2025

Project 2: GPS positioning

STUDENT ID: 800777831

GREGORY E. SCHWARTZ

Abstract

This report addresses two primary tasks related to GPS positioning in Project 2. In Task 1, we solve a system of nonlinear equations—derived from satellite positions, transmission times, and clock drift—using Newton's method. The solver converged in 2 iterations to a receiver solution of approximately $[-\mathbf{41.77},\ -\mathbf{16.79},\ \mathbf{6370.06},\ -\mathbf{0.00320}]$, with residuals on the order of $\mathbf{10^{-8}}$. In Task 2, we analyze the conditioning of the GPS problem by generating satellite positions from spherical coordinates, computing baseline travel times, and perturbing these travel times by $\pm\mathbf{10^{-8}}$s. The perturbation analysis revealed a maximum position error of **0.00313** km (**3.13** m) and an estimated error magnification factor (condition number) of approximately **1.044**, indicating the system is well-conditioned.

Introduction

The Global Positioning System (GPS) determines a receiver's location by solving nonlinear equations based on known satellite positions and precise timing. In this project, two aspects of the problem are explored. First, a Newton-based solver is implemented to compute the receiver's position and clock bias from four satellite signals. Second, the sensitivity of the computed solution to small perturbations in the transmission times is examined to assess the system's conditioning. These investigations are crucial for understanding both the accuracy and stability of GPS positioning. All computations are performed in Python using NumPy, and our approach extends the professor's provided Newton solver with GPS-specific equations and a conditioning analysis framework.

Body of the Project and the Sections

Task 1: Solving the Nonlinear System

The GPS positioning problem is modeled by the equations

$$\sqrt{(x-A_i)^2+(y-B_i)^2+(z-C_i)^2} = c\,(t_i - d) \quad (i = 1, \dots, 4)$$

where $(A_i,\ B_i,\ C_i)$ are the satellite coordinates, t_i are the measured transmission times, $c\ =\ 299792.458$ km/s is the speed of light, and d is the receiver's clock bias. Using the professor's Newton solver, the initial guess was set to

$$(x, y, z, d) = (0,0,6370,0)$$

and the system was solved iteratively. The resulting solution was

$$[-41.77,\ -16.79,\ 6370.06,\ -0.00320],$$

indicating a receiver position at approximately $(-41.77,\ -16.79,\ 6370.06)$ km with a clock bias of -0.00320 s. The solver converged in 2 iterations, with residuals on the order of 10^{-8}, confirming that the solution accurately satisfies the nonlinear equations.

Task 2: Conditioning Analysis

For the conditioning analysis, satellite positions were generated using spherical coordinates as follows:

$$A_i = \rho \cos\phi_i \cos\theta_i,$$
$$B_i = \rho \cos\phi_i \sin\theta_i,$$
$$C_i = \rho \sin\phi_i,$$
$$\rho = 26570\,\text{km}, \quad \phi_i = \frac{\pi}{6}, \quad \theta_i = 0, \frac{\pi}{2}, \pi, \frac{3\pi}{2}$$

With $\rho = 26570$ km. To ensure the satellites lie in the upper hemisphere, we chose $\phi i = \pi/6$ for all satellites and $\theta_i = 0, \frac{\pi}{2}, \pi, \frac{3\pi}{2}$ respectively. The receiver is assumed to be at $(0,0,6370)$ km with a clock bias $d = 0.0001$ s. Baseline travel times are computed as

$$t_i = d + \frac{R_i}{c} \quad \text{with} \quad R_i = \sqrt{A_i^2 + B_i^2 + (C_i - 6370)^2}$$

The baseline solution was verified using the Newton solver. Next, each t_i was perturbed by $\pm 10^{-8}$ (covering all 16 combinations), and the system was re-solved using a modified Newton method that employs a pseudo-inverse to handle singular Jacobians. For each perturbation, the forward error

$$\|\Delta(x, y, z)\|_\infty$$

was computed as the maximum absolute difference between the perturbed and baseline receiver positions. The input error is calculated as

$$c\,\|\Delta t\|_\infty$$

where a perturbation of 10^{-8} s corresponds to approximately 0.003 km (or 3 m). The error magnification factor (condition number) is given by

$$\frac{\|\Delta(x, y, z)\|_\infty}{c\,\|\Delta t\|_\infty}$$

Our experiments yielded a maximum position error of 0.00313 km (3.13 m) and an estimated condition number of approximately 1.044, indicating that the system is well-conditioned; small timing errors are amplified by only about 4.4%.

