This project considers the Global Positioning System (GPS). To make things simple, we start with 4 satellites. At an instant time $t = d$, the receiver collects the synchronized signal from the satellites, which was sent at time t_i from the coordinate (A_i, B_i, C_i). To find out the receiver's position (x, y, z), we have the following system of equations,

$$\begin{cases} r_1 = \sqrt{(x-A_1)^2 + (y-B_1)^2 + (z-C_1)^2} = c(t_1 - d), \\ r_2 = \sqrt{(x-A_2)^2 + (y-B_2)^2 + (z-C_2)^2} = c(t_2 - d), \\ r_3 = \sqrt{(x-A_3)^2 + (y-B_3)^2 + (z-C_3)^2} = c(t_3 - d), \\ r_4 = \sqrt{(x-A_4)^2 + (y-B_4)^2 + (z-C_4)^2} = c(t_4 - d), \end{cases}$$

Here $c \approx 299792.458$ km/sec is the speed of light. Notice d should also be considered to be unknown because the receiver's time is inaccurate.

Setup -

First, I defined functions which would return a function to evaluate the following for given values of A_i, B_i, C_i, D_i:

$$f(x, y, z, d) = \sqrt{(x-A_i)^2 + (y-B_i)^2 + (z-C_i)^2} - c * (t_i - d)$$

$$f_x(x, y, z, d) = \frac{(x - A_i)}{sqrt(x-A_i)^2 + (y-B_i)^2 + (z-C_i)^2}$$

$$f_y(x, y, z, d) = \frac{(y - B_i)}{sqrt(x-A_i)^2 + (y-B_i)^2 + (z-C_i)^2}$$

$$f_z(x, y, z, d) = \frac{(z - C_i)}{sqrt(x-A_i)^2 + (y-B_i)^2 + (z-C_i)^2}$$

$$f_d(x, y, z, d) = c$$

The code for which is as follows:

```
# returns f with certain values of a, b, c, and t plugged in
def f(a: float, b: float, c: float, t: float, speed: float):
      def f_const(x: float, y: float, z: float, d: float):
      return sqrt((a - x)**2 + (b - y)**2 + (c - z)**2) - speed * (t - d)

      return f_const
```

```
# returns the partial derivative of f wrt x for a certain value of a, b, c,
and t
def fx(a: float, b: float, c: float, t: float, speed: float):
      def fx_const(x: float, y: float, z: float, d: float):
      return (x - a)/sqrt((a - x)**2 + (b - y)**2 + (c - z)**2)

      return fx_const


# returns the partial derivative of f wrt y for a certain value of a, b, c,
and t
def fy(a: float, b: float, c: float, t: float, speed: float):
      def fy_const(x: float, y: float, z: float, d: float):
      return (y - b)/sqrt((a - x)**2 + (b - y)**2 + (c - z)**2)

      return fy_const


# returns the partial derivative of f wrt z for a certain value of a, b, c,
and t
def fz(a: float, b: float, c: float, t: float, speed: float):
      def fz_const(x: float, y: float, z: float, d: float):
      return (z - c)/sqrt((a - x)**2 + (b - y)**2 + (c - z)**2)

      return fz_const

# returns the partial derivative of f wrt d for a certain value of a, b, c,
and t
def fd(a: float, b: float, c: float, t: float, speed: float):
      def fd_const(x: float, y: float, z: float, d: float):
      return speed

      return fd_const
```

Part I

Write a program to solve (x, y, z, d) with given (A_i, B_i, C_i, t_i), using Multivariable Newton's method. Notice, the system might have 2 solutions, one is near earth, one is far from earth. We only want the near earth solution. The earth center is fixed at (0, 0, 0).

