Project 1: GPS POSITIONING

Group members: Shilpa Kuppili , Sathwik Kuchana,
Praveen Kumar Govind Reddy, Andria Grace, Jayanth Vodnala

Introduction:

The global positioning system or GPS is a constellation of satellites that are used to approximate the location of a GPS reciever on Earth. GPS uses 4 satellites that are in view of the reciever to solve 4 equations for the (x,y,z) coordinates of the reciever and a value d which is the difference in time between the reciever's clock and the satellites' clocks. The difference in time is important because the location is approximated using the travel time of a signal from the satellite to the reciever, and this happens in much less than a second.

The equations that are solved to approximate a reciever's location using GPS are:

$(x - A1)\ 2 + (y - B1)\ 2 + (z - C1)\ 2 = [c(t1 - d)]2$
$(x - A2)\ 2 + (y - B2)\ 2 + (z - C2)\ 2 = [c(t2 - d)]2$
$(x - A3)\ 2 + (y - B3)\ 2 + (z - C3)\ 2 = [c(t3 - d)]2$
$(x - A4)\ 2 + (y - B4)\ 2 + (z - C4)\ 2 = [c(t4 - d)]2$

Solution 1:

The receiver position (x, y, z) and time correction d for simultaneous satellite positions (Ai , Bi , Ci) are [-4.17727096e+01 -1.67891941e+01 6.37005956e+03] and -3.20156583e-03 respectively

Code:

```
# problem 1

import numpy as np


def receiver_position(x, y, z, d):
    # Sort the initial position
    u = np.array([x, y, z, d])

    # Set number of iterations
    n = 10

    # Initial positions and time of satellites that are given
    A1, B1, C1 = 15600, 7540, 20140
    A2, B2, C2 = 18760, 2750, 18610
    A3, B3, C3 = 17610, 14630, 13480
    A4, B4, C4 = 19170, 610, 18390

    t = np.array([0.07074, 0.07220, 0.07690, 0.07242])
```

```
    # Speed of light
    c = 299792.458

    x0, y0, z0, d0 = x, y, z, d

    for i in range(n):
        # Find F
        F = np.array([(x0 - A1)**2 + (y0 - B1)**2 + (z0 - C1)**2 -
(c*(t[0] - d0))**2,
                      (x0 - A2)**2 + (y0 - B2)**2 + (z0 - C2)**2 -
(c*(t[1] - d0))**2,
                      (x0 - A3)**2 + (y0 - B3)**2 + (z0 - C3)**2 -
(c*(t[2] - d0))**2,
                      (x0 - A4)**2 + (y0 - B4)**2 + (z0 - C4)**2 -
(c*(t[3] - d0))**2])

        # Find dF
        dF = np.array([[2*(x0 - A1), 2*(y0 - B1), 2*(z0 - C1),
2*(c**2*(t[0] - d0))],
                       [2*(x0 - A2), 2*(y0 - B2), 2*(z0 - C2),
2*(c**2*(t[1] - d0))],
                       [2*(x0 - A3), 2*(y0 - B3), 2*(z0 - C3),
2*(c**2*(t[2] - d0))],
                       [2*(x0 - A4), 2*(y0 - B4), 2*(z0 - C4),
2*(c**2*(t[3] - d0))]])

        # Apply Newton Method to find next position


        v = np.linalg.solve(dF, -F)

        u += v

        x0, y0, z0, d0 = u[0], u[1], u[2], u[3]

    return u

p = receiver_position(0.0,0.0,6370.0,0.0)
print(p)
```

Importing NumPy: Import the NumPy library, which is commonly used for numerical and matrix operations.
The receiver_position function takes four parameters: x, y, z, and d, which represent the initial estimates of the receiver's position in the x, y, z coordinates, and the clock bias d.

Initializing Variables: u is initialized as a NumPy array containing the initial position estimates (x, y, z, d).n is set to 10, which represents the number of iterations for the Newton-Raphson method.
Satellite Data: defines the positions of four satellites (A1, B1, C1, A2, B2, C2, A3, B3, C3, A4, B4, C4) in space, as well as the time of signal transmission from each satellite (t).
Speed of Light:
The speed of light c is defined as 299,792.458 km/s.

Iterative Position Estimation: The code enters a loop that will run for n iterations. In each iteration, it attempts to refine the receiver's position estimate using the Newton-Raphson method.
F and dF Calculation: Inside the loop, it calculates two important values:

F is a NumPy array containing four equations, each of which represents the difference between the estimated distance between the receiver and a satellite and the actual distance (calculated from the satellite's transmission time and the speed of light).

dF is a 4x4 NumPy array representing the Jacobian matrix, which contains partial derivatives of the four equations with respect to the receiver's coordinates (x, y, z) and clock bias d.
Newton-Raphson Step: It solves a system of linear equations using np.linalg.solve to find v, a 4x1 NumPy array. v contains the updates that need to be applied to the current position estimate to minimize the error represented by F.

