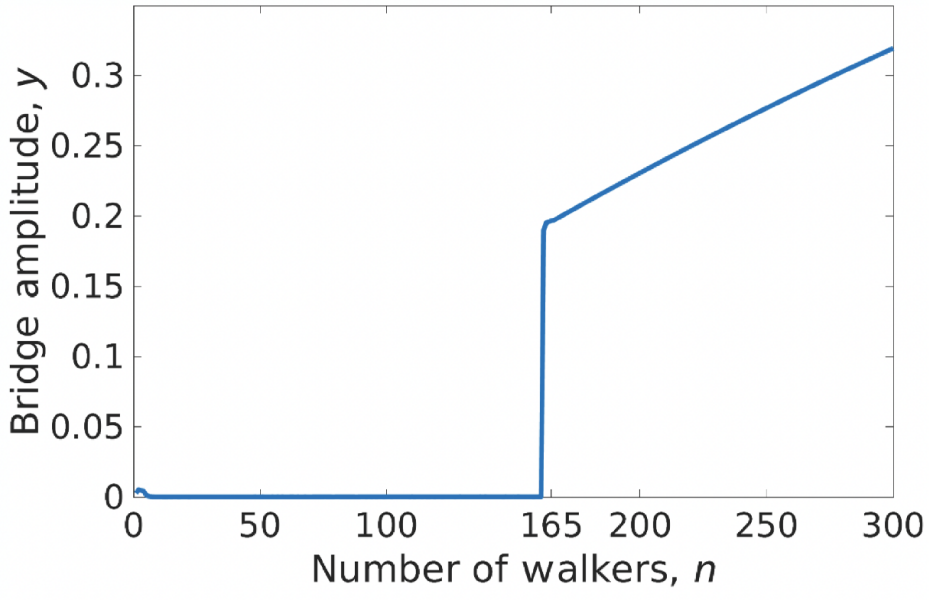
Out[32]:
```
<Figure size 2520x1080 with 0 Axes>
```

Out[32]:
```
<matplotlib.image.AxesImage at 0x7fe0e95837d0>
```

Out[32]:
```
Text(0.5, 1.0, 'Figure 13: A Diagram Showing Amplitude of Horizontal Oscillations Against Number of Pedestrians for The Millennium Bridge')
```

Out[32]:
```
(-0.5, 1947.5, 1261.5, -0.5)
```

Figure 13: A Diagram Showing Amplitude of Horizontal Oscillations Against Number of Pedestrians for The Millennium Bridge



3.5.2.4 Questions For Future Research of Foot-Force Models

The future of research for force-foot models aims to answer some fundamental questions summarised by the four points below:

1. Can bridge oscillations be enforced by secondary oscillations of the bridge as opposed to the force created by crowd synchrony and phase locking alone?
2. Which other conditions would be required, with weather or otherwise for the bridge to wobble?
3. Can the inverted pendulum model be altered to account for knee movements of pedestrians as well as hip movements and how would this extra degree of movement affect the results to numerical models?
4. How do the speeds of crowds change as the number of pedestrians walking across the bridge increase?

4. Conclusion

4.1 Outlooks for The Travelling Salesperson Problem

4.1.1 Finding the Longest Path

In section 2.7, we suggested a final solution to the traveling salesman problem with the objective of producing the tour of shortest distance. The optimal solution for this problem exists in the global minima in the energy landscape of possible paths. Likewise, finding the ‘optimal’ longest path to a similar problem would correspond to finding the path which produced the global maxima in this energy landscape. A similar approach would be used to approximate this global maximum, but with multiple modifications.

Firstly, rather than beginning the local maxima search from points of high energy and continuing onto points of lower energy, the algorithm would start from points of lower energy and move onto points of larger energy. While initial conditions for T_o, α and decay time would be selected in order to maximise the distance of the paths produced, the value of T_o would be made smaller in order to start the local maxima search from a point of lower energy. Furthermore, α would be made negative in order to aid in the exponential increase of the temperature profile. Should exponentially increasing the profile in this way be too harsh, a logarithmic temperature profile would instead be used. In suggested pair-wise exchanges, paths of larger rather than shorter distance would also be favoured. Finally, the code would be repeated in order to find multiple local maxima and approximate the longest obtained path as the global maximum in the energy landscape.