I created a method to take in a starting guess, a system of functions, a system of derivatives of those functions, and a max number of iterations to run, and implemented multivariate Newton's method.

```
# does multivariable newton's method
def multivariable_newtons(start, funcs, derivs: np.array, maxits: int) ->
np.array:
      # initialize our starting point
      new_point = start.copy()
      current_f = np.zeros([len(funcs), 1])
      jacobian = np.zeros([len(funcs), len(derivs)])

      # delta = how much the iteration has changed
      # its = how many iterations are we on?
      delta = 1
      its = 0
      while delta > 1e-16 and its < maxits:
      # xi = xi-1
      current_point = new_point.copy()

      # calculate the jacobian and the values of f(x)
      for i in range(len(funcs)):
            for j in range(derivs.shape[1]):
                  # derivs[i][j] is holds the derivative of the ith
equation wrt the jth variable
                  # plug our current point into that
                  jacobian[i][j] = derivs[i][j](*current_point)
            # funcs[i] holds our ith function
            # plug current point into that
            current_f[i] = funcs[i](*current_point)

      # solve -J*h = f
      h = np.linalg.solve(-jacobian, current_f)

      # set our new point
      new_point = current_point + h

      # calculate the absolute distance of the norms between last iter and
this iter
      delta = fabs(np.linalg.norm(new_point) -
np.linalg.norm(current_point))
      its += 1

      # returns our best guess
      return new_point
```

This program, broadly, does the following:

- Set x_new = initial guess
- Loop until the change is small or we reach maxits iterations
    - Set x = x_new
    - Calculate $f(x) = f_1(x), f_2(x), \ldots, f_k(x)$
    - Calculate the Jacobian matrix, J, of $f_1, f_2, \ldots, f_k$ on x
    - Solve f(x) + J * h = 0 for h
    - Set x_new = x + h
    - Set delta = |norm(x_new) - norm(x)|
- Return x_new, our best answer

Part II

Given (15600, 7540, 20140, 0.07074), (18760, 2750, 18610, 0.07220), (17610, 14630, 13480, 0.07690), (19170, 610, 18390, 0.07242), as the position (in km) and time (in second) of the four satellites. Use the program in part (I) to find the receiver's position and time. Set the initial guess (0, 0, 6670, 0).

The data I got was:

```
PART A/B - INFO
The coordinates are:
P1 = (A1 =   5600.000, B1 =   7540.000, C1 =  20140.000, D1 = 0.07074)
P2 = (A2 =  18760.000, B2 =   2750.000, C2 =  18610.000, D2 = 0.07220)
P3 = (A3 =  17610.000, B3 =  14630.000, C3 =  13480.000, D3 = 0.07690)
P4 = (A4 =  19170.000, B4 =    610.000, C4 =  18390.000, D4 = 0.07242)

Distance Matrix:
1      0.000
2  14087.959      0.000
3  15455.219  12991.297      0.000
4  15337.285   2190.000  14936.603      0.000
           1          2          3          4

PART A/B - SOLUTION
The solution is:
x = 8783.51290
y = 1499.49768
z = 6102.17482
d =    0.01867

Results:
[0.]
[0.]
[0.]
[0.]
```

The Distance Matrix shows how far two points are from each other, ignoring the time value. I added this for extra analysis in part III.

The final solution I found to the system was:

(x = 8783.51290, y = 1499.49768, z = 6102.17482, d = 0.01867)

Plugging this back into all the equations, each evaluated to zero, making this an exact solution.

Part III

The clocks aboard the satellites are correct up to about 10^{-8} seconds. Change each t_i in Part (II) with $\pm 10^{-8}$, then the position as an output will be changed by $(\Delta x, \Delta y, \Delta z)$.
The distance $||(\Delta x, \Delta y, \Delta z)||_2$ tells the error estimate of the position found in Part (II).
Notice, some ti could be changed by 10^{-8}, some could be changed by -10^{-8}. Please try all possible combinations, and find the largest distance (error).