Methodological Decisions

To overcome issues with singular Jacobians during the perturbation analysis, the Newton update was modified to use the pseudo-inverse. This ensures robust computation of the update even when the Jacobian is nearly singular. All parameter values (such as the speed of light, receiver position, and clock bias) were maintained as specified in the assignment.

Computer Experiments/Simulations and Results

The experiments were implemented in Python using NumPy and itertools. For Task 1, the Newton solver converged in 2 iterations with a tolerance of 1×10^{-4}, yielding a solution that accurately recovered the receiver's position and clock bias. For Task 2, perturbing the travel times by $\pm 10^{-8}$ s led to a maximum position error of **0.00313 km (3.13 m)** and an error magnification factor of approximately **1.044**. These results demonstrate that the GPS system is well-conditioned; that is, the system's response to small input errors in transmission times is nearly proportional to the error itself.

**See the appendix for the calculations*

Conclusions

This project demonstrates an effective application of Newton's method for solving a GPS positioning problem and provides a thorough sensitivity analysis via conditioning. Task 1 successfully computed the receiver's position and clock bias with high accuracy, as evidenced by the minimal residuals and rapid convergence. Task 2's perturbation analysis revealed that the system is well-conditioned, with an error magnification factor of roughly **1.044**. The use of a pseudo-inverse in the Newton solver ensured robustness against singularities in the Jacobian. Future work could explore alternative approaches such as least-squares formulations or adaptive damping techniques to improve stability under varying conditions further.

References

- Sauer, T. (n.d.). *Numerical Analysis 3e*. [Textbook].
- Professors provided Newton solver code.
- NumPy Documentation, https://numpy.org/doc/

- Moore–Penrose Pseudoinverse, https://en.wikipedia.org/wiki/Moore%E2%80%93Penrose_inverse
- Condition Number, https://en.wikipedia.org/wiki/Condition_number
- Class lecture and class notes.

Appendix

Task 1 solution

March 4, 2025

[3]:
```python
import numpy as np

def Newton_system(F, DF, x0, N, eps):
    """
    Solve F(x) = 0 using Newton's method.

    Inputs:
      F: Function returning the residual vector F(x)
      DF: Function returning the Jacobian matrix DF(x)
      x0: Initial guess (vector)
      N: Maximum number of iterations
      eps: Convergence tolerance (stop when ||F(x)||_2 < eps)

    Output:
      x: Approximate solution vector
      steps: Number of steps taken (or -1 if convergence was not achieved)
    """
    x = x0.copy()             # Start with a copy of the initial guess
    F_val = F(x)              # Evaluate F at the current guess
    F_norm = np.linalg.norm(F_val, ord=2)  # Compute the Euclidean norm of F(x)
    steps = 0
    while F_norm > eps and steps < N:
        # Solve the linear system: DF(x)*s = F(x)
        s = np.linalg.solve(DF(x), F_val)
        x = x - s             # Update the solution: x_new = x - s
        F_val = F(x)          # Recompute the residual vector at the new guess
        F_norm = np.linalg.norm(F_val, ord=2)  # Update the norm of F(x)
        steps += 1
    if F_norm > eps:
        steps = -1            # Indicate failure to converge if tolerance not met
    return x, steps

def F_GPS_params(x, A, B, C, t, c):
    """
    Computes the residuals for the GPS system.

    For each satellite i:
```

```
      F_i(x,y,z,d) = sqrt((x - A_i)^2 + (y - B_i)^2 + (z - C_i)^2) - c*(t_i - d)

    x: Vector [x, y, z, d]
    A, B, C: Arrays of satellite positions (km)
    t: Array of travel times (seconds)
    c: Speed of light (km/s)

    Returns a 4-element vector of residuals.
    """
    distances = np.sqrt((x[0] - A)**2 + (x[1] - B)**2 + (x[2] - C)**2)
    return distances - c * (t - x[3])

def DF_GPS_params(x, A, B, C, c):
    """
    Computes the 4x4 Jacobian matrix for the GPS system.

    For each satellite i:
       F_i/ x = (x - A_i)/r_i,
       F_i/ y = (y - B_i)/r_i,
       F_i/ z = (z - C_i)/r_i,
       F_i/ d = c,
    where r_i = sqrt((x - A_i)^2 + (y - B_i)^2 + (z - C_i)^2).
    """
    J = np.zeros((4, 4))
    for i in range(4):
        r_i = np.sqrt((x[0] - A[i])**2 + (x[1] - B[i])**2 + (x[2] - C[i])**2)
        J[i, 0] = (x[0] - A[i]) / r_i
        J[i, 1] = (x[1] - B[i]) / r_i
        J[i, 2] = (x[2] - C[i]) / r_i
        J[i, 3] = c
    return J

# Given satellite positions (km) as provided in the assignment:
A = np.array([15600, 18760, 17610, 19170])
B = np.array([7540, 2750, 14630, 610])
C = np.array([20140, 18610, 13480, 18390])
# Given measured travel times (seconds):
t = np.array([0.07074, 0.07220, 0.07690, 0.07242])
c = 299792.458  # km/s

# Baseline receiver parameters: (x, y, z, d)
# Receiver at (0, 0, 6370) km, clock bias d = 0 s (per assignment, initial␣
 ↪guess is d=0)
x0 = np.array([0.0, 0.0, 6370.0, 0.0])

solution, steps = Newton_system(
    lambda x: F_GPS_params(x, A, B, C, t, c),
```

```
    lambda x: DF_GPS_params(x, A, B, C, c),
    x0, 100, 1e-4)

print("Solution (Task 1):", solution)
print(f"x ≈ {solution[0]:.2f} km, y ≈ {solution[1]:.2f} km, z ≈ {solution[2]:.
  ↪2f} km, d ≈ {solution[3]:.5f} s")
print("Number of steps:", steps)
print("Residual:", F_GPS_params(solution, A, B, C, t, c))
```

```
Solution (Task 1): [-4.17727094e+01 -1.67891940e+01  6.37005956e+03
-3.20156583e-03]
x ≈ -41.77 km, y ≈ -16.79 km, z ≈ 6370.06 km, d ≈ -0.00320 s
Number of steps: 2
Residual: [5.22850314e-08 4.41614247e-08 2.23371899e-08 4.76356945e-08]
```