4.1.2 The Importance of Temperature Schedules In Simulated Annealing For The Travelling Salesman Problem

The importance of the temperature schedule for the TSP lies in the acceptance probability for pair-wise exchanges of a current path with a path of larger distance. If the acceptance probability for longer suggested paths is too high (for instance by using a value of α which is too small) the current path continually becomes longer and therefore less optimal. However, if the acceptance probability for pair-wise exchanges with paths of longer distance than the current path is too small, the current path will only be improved if a path of shorter distance is generated. This means that the annealing algorithm produces paths which become ‘stuck’ in local minima in the energy landscape, thereby reducing the probability of obtaining the global minimum. Therefore, a temperature schedule must alter the acceptance probability for suggested pair-wise exchanges which are longer than the current path such that they do not continually increase the distance of the current path, but also do not cause the current path to become trapped in a local minimum. As is the purpose of multiple forms of temperature profile used in literature for the problem, which for instance exist in exponential and logarithmic forms.

4.1.3 Avoiding Cross Country Travel

To solve a TSP problem which avoids cross country travel, cities on the map would be provided weightings based on their distances from the centre coordinate of the map. Alternatively, weightings could be increased based on the shortest distance of each city to the ocean or by using a function. Here, the probability of accepting pair-wise exchanges with cities of larger weightings would be reduced, producing a system which favours coastal (rather than cross-country) pair-wise exchanges.

4.1.4 Applications of Simulated Annealing in Science

Two other applications of simulated annealing in science are in the numerical modelling of protein folding [12] and in electrical power systems [13]. In the modelling of protein folding, proteins made from amino acids form complex chains which fold into various structures which simulated annealing is used to model. In electric power systems, simulated annealing is used for optimisation problems such as GEP (Generation Expansion Planning) and TEP (Transmission Expansion Planning) in order to manage the number of power generators used in a power system based on given constraints, for instance. Next, we discuss outlooks for the Tacoma bridge problem.

4.2 Outlooks for The Tacoma Bridge Problem

4.2.1 Modelling the Entire Tacoma Bridge

Since our exploration of the Tacoma bridge problem only models one bridge section, a natural extension to this would be exploring the behaviour of the bridge as a whole. While the equations of motion used for this section would be similar to those used in Section 3.2, one set of equations would be derived from applying Newton's second law on each bridge section, and another from the conservation of angular momentum. These equations would be manipulated to be of a similar form and unified into a matrix equation which could be solved using the 'linalg' scipy module. The equations could also be modified to account for the McKenna model and would be different for sections on either end of the bridge. Here, simulations could be ran as to how the bridge behaves when it is composed of different numbers of sections and how the force is distributed through sections of bridge while it resonates.

4.2.2 Modelling Other Examples of Resonance in Physics

Two other examples in Physics in which resonant and chaotic motion is displayed are double pendulum systems and the resonance of buildings due to seismic activity. Both these examples have heavy dependence on initial conditions, in that for instance if a double pendulum is initially slightly perturbed then large variations in the chaotic behaviour it later exhibits are observed.

Ordered motion for a double pendulum can be modelled from using basic physical laws and concepts to calculate its fundamental modes. In the simplest case, Newton's second law with polar coordinates and small angle approximations is used to generate linear algebra equations stored in matrix equations. These can be solved for the angular velocities at which the pendula display ordered motion.

The second example in which resonant and chaotic motion is seen is in the resonance of buildings due to seismic activity [14]. Here, layers of rock below buildings act as 'resonant chambers.' How the chambers respond to seismic waves depends on their symmetries and which materials they are comprised of. In the case that the basin 'traps' seismic waves, it resonates in an ordered manner which causes structural damage to buildings constructed on top of it. However, less building damage occurs in the event that the chamber does not trap seismic waves and exhibits more chaotic unpredictable motion. The approach used to model the resonant behaviour of these resonant basins also calculates their fundamental modes and is based on the 'Einstein-Brillouin Keller' or EBK method. This method is traditionally used for calculating electron energy levels among others. However, it is able to successfully predict how resonant chambers of different geometries behave under seismic activity, if for instance they are ridged or otherwise.