For this part, I developed the below function that provides all 2^n combinations of $\pm 10^{-8}$ for some arbitrary n, and used this to modulate the t_i's.

```
# generates 2^size unique combinations of size choices of +/-
def generate_binary(size: int) -> np.array:
      final = np.zeros(size)
      current = np.zeros(size)

      # we're going to have 2^n ways to form a string of n bits
      # iterate 2^n - 1 times since we already have a row of all zeros
      for version in range(0, 2**size-1):
      # increment the last bit by one
      current[-1] += 1

      # iterate over the size while making sure we don't have any 2's
      for i in range(1, size):

            # handle overflow
            if current[size - i] == 2:
                  current[size - i] = 0
                  current[size - i - 1] += 1

      # add the current bit pattern to our final array
      final = np.block([[final], [current]])
      return final
```

This final is a list of all possible binary strings of length size, which is then transformed by:

$$v_{i,new} = 2 * (v_i - 1) * 10^{-8}$$

To receive all combinations of 10^{-8} and -10^{-8}.

I plugged in all combination of variations from the original values and found that the largest error was had when satellites 1, 3, and 4 were modulated by -10^{-8}, and satellite 2 was modulated by 10^{-8}. The error from this combination was 0.09793590532841971 s, or about 9.79×10^{-2} seconds.

Distance Matrix:

1	0.000			
2	14087.959	0.000		
3	15455.219	12991.297	0.000	
4	15337.285	2190.000	14936.603	0.000
	1	2	3	4

Based on the distance matrix for these 4 satellites, I suspect this is because satellite 2 was generally closer to all the other satellites than they were to each other, so this small discrepancy was far more noticeable, since we'd expect satellite 2 to be getting roughly similar readings to the rest, especially satellite 4. But this combination of values meant that, instead, satellite 2 was getting the most dissimilar readings from the rest.

Part IV

In order to increase accuracy, we add another 4 satellites, so that we have 8 satellites in total. Design a program to solve the least square problem using Gauss–Newton's method.

Gauss-Newton's method is a small variation from Newton's multivariate method that handles the situation where there are m equations and n unknowns, where m > n. As such, it can be built out of the framework of Newton's multivariate method.

I implemented Gauss-Newton with the following code.

```
# gauss-newton method
# takes in start: our starting guess, funcs: our list of functions,
# derivs: a matrix of functions defining our partial derivatives, maxits:
how many iterations to go over at max
def gauss_newton(start, funcs, derivs: np.array, maxits: int) -> np.array:
    # set up
    new_point = start.copy()
    current_f = np.zeros([len(funcs), 1])
```

```
	jacobian = np.zeros([len(funcs), derivs.shape[1]])

	# delta = how much change since last iteration?
	delta = 1
	# its = how many iterations are we on?
	its = 0
	while delta > 1e-16 and its < maxits:
	# xi = xi-1
	current_point = new_point.copy()

	# calculate the jacobian and the current value of f(x)
	for i in range(len(funcs)):
		for j in range(derivs.shape[1]):
			# derivs[i][j] holds the derivative of the ith function
wrt the jth variable
			# plug in the current x
			jacobian[i][j] = derivs[i][j](*current_point)
		# funcs[i] holds the ith function
		# plug in the current x
		current_f[i] = funcs[i](*current_point)

	# for line spacing, store pseudo inverse = (JT J)^-1 JT
	pseudo_inverse = np.linalg.inv(jacobian.T @ jacobian) @ jacobian.T
	# xi+1 = xi - pseudo_inverse f(xi)
	new_point = current_point - pseudo_inverse @ current_f

	# how much have we changed?
	delta = fabs(np.linalg.norm(new_point) -
np.linalg.norm(current_point))
	its += 1

	return new_point
```

This is practically the same method as Newton's multivariate method, but instead of solving f(x) + Jh = 0, it instead calculates the left pseudoinverse of the Jacobian matrix to determine how much to change x by.

Part V

Arbitrarily choose the location (Ai, Bi, Ci) of the eight satellites on a sphere of radius 26570 km.
In order for the receiver at (0, 0, 6670) to receive the signal, we require $C_i > 6670$.
In order to have good accuracy, we do not want the satellites to be too close to each other.