Task 2 solution

March 4, 2025

[34]:
```python
###Solving Task Two od Project 2 ###
import numpy as np
import itertools

def Newton_system(F, DF, x0, N, eps):
    """
    Solve F(x) = 0 using Newton's method with a pseudo-inverse to handle␣
 ↪singular
    Jacobians that occurred when calculated not using a pseudo-inverse

    Inputs:
      F: Function returning the residual vector F(x)
      DF: Function returning the Jacobian matrix DF(x)
      x0: Initial guess (vector)
      N: Maximum number of iterations
      eps: Convergence tolerance (stop when ||F(x)||_2 < eps)

    Output:
      x: Approximate solution vector
      steps: Number of steps taken (or -1 if convergence was not achieved)
    """
    x = x0.copy()
    F_val = F(x)
    F_norm = np.linalg.norm(F_val, ord=2)
    steps = 0
    while F_norm > eps and steps < N:
        # Instead of using np.linalg.solve (which fails if the Jacobian is␣
 ↪singular),
        # we use the pseudo-inverse (np.linalg.pinv) to compute an update.
        s = np.dot(np.linalg.pinv(DF(x)), F_val)
        x = x - s
        F_val = F(x)
        F_norm = np.linalg.norm(F_val, ord=2)
        steps += 1
    if F_norm > eps:
        steps = -1
    return x, steps
```

[35]:
```
def satellite_positions_spherical(rho, phi, theta):
    """
    Given spherical coordinates:
      A = rho * cos(phi) * cos(theta)
      B = rho * cos(phi) * sin(theta)
      C = rho * sin(phi)
    Returns (A, B, C) for a single satellite.
    """
    A = rho * np.cos(phi) * np.cos(theta)
    B = rho * np.cos(phi) * np.sin(theta)
    C = rho * np.sin(phi)
    return A, B, C

# Parameters for satellite positions:
rho = 26570  # km (given)
# Choose phi values in [0, pi/2] (e.g., pi/6 for all to ensure they are in the␣
 ↪upper hemisphere)
phi_vals = np.array([np.pi/6, np.pi/6, np.pi/6, np.pi/6])
# Choose theta values arbitrarily in [0, 2pi]: 0, pi/2, pi, 3pi/2.
theta_vals = np.array([0, np.pi/2, np.pi, 3*np.pi/2])

A = np.zeros(4)
B = np.zeros(4)
C = np.zeros(4)
for i in range(4):
    A[i], B[i], C[i] = satellite_positions_spherical(rho, phi_vals[i],␣
 ↪theta_vals[i])
```