4.3 Key Findings & Closing Remarks

Overall, in this course we have learnt of key ideas used in numerical modelling - including topics of:

- Randomness: generating random numbers, using random numbers to numerically solve integration problems and random walk problems
- The Ising model and metropolis algorithm for modelling spin states of dipoles in a lattice under different temperatures
- Numerically solving first and second order differential equations for a pendulum and quantum box using the Taylor method
- Using linear algebra in python to solve systems of equations
- Using singular value decomposition in order to achieve image compression in python
- Machine learning and current topics in Physics which use numerical modelling in python and other coding languages

In this report specifically, we have modelled two examples of optimisation problems: the travelling salesman problem using simulated annealing and Monte Carlo sampling and the historic collapse of the Tacoma Narrows bridge using coupled second order differential equations, the Cromer and Taylor methods.

It was first determined for the travelling salesman problem that the best path between thirty cities on a map of north America had a distance of 2233 units of distance. However, the simulated annealing algorithm was criticised due to its reliance on initial conditions and that the solutions obtained are only heuristic. Using this algorithm for a travelling salesman problem on the London Underground was then discussed and outlooks explored as an extension to the problem. The

significance of these findings demonstrates the importance of heuristic solutions for seemingly simple problems we may yet have no methods to generate an optimal solution for.

Secondly, it was determined that while modelling the Tacoma bridge collapse, the ideal conditions required for the wind force driving the bridge to cause it to collapse were a wind force amplitude equal to 2 Newtons for the modelled section of bridge and an angular velocity equal to 3 radians per second, which matched footage of the bridge as it collapsed. Finally, outlooks for the Tacoma bridge were discussed, with an exploration of literature surrounding foot-force models as an extension to the problem. The significance of these findings show the importance of simulating how new bridge designs respond to the effects of wind and pedestrians before their construction in order to prevent similar bridge collapses and wobbling.

Both problems may ideally be summarised as optimisation problems which exist for the purpose of reducing cost while achieving the function either of a functional bridge or of maximising sales. This basic idea of cost reduction in order to maximise profit with regards to a constraint is a keystone concept of modern society in existing sectors such as investing, production, agriculture and energy and is likely to play a key role in new sectors which have yet to exist.

5. Appendices

The cell below stores the eight obtained paths found for the travelling salesman problem in section 2.7:

