```
"""
Theoretical Orbit Determination Calculations

This script calculates the answers to 3 theoretical questions about
GNSS positioning effects for a LEO satellite receiver.
"""

import numpy as np

# ============================================================================
# CONSTANTS
# ============================================================================
c = 299792.458  # Speed of light [km/s]
omega_e = 7.292115e-5  # Earth rotation rate [rad/s]
mu_earth = 398600.4418  # Earth gravitational parameter [km^3/s^2]
Re = 6378.137  # Earth radius [km]

# ============================================================================
# SCENARIO PARAMETERS
# ============================================================================
# LEO satellite receiver
h_leo = 500  # Altitude [km]
r_leo = Re + h_leo  # Orbital radius [km]

# GPS satellites
a_gps = 26560  # Semi-major axis [km]
e_gps = 0.01  # Eccentricity

# Clock error
delta_t_clk = 0.1e-3  # Transmitter clock error [s] (0.1 ms)

print("="*70)
print("THEORETICAL ORBIT DETERMINATION CALCULATIONS")
print("="*70)
print("\nCONSTANTS:")
print(f"  Speed of light (c):           {c} km/s")
print(f"  Earth rotation rate (Ï\211_e):    {omega_e} rad/s")
print(f"  Earth gravity (Î¼):            {mu_earth} kmÂ³/sÂ²")
print(f"  Earth radius (R_e):           {Re} km")
print("\nSCENARIO:")
print(f"  LEO altitude:                 {h_leo} km")
print(f"  LEO orbital radius:           {r_leo} km")
print(f"  GPS semi-major axis:          {a_gps} km")
print(f"  GPS eccentricity:             {e_gps}")
print(f"  Transmitter clock error:      {delta_t_clk*1e3} ms")
print("="*70)

# ============================================================================
# QUESTION 1: CLOCK OFFSET EFFECT
# ============================================================================
print("\n" + "="*70)
print("QUESTION 1: CLOCK OFFSET EFFECT")
print("="*70)

# Range error due to clock error
Delta_rho_clock = c * delta_t_clk  # [km]

print(f"\nFormula: Î\224Ï\201 = c Ã\227 Î´t")
print(f"\nCalculation:")
print(f"  Î\224Ï\201 = {c} km/s Ã\227 {delta_t_clk*1e3} ms")
print(f"  Î\224Ï\201 = {c} km/s Ã\227 {delta_t_clk} s")
print(f"  Î\224Ï\201 = {Delta_rho_clock} km")
print(f"\n>>> ANSWER: Range error = {Delta_rho_clock} km = {Delta_rho_clock*1000} m")

# ============================================================================
# QUESTION 2: LIGHT TIME (SAGNAC) EFFECT
# ============================================================================
print("\n" + "="*70)
print("QUESTION 2: LIGHT TIME (SAGNAC) EFFECT")
```

```
print("="*70)

# Calculate GPS satellite position range (min and max)
r_gps_perigee = a_gps * (1 - e_gps)  # Closest point
r_gps_apogee = a_gps * (1 + e_gps)   # Farthest point

# Geometry for LEO-GPS ranging
# Minimum range: GPS at zenith (directly above LEO)
# Use apogee distance for worst case
rho_min = r_gps_apogee - r_leo  # [km]

# Maximum range: GPS at horizon (tangent to LEO orbit)
# Geometric distance from LEO to horizon + GPS to horizon
tangent_leo = np.sqrt(r_leo**2 - Re**2)
tangent_gps = np.sqrt(r_gps_apogee**2 - Re**2)

rho_max = tangent_leo + tangent_gps

# Signal travel times
tau_min = rho_min / c  # [s]
tau_max = rho_max / c  # [s]