[36]:
```
def F_GPS_params(x, A, B, C, t, c):
    """
    Computes the residuals for the GPS system.

    For each satellite i:
      F_i(x,y,z,d) = sqrt((x - A_i)^2 + (y - B_i)^2 + (z - C_i)^2) - c*(t_i - d)

    x: Vector [x, y, z, d]
    A, B, C: Arrays of satellite positions
    t: Array of travel times (can be baseline or perturbed)
    c: Speed of light (km/s)

    Returns a 4-element vector of residuals.
    """
    distances = np.sqrt((x[0] - A)**2 + (x[1] - B)**2 + (x[2] - C)**2)
    return distances - c * (t - x[3])

def DF_GPS_params(x, A, B, C, c):
    """
```

```
    Computes the 4x4 Jacobian matrix for the GPS system.

    For each satellite i:
       F_i/x = (x - A_i)/r_i,
       F_i/y = (y - B_i)/r_i,
       F_i/z = (z - C_i)/r_i,
       F_i/d = c,
    where r_i = sqrt((x-A_i)^2 + (x[1]-B_i)^2 + (x[2]-C_i)^2).
    """
    J = np.zeros((4, 4))
    for i in range(4):
        r_i = np.sqrt((x[0] - A[i])**2 + (x[1] - B[i])**2 + (x[2] - C[i])**2)
        J[i, 0] = (x[0] - A[i]) / r_i
        J[i, 1] = (x[1] - B[i]) / r_i
        J[i, 2] = (x[2] - C[i]) / r_i
        J[i, 3] = c
    return J
```

[37]:
```
# Baseline receiver: (x, y, z, d) = (0, 0, 6370, 0.0001)
x_true = np.array([0.0, 0.0, 6370.0, 0.0001])
c = 299792.458  # km/s

# Compute the satellite range for each satellite:
R = np.sqrt(A**2 + B**2 + (C - 6370)**2)
# Compute baseline travel times: t = d + R/c
t_baseline = x_true[3] + R/c
```

[38]:
```
# Use the baseline as the initial guess.
x0 = x_true.copy()

solution_baseline, steps_baseline = Newton_system(
    lambda x: F_GPS_params(x, A, B, C, t_baseline, c),
    lambda x: DF_GPS_params(x, A, B, C, c),
    x0, 100, 1e-4)

print("Baseline solution:", solution_baseline)
print("Baseline residual:", F_GPS_params(solution_baseline, A, B, C,␣
 ↪t_baseline, c))
```

```
Baseline solution: [0.00e+00 0.00e+00 6.37e+03 1.00e-04]
Baseline residual: [0. 0. 0. 0.]
```

[39]:
```
# We will perturb each travel time by ±10^-8 seconds.
perturbations = [-1e-8, 1e-8]
max_pos_error = 0
max_error_magnification = 0

# Loop over all 16 combinations of perturbations for the 4 satellites.
```

```
for deltas in itertools.product(perturbations, repeat=4):
    delta_t = np.array(deltas)
    t_perturbed = t_baseline + delta_t
    solution_perturbed, steps_perturbed = Newton_system(
        lambda x: F_GPS_params(x, A, B, C, t_perturbed, c),
        lambda x: DF_GPS_params(x, A, B, C, c),
        x0, 100, 1e-4)
    # Compute the forward error: difference in (x, y, z) compared to the␣
 ↪baseline (x_true)
    pos_error = np.linalg.norm(solution_perturbed[0:3] - x_true[0:3], ord=np.
 ↪inf)
    # Compute the input error: c * ||Δt||∞ (Δt in seconds, converted to␣
 ↪distance)
    input_error = c * np.linalg.norm(delta_t, ord=np.inf)
    error_magnification = pos_error / input_error if input_error != 0 else 0
    if pos_error > max_pos_error:
        max_pos_error = pos_error
    if error_magnification > max_error_magnification:
        max_error_magnification = error_magnification

print("Maximum position error (KM):", max_pos_error)
max_pos_error_meters = max_pos_error * 1000
print("Maximum position error (Meters):", max_pos_error_meters)
print("Estimated condition number (error magnification factor):",␣
 ↪max_error_magnification)
```

```
Maximum position error (KM): 0.0031303716345498778
Maximum position error (Meters): 3.1303716345498778
Estimated condition number (error magnification factor): 1.0441795819125903
```