In [25]:
```
Path_1 = [('Hartford', (719.6, 205.2)),('Albany', (702.0, 193.6)),('Providence', (735.2, 201.2)),
          ('Boston', (738.4, 190.8)),('Harrisburg', (670.8, 244.0)),('Columbus', (590.8, 263.2)),
          ('Lansing', (563.6, 216.4)),('Madison', (500.8, 217.6)),('Saint Paul', (451.6, 186.0)),
          ('Des Moines', (447.6, 246.0)),('Oklahoma City', (392.8, 356.4)),('Denver', (293.6, 274.0)),
          ('Salt Lake City', (204.0, 243.2)),('Boise', (159.6, 182.8)),('Salem', (80.0, 139.2)),
          ('Sacramento', (68.4, 254.0)),('Phoenix', (179.6, 371.2)),('Austin', (389.2, 448.4)),
          ('Nashville', (546.4, 336.8)),('Indianapolis', (548.0, 272.8)),('Little Rock', (469.2, 367.2)),
          ('Jackson', (501.6, 409.6)),('Baton Rouge', (489.6, 442.0)),('Montgomery', (559.6, 404.8)),
          ('Tallahassee', (594.8, 434.8)),('Atlanta', (585.6, 376.8)),('Columbia', (632.4, 364.8)),
          ('Raleigh', (662.0, 328.8)),('Richmond', (673.2, 293.6)),('Trenton', (698.8, 239.6))]

Path_2 = [('Montgomery', (559.6, 404.8)),('Baton Rouge', (489.6, 442.0)), ('Jackson', (501.6, 409.6)),
          ('Austin', (389.2, 448.4)),('Phoenix', (179.6, 371.2)),('Sacramento', (68.4, 254.0)),
          ('Salem', (80.0, 139.2)),('Boise', (159.6, 182.8)),('Salt Lake City', (204.0, 243.2)),
          ('Denver', (293.6, 274.0)),('Oklahoma City', (392.8, 356.4)),('Little Rock', (469.2, 367.2)),
          ('Indianapolis', (548.0, 272.8)),('Columbus', (590.8, 263.2)),('Madison', (500.8, 217.6)),
          ('Saint Paul', (451.6, 186.0)),('Des Moines', (447.6, 246.0)),('Lansing', (563.6, 216.4)),
          ('Harrisburg', (670.8, 244.0)),('Hartford', (719.6, 205.2)),('Providence', (735.2, 201.2)),
          ('Boston', (738.4, 190.8)),('Albany', (702.0, 193.6)),('Trenton', (698.8, 239.6)),
          ('Richmond', (673.2, 293.6)),('Raleigh', (662.0, 328.8)),('Columbia', (632.4, 364.8)),
          ('Nashville', (546.4, 336.8)),('Atlanta', (585.6, 376.8)),('Tallahassee', (594.8, 434.8))]

Path_3 = [('Oklahoma City', (392.8, 356.4)),('Denver', (293.6, 274.0)),('Salt Lake City', (204.0, 243.2)),
          ('Boise', (159.6, 182.8)),('Salem', (80.0, 139.2)),('Sacramento', (68.4, 254.0)),
          ('Phoenix', (179.6, 371.2)),('Little Rock', (469.2, 367.2)),('Des Moines', (447.6, 246.0)),
          ('Saint Paul', (451.6, 186.0)),('Madison', (500.8, 217.6)),('Indianapolis', (548.0, 272.8)),
          ('Lansing', (563.6, 216.4)),('Columbus', (590.8, 263.2)),('Harrisburg', (670.8, 244.0)),
          ('Trenton', (698.8, 239.6)),('Providence', (735.2, 201.2)),('Boston', (738.4, 190.8)),
          ('Hartford', (719.6, 205.2)),('Albany', (702.0, 193.6)),('Richmond', (673.2, 293.6)),
          ('Raleigh', (662.0, 328.8)),('Columbia', (632.4, 364.8)), ('Atlanta', (585.6, 376.8)),
          ('Tallahassee', (594.8, 434.8)),('Nashville', (546.4, 336.8)), ('Montgomery', (559.6, 404.8)),
          ('Jackson', (501.6, 409.6)),('Baton Rouge', (489.6, 442.0)), ('Austin', (389.2, 448.4))]

Path_4 = [('Trenton', (698.8, 239.6)),('Lansing', (563.6, 216.4)),('Madison', (500.8, 217.6)),
          ('Saint Paul', (451.6, 186.0)),('Des Moines', (447.6, 246.0)),('Indianapolis', (548.0, 272.8)),
          ('Raleigh', (662.0, 328.8)),('Columbia', (632.4, 364.8)),('Atlanta', (585.6, 376.8)),
          ('Montgomery', (559.6, 404.8)),('Tallahassee', (594.8, 434.8)),('Baton Rouge', (4
```

```
89.6, 442.0)),
          ('Oklahoma City', (392.8, 356.4)),('Denver', (293.6, 274.0)),('Salt Lake City', (
204.0, 243.2)),
          ('Boise', (159.6, 182.8)),('Salem', (80.0, 139.2)),('Sacramento', (68.4, 254.0)),
          ('Phoenix', (179.6, 371.2)),('Austin', (389.2, 448.4)),('Jackson', (501.6, 409.6
)),
          ('Little Rock', (469.2, 367.2)),('Nashville', (546.4, 336.8)),('Columbus', (590.8
, 263.2)),
          ('Richmond', (673.2, 293.6)),('Harrisburg', (670.8, 244.0)),('Albany', (702.0, 19
3.6)),
          ('Boston', (738.4, 190.8)),('Providence', (735.2, 201.2)),('Hartford', (719.6, 20
5.2))]