# Sagnac effect calculation
# Note: Lecture notes Section 6.2.1 refers to this as "Light time effect"
# caused by motion "due to Earth rotation".
# The Sagnac correction accounts for Earth rotation during signal travel
# The magnitude is approximately: Î\224Ï\201_sagnac â\211\210 Ï\211_e * tau * r * sin(angle
)
# For a more accurate calculation, the correction is:
# Î\224Ï\201_sagnac = (Ï\211_e / c) * (r_gps Ã\227 r_leo)_z component projected
# Or approximately: Ï\211_e * tau * (r_gps + r_leo) / 2 for the perpendicular component

# Conservative estimate using the cross-range displacement
# Maximum Sagnac occurs when GPS and LEO are perpendicular in the ECEF frame
# Î\224Ï\201_sagnac â\211\210 Ï\211_e * tau * r * sin(angle)

# For MINIMUM Sagnac (Zenith case):
# The geometry is nearly aligned along the radial direction (angle â\211\210 0)
# sin(angle) â\211\210 0, so Sagnac effect is minimal
# For practical purposes, even a small misalignment gives a small effect
angle_min = 0.0  # Radians (aligned geometry)
Delta_rho_sagnac_min = omega_e * tau_min * r_leo * np.sin(angle_min)  # [km]

# For MAXIMUM Sagnac (Horizon case):
# Maximum cross-range displacement occurs when geometry is perpendicular (angle â\211\210 9
0Â°)
# sin(90Â°) = 1
angle_max = np.pi / 2  # 90 degrees in radians (perpendicular geometry)
Delta_rho_sagnac_max = omega_e * tau_max * r_gps_apogee * np.sin(angle_max)  # [km]

print(f"\nGeometry Analysis:")
print(f"  LEO orbital radius:           {r_leo:.3f} km")
print(f"  GPS perigee radius:           {r_gps_perigee:.3f} km")
print(f"  GPS apogee radius:            {r_gps_apogee:.3f} km")
print(f"\nRange Calculations:")
print(f"  Minimum range (Zenith):       {rho_min:.3f} km")
print(f"  Maximum range (Horizon):      {rho_max:.3f} km")
print(f"\nSignal Travel Time:")
print(f"  Ï\204_min = {rho_min:.3f} / {c} = {tau_min*1e3:.6f} ms")
print(f"  Ï\204_max = {rho_max:.3f} / {c} = {tau_max*1e3:.6f} ms")
print(f"\nSagnac Correction (Ï\211_e Ã\227 Ï\204 Ã\227 r):")
print(f"  Î\224Ï\201_sagnac_min â\211\210 {omega_e:.6e} Ã\227 {tau_min:.6f} Ã\227 {r_leo:.3
f}")
print(f"  Î\224Ï\201_sagnac_min â\211\210 {Delta_rho_sagnac_min*1e3:.3f} m")
print(f"\n  Î\224Ï\201_sagnac_max â\211\210 {omega_e:.6e} Ã\227 {tau_max:.6f} Ã\227 {r_gps_
apogee:.3f}")
print(f"  Î\224Ï\201_sagnac_max â\211\210 {Delta_rho_sagnac_max*1e3:.3f} m")
print(f"\n>>> ANSWER:")
print(f"    Minimum Sagnac effect: {Delta_rho_sagnac_min*1e3:.3f} m")
```

```
print(f"    Maximum Sagnac effect: {Delta_rho_sagnac_max*1e3:.3f} m")

# ============================================================================
# QUESTION 3: RELATIVISTIC EFFECT (ECCENTRICITY)
# ============================================================================
print("\n" + "="*70)
print("QUESTION 3: RELATIVISTIC EFFECT (ECCENTRICITY)")
print("="*70)

# Relativistic effect formula: Î´t_rel = -2 * (r Â• v) / c^2
# For an elliptical orbit, the radial component of velocity is:
# v_r = dr/dt = (n * a * e * sin(f)) / sqrt(1 - e^2)
# where n is the mean motion, f is true anomaly
#
# The dot product r Â• v equals r * v_r (radial component)
# Maximum occurs when sin(f) = Â±1 (at f = 90Â° or 270Â°)

# Mean motion
n = np.sqrt(mu_earth / a_gps**3)  # [rad/s]

# At true anomaly f = 90Â° or 270Â°:
# r = a(1 - e^2) / (1 + e*cos(90Â°)) = a(1 - e^2)
r_at_90 = a_gps * (1 - e_gps**2)  # [km]

# Radial velocity at f = 90Â°:
# v_r = (n * a * e * sin(90Â°)) / sqrt(1 - e^2)
v_r_max = (n * a_gps * e_gps * 1.0) / np.sqrt(1 - e_gps**2)  # [km/s]

# Maximum r Â• v
r_dot_v_max = r_at_90 * v_r_max  # [km^2/s]

# Relativistic time effect
delta_t_rel_max = 2.0 * np.abs(r_dot_v_max) / (c**2)  # [s]

# Convert to range error
Delta_rho_rel_max = c * delta_t_rel_max  # [km]

# Minimum effect occurs at apogee and perigee where sin(f) = 0
Delta_rho_rel_min = 0.0  # [km]

print(f"\nOrbital Parameters:")
print(f"  Mean motion (n):              {n:.6e} rad/s")
print(f"  Radius at f=90Â°:              {r_at_90:.3f} km")
print(f"\nRadial Velocity Calculation:")
print(f"  v_r = (n Ã\227 a Ã\227 e) / â\210\232(1-eÂ²)")
print(f"  v_r_max = ({n:.6e} Ã\227 {a_gps} Ã\227 {e_gps}) / â\210\232(1-{e_gps}Â²)")
print(f"  v_r_max = {v_r_max:.6f} km/s")
print(f"\nDot Product rÂ•v:")
print(f"  (rÂ•v)_max = r Ã\227 v_r")
print(f"  (rÂ•