The updated position estimate u is then obtained by adding v to the previous estimate.

Updating Initial Estimates: The x0, y0, z0, and d0 variables are updated with the new values from u.

After n iterations, the function returns the final estimate of the receiver's position in the form of a NumPy array u.

Lastly, the function is called with initial values (receiver's position) and the estimated receiver position is printed.

In conclusion, the code implements an iterative method to estimate the position (x, y, z) of a receiver and its clock bias (d) based on the time signals received from four satellites and the speed of light. The Newton-Raphson method is used to iteratively improve the position estimate until convergence.

Solution 2:

The error magnification factor: 2124801.1129628475
The maximum position error found: 6370.000087427847
The condition number of the problem is estimated as 2124802.9929628475

Code:

```
# Problem 2

import numpy as np

def getSphericalCoordinates(phi, theta, ρ=26570):
    x = ρ * np.cos(phi) * np.cos(theta)
```

```
    y = ρ * ρ * np.cos(phi) * np.sin(theta)
    z = ρ * np.sin(phi)
    return np.array([x, y, z])

def calculateRange(satellitePosition):
    x_diff = satellitePosition[0]
    y_diff = satellitePosition[1]
    z_diff = satellitePosition[2] - 6370
    return np.sqrt(x_diff**2 + y_diff**2 + z_diff**2)

def quadratic_formula(travel_times, satellitePositions, param, x, y,
z):
    A = np.zeros((len(satellitePositions), 4))
    for i, satPos in enumerate(satellitePositions):
        A[i, 0] = 2 * (satPos[0] - x)
        A[i, 1] = 2 * (satPos[1] - y)
        A[i, 2] = 2 * (satPos[2] - z)
        A[i, 3] = 2 * 299792.458 * (0.0001 - travel_times[i])

    b = np.zeros(len(satellitePositions))
    for i, satPos in enumerate(satellitePositions):
        b[i] = (satPos[0] - x)**2 + (satPos[1] - y)**2 + (satPos[2] -
z)**2 - (299792.458 * (0.0001 - travel_times[i]))**2

    x_solution = np.linalg.lstsq(A, b, rcond=None)[0]

    return x_solution[param - 1]

def generateOnes(numOfSatellites):
    return np.ones((2**numOfSatellites, numOfSatellites))

def error_magnification(numOfSatellites, theta, phi,infty_norm_t):
    x = 0
    y = 0
    z = 6370

    satellitePositions = np.zeros((numOfSatellites, 3))

    for i in range(numOfSatellites):
        mytheta = theta[i]
        myphi = phi[i]
        satellitePositions[i, :] = getSphericalCoordinates(myphi,
mytheta)

    Ri_vector = np.zeros(numOfSatellites)
    travel_times = np.zeros(numOfSatellites)

    for i in range(numOfSatellites):
```

```
        Ri_vector[i] = calculateRange(satellitePositions[i])
        travel_times[i] = 0.0001 + Ri_vector[i] / 299792.458

    travel_times_change = generateOnes(numOfSatellites) * 0.0000001

    travel_times_row_vector = travel_times

    conditionNumber = 1

    for row in range(2**numOfSatellites):
        travel_times_my_row = (travel_times_row_vector +
travel_times_change[row, :])
        x_bar = quadratic_formula(travel_times_my_row,
satellitePositions, 1, x, y, z)
        y_bar = quadratic_formula(travel_times_my_row,
satellitePositions, 2, x, y, z)
        z_bar = quadratic_formula(travel_times_my_row,
satellitePositions, 3, x, y, z)
        d_bar = quadratic_formula(travel_times_my_row,
satellitePositions, 4, x, y, z)

        delta_x = abs(x - x_bar)
        delta_y = abs(y - y_bar)
        delta_z = abs(z - z_bar)

        infty_norm_xyz = max(delta_x, delta_y, delta_z)

        emf = infty_norm_xyz / (infty_norm_t * 299792.458)

        if row == 0:
            conditionNumber = emf
        else:
            if emf > conditionNumber:
                conditionNumber = emf
    print("emf =",emf )
    print("position error found =",infty_norm_xyz)
    return conditionNumber

# Example usage

numOfSatellites = 4
theta = [0,(np.pi/3), ((np.pi)*2)/3, 2* (np.pi)]
phi = [0,(np.pi)/6, (np.pi)/3, (np.pi)/2]

conditionNumber = error_magnification(numOfSatellites, theta,
phi,0.00000001)
theConditionNumber = conditionNumber
travel_times_change = generateOnes(numOfSatellites) * 0.00000001
```

```
theGeneratingTrow =
travel_times_change[np.argmax([error_magnification(numOfSatellites,
theta, phi + (np.arange(numOfSatellites) - 0.5) *
0.00000001,0.00000001) for phi in phi])]
positionMinusEMF = abs(theConditionNumber - (theConditionNumber *
299792.458 * theGeneratingTrow).max() / 0.00000001)
print("theConditionNumber =", theConditionNumber)
```

The getSphericalCoordinates function converts spherical coordinates (ϕ, θ, ρ) to Cartesian coordinates (x, y, z).
ϕ represents the polar angle (elevation) in radians.
θ represents the azimuthal angle (longitude) in radians.
ρ is the radial distance from the origin (default value is 26570 units).