Path_5 = [('Jackson', (501.6, 409.6)),('Atlanta', (585.6, 376.8)),('Columbia', (632.4, 364.
8)),
          ('Raleigh', (662.0, 328.8)),('Richmond', (673.2, 293.6)),('Trenton', (698.8, 239.
6)),
          ('Hartford', (719.6, 205.2)),('Providence', (735.2, 201.2)),('Boston', (738.4, 19
0.8)),
          ('Albany', (702.0, 193.6)),('Harrisburg', (670.8, 244.0)),('Columbus', (590.8, 26
3.2)),
          ('Indianapolis', (548.0, 272.8)),('Lansing', (563.6, 216.4)),('Madison', (500.8,
217.6)),
          ('Saint Paul', (451.6, 186.0)),('Des Moines', (447.6, 246.0)),('Nashville', (546.
4, 336.8)),
          ('Montgomery', (559.6, 404.8)),('Tallahassee', (594.8, 434.8)),('Little Rock', (4
69.2, 367.2)),
          ('Baton Rouge', (489.6, 442.0)),('Austin', (389.2, 448.4)),('Denver', (293.6, 27
4.0)),
          ('Phoenix', (179.6, 371.2)),('Sacramento', (68.4, 254.0)),('Salem', (80.0, 139.2
)),
          ('Boise', (159.6, 182.8)),('Salt Lake City', (204.0, 243.2)),('Oklahoma City', (3
92.8, 356.4))]

Path_6 = [('Tallahassee', (594.8, 434.8)),('Atlanta', (585.6, 376.8)),('Columbia', (632.4,
364.8)),
          ('Raleigh', (662.0, 328.8)),('Richmond', (673.2, 293.6)),('Trenton', (698.8, 239.
6)),
          ('Hartford', (719.6, 205.2)),('Providence', (735.2, 201.2)),('Boston', (738.4, 19
0.8)),
          ('Albany', (702.0, 193.6)),('Harrisburg', (670.8, 244.0)),('Columbus', (590.8, 26
3.2)),
          ('Jackson', (501.6, 409.6)),('Little Rock', (469.2, 367.2)),('Baton Rouge', (489.
6, 442.0)),
          ('Austin', (389.2, 448.4)),('Oklahoma City', (392.8, 356.4)),('Phoenix', (179.6,
371.2)),
          ('Sacramento', (68.4, 254.0)),('Salem', (80.0, 139.2)),('Boise', (159.6, 182.8)),
          ('Salt Lake City', (204.0, 243.2)),('Denver', (293.6, 274.0)), ('Des Moines', (44
7.6, 246.0)),
          ('Saint Paul', (451.6, 186.0)),('Madison', (500.8, 217.6)),('Lansing', (563.6, 21
6.4)),
          ('Indianapolis', (548.0, 272.8)),('Nashville', (546.4, 336.8)),('Montgomery', (55
9.6, 404.8))]

##Optimal path:
Path_7 = [('Hartford', (719.6, 205.2)),('Albany', (702.0, 193.6)),('Boston', (738.4, 190.8
)),
          ('Providence', (735.2, 201.2)),('Richmond', (673.2, 293.6)),('Raleigh', (662.0, 3
28.8)),
          ('Columbia', (632.4, 364.8)),('Nashville', (546.4, 336.8)),('Atlanta', (585.6, 37
6.8)),
          ('Tallahassee', (594.8, 434.8)),('Montgomery', (559.6, 404.8)),('Jackson', (501.6
, 409.6)),
          ('Baton Rouge', (489.6, 442.0)),('Austin', (389.2, 448.4)),('Phoenix', (179.6, 37
1.2)),
          ('Sacramento', (68.4, 254.0)),('Salem', (80.0, 139.2)),('Boise', (159.6, 182.8)),
          ('Salt Lake City', (204.0, 243.2)),('Denver', (293.6, 274.0)),('Oklahoma City',(3
92.8, 356.4)),
          ('Little Rock', (469.2, 367.2)),('Des Moines', (447.6, 246.0)),('Saint Paul', (45
1.6, 186.0)),
          ('Madison', (500.8, 217.6)),('Lansing', (563.6, 216.4)),('Indianapolis', (548.0,
272.8)),
          ('Columbus', (590.8, 263.2)),('Harrisburg', (670.8, 244.0)),('Trenton', (698.8, 2
39.6))]

Path_8 = [('Baton Rouge', (489.6, 442.0)),('Austin', (389.2, 448.4)),('Oklahoma City', (39
2.8, 356.4)),
          ('Denver', (293.6, 274.0)),('Phoenix', (179.6, 371.2)),('Sacramento', (68.4, 254.
0)),
          ('Salem', (80.0, 139.2)),('Boise', (159.6, 182.8)),('Salt Lake City', (204.0, 24
3.2)),
          ('Saint Paul', (451.6, 186.0)),('Indianapolis', (548.0, 272.8)),('Columbus', (59
0.8, 263.2)),
          ('Trenton', (698.8, 239.6)),('Hartford', (719.6, 205.2)),('Providence', (735.2, 2
01.2)),
          ('Boston', (738.4, 190.8)),('Albany', (702.0, 193.6)),('Harrisburg', (670.8, 244.
0)),
          ('Richmond', (673.2, 293.6)),('Raleigh', (662.0, 328.8)),('Columbia', (632.4, 36
4.8)),
          ('Lansing', (563.6, 216.4)), ('Madison', (500.8, 217.6)),('Des Moines', (447.6, 2
46.0)),
          ('Little Rock', (469.2, 367.2)),('Nashville', (546.4, 336.8)),('Atlanta', (585.6,
376.8)),
          ('Montgomery', (559.6, 404.8)),('Tallahassee', (594.8, 434.8)),('Jackson', (501.6
, 409.6))]
```