The time t_i is given by:

$$t_i = \frac{\sqrt{A_i^2 + B_i^2 + (6670 - C_i)^2}}{c}$$

Now the system of eight equations and four unknowns have an exact solution (0, 0, 6670, 0). Estimate the error in position by changing each t_i by 10^{-8} second

To create my system, I randomly chose z, based on the factor that it had to be greater than 6670, but smaller than the radius. Then, I randomly generated y such that it could feasibly be a point on the given sphere with the generated z. And lastly, I solved for x since I had r, y, and z at this point.

This was implemented in the function below:

```
# generates an n points on a sphere of radius=radius, which have to have z
> min_height
def generate_points_on_sphere(radius: float, min_height: float, n: int) ->
np.array:
      points = np.array([0, 0, 0])

      # generates n points (x, y, z)
      for i in range(n):
      # z is at least min_height, but random otherwise
      z = min_height + random() * (radius - min_height)

      # y is a random value between 0 and r^2 - z^2
      random_fact = 2 * random() - 1
      y = random_fact/fabs(random_fact) *
sqrt((fabs(random_fact))*(radius**2 - z**2))

      # x we can solve for with sqrt(r^2 - y^2 - z^2)
      sign = random() - 0.5
      x = sign/fabs(sign) * sqrt(radius**2 - y**2 - z**2)

      # add to our list of points
      points = np.block([[points], [np.array([x, y, z])]])

      # return all but the initial (empty) value
      return points[1:]
```

I generated 8 points this way, resulting with the final coordinate system:

```
PART D/E - INFO
The coordinates are:
P1 = (A1 =  -8660.449, B1 =   8940.531, C1 =  23473.995, D1 = 0.06975)
P2 = (A2 = -23525.037, B2 =   3573.146, C2 =  11822.443, D2 = 0.08121)
P3 = (A3 = -11289.982, B3 =  -9091.511, C3 =  22267.592, D3 = 0.07103)
P4 = (A4 =   8266.386, B4 =  17421.219, C4 =  18279.303, D4 = 0.07508)
P5 = (A5 =  16466.808, B5 =  16853.658, C5 =  12278.573, D5 = 0.08079)
P6 = (A6 =  10144.539, B6 =  21615.097, C6 =  11655.076, D6 = 0.08136)
P7 = (A7 =  -6039.724, B7 =  12245.450, C7 =  22793.323, D7 = 0.07048)
P8 = (A8 =   7208.013, B8 =  14312.183, C8 =  21193.652, D8 = 0.07214)
```

I verified that these lead to an exact solution, and it did, so I kept on with the last part.

I followed the same procedure from part III to generate the new points and evaluated the error at each step. The maximum error I got was 0.012668867365909948 s, or roughly 1.267×10^{-2} s. This happened when satellites 2, 3, 5, and 6 were modulated by -10^{-8}s, and the rest were modulated by 10^{-8}s.

Looking at the distance matrix of these 8 satellites:

```
Distance Matrix:
1      0.000
2  19634.751      0.000
3  18262.649  20474.160      0.000
4  19632.235  35272.577  33185.581      0.000
5  28624.013  42141.757  39285.769  10177.318      0.000
6  25572.623  38199.219  38922.486   8062.038   7939.215      0.000
7   4272.471  22389.846  21979.706  15869.145  25265.382  21766.516      0.000
8  16907.474  33877.231  29850.662   4390.864  13102.023  12366.898  13503.069      0.000
           1          2          3          4          5          6          7          8
```

The connection between proximity and error-sensitivity is far messier here. In many cases, the nearest neighbor is of the same shift. However, one notable change is that the satellites which got modulated with a negative shift are often very spread out from one another. They cover more ground this way and are thus less affected by small shifts in the timing, while the nearer satellites are far more affected.