The calculateRange function calculates the Euclidean distance (range) between a satellite's position and the Earth's surface (assumed to be a sphere with a radius of 6370 units).
Further, estimate the position coordinates (x, y, z) and clock bias (d) based on satellite measurements.

This generateOnes function generates a matrix of ones with dimensions (2^numOfSatellites, numOfSatellites).

The error_magnification function calculates the error magnification factor (EMF) for a given configuration of satellites.
It takes as input the number of satellites, arrays of ϕ, θ, and a value for the infinity norm (infty_norm_t).
It iteratively evaluates the EMF for different satellite configurations and determines the maximum EMF.
It defines the number of satellites (numOfSatellites) and arrays of ϕ and θ coordinates for the satellites.
It calculates the EMF for the initial configuration of satellites.
It then iterates over different configurations by slightly changing the ϕ coordinates for the satellites and calculates the EMF for each configuration.
It finds the configuration with the highest EMF and stores it in theGeneratingTrow.
Finally, it calculates positionMinusEMF, which represents the difference between the EMF and the maximum EMF for the configuration with slightly changed ϕ coordinates.
It performs an error analysis to understand how changes in the positions of the satellites affect the error magnification in the estimation of the receiver's position. The code iterates over different satellite configurations to find the worst-case scenario where the error magnification is maximized.

Solution 3:

The maximum position with error : 8658.804892131084
The Condition Number = 2125736.9315818413
The error magnification factor with error : 2046357.126836302

The maximum position without error : 6729.058026236874
The Condition Number = inf
The error magnification factor without error : inf

The conditioning of the GPS problem when the satellites are tightly or loosely bunched: We found that solving for (x, y,z,d) is ill-conditioned when the satellites are bunched closely in the sky. Maximum position error is high when satellites are tightly bunched in the sky when compared to maximum position error of loosely bunched satellites.

Code:

```
# Problem 3

import numpy as np


# Example usage
numOfSatellites = 4
theta = [0.1,0.105, 0.110, 0.115]
phi = [0.25,0.2625, 0.275, 0.2875]

# with error
print("with error")
conditionNumber = error_magnification(numOfSatellites, theta,
phi,0.00000001)
theConditionNumber = conditionNumber

travel_times_change = generateOnes(numOfSatellites) * 0.00000001
theGeneratingTrow =
travel_times_change[np.argmax([error_magnification(numOfSatellites,
theta, phi + (np.arange(numOfSatellites) - 0.5) *
0.00000001,0.00000001) for phi in phi])]
positionMinusEMF = abs(theConditionNumber - (theConditionNumber *
299792.458 * theGeneratingTrow).max() / 0.00000001)
print("theConditionNumber =", theConditionNumber)


print("\nwithout error")

conditionNumber = error_magnification(numOfSatellites, theta, phi,0)
theConditionNumber = conditionNumber

travel_times_change = generateOnes(numOfSatellites) * 0
theGeneratingTrow =
travel_times_change[np.argmax([error_magnification(numOfSatellites,
theta, phi + (np.arange(numOfSatellites) - 0.5) * 0,0) for phi in
phi])]
positionMinusEMF = abs(theConditionNumber - (theConditionNumber *
299792.458 * theGeneratingTrow).max() / 0)
print("theConditionNumber =", theConditionNumber)
```

It is specified here, the number of satellites (numOfSatellites) and two lists: theta and phi. These lists contain the spherical coordinates for the satellites. Each list contains four values.

Two scenarios are examined: one with error (non-zero travel time changes) and one without error (zero travel time changes).
"With Error" Scenario:
The EMF for the initial configuration is calculated and stored in theConditionNumber.
The code iterates over different configurations by slightly changing the phi coordinates for the satellites and calculates the EMF for each configuration.
It finds the configuration with the highest EMF and stores it in theGeneratingTrow.
Finally, it calculates positionMinusEMF, which represents the difference between the EMF and the maximum EMF for the configuration with slightly changed phi coordinates.
"Without Error" Scenario:
The EMF for the initial configuration is calculated and stored in theConditionNumber.
Similar to the "With Error" scenario, the code iterates over different configurations by slightly changing the phi coordinates for the satellites and calculates the EMF for each configuration.
It finds the configuration with the highest EMF and stores it in theGeneratingTrow.
Finally, it calculates positionMinusEMF for this scenario.
We compare the EMF in two scenarios: one with measurement errors (non-zero travel time changes) and one without errors (zero travel time changes).We examine how different configurations of satellite positions affect the EMF in both scenarios.