6. References

[1] : Dan Fylstra et al., "EXAMPLES OF OPTIMIZATION PROBLEMS," Frontline Solvers, 2018. [Online]. Available: https://www.solver.com/examples-optimization-problems (https://www.solver.com/examples-optimization-problems).

[2] : S. Jameson et al., "The travelling salesman problem," in Edexcel AS and A level Further Mathematics Decision Mathematics 1, London, Pearson Education Limited, 2017, pp. 102-120.

[3] : A. Augustyn, "Heuristic reasoning," Britannica, 2008. [Online]. Available: https://www.britannica.com/topic/heuristic-reasoning (https://www.britannica.com/topic/heuristic-reasoning).

[4] : F. Schwandt et al., "Spending of international visitors to London from 2009 to 2019," Statista, 2020. [Online]. Available: https://www.statista.com/statistics/487536/expenditure-overseas-visitors-london-united-kingdom/ (https://www.statista.com/statistics/487536/expenditure-overseas-visitors-london-united-kingdom/).

[5] : "Tube," 2017. [Online]. Available: https://tfl.gov.uk/maps/track/tube (https://tfl.gov.uk/maps/track/tube).

[6] : G. Laporte, "The Tube Challenge," Information Systems and Operational Research, vol. 52, no. 1, pp. 10-13, 2014.

[7] : K. Y. Billah et al., "Resonance, Tacoma Narrows bridge failure and undergraduate physics textbooks," American Journal of Physics, vol. 59, no. 2, pp. 118-123, 1991.

[8] : M. Galley, "TACOMA NARROWS BRIDGE COLLAPSE," Think Reliability, 2008. [Online]. Available: https://www.thinkreliability.com/case_studies/tacoma-narrows-bridge-collapse/ (https://www.thinkreliability.com/case_studies/tacoma-narrows-bridge-collapse/).

[9] : Y. Gao, "Heuristic Algorithms for the Traveling Salesman Problem," Opex Analytics, 2020. [Online]. Available: https://medium.com/opex-analytics/heuristic-algorithms-for-the-traveling-salesman-problem-6a53d8143584 (https://medium.com/opex-analytics/heuristic-algorithms-for-the-traveling-salesman-problem-6a53d8143584).

[10] : S. Thomas, "Distance between TfL stations," WhatDoTheyKnow, 2017. [Online]. Available: https://www.whatdotheyknow.com/request/distance_between_tfl_stations (https://www.whatdotheyknow.com/request/distance_between_tfl_stations).

[11] : "The generalized traveling salesman problem solved with ant algorithms," Complex Adaptive Systems Modeling, vol. 5, no. 8, 2017.

[12] : A. Friedman, "Simulated annealing in protein folding," in Mathematics in Industrial Problems: Part 5, New York, Springer New York, 1992, pp. 78-87.

[13] : Y.-C. Huang et al., "Applications of Simulated Annealing-Based Approaches to Electric Power Systems," in Simulated Annealing - Advances, Applications and Hybridizations, 2012.

[14] : J. A. Rial et al., "Earthquake-Induced Resonance in Sedimentary Basins," American Scientist, vol. 80, no. 6, pp. 566--578, 1992.

[15] : F. &. Eckhard, "Crowd synchrony on the Millennium Bridge," Nature, vol. 43, no. 4, p. 438, 2005.

[16] : I. Belykh, "Foot force models of crowd dynamics on a wobbly bridge," Sscience Advances, vol. 3, no. 11, 2017.

In [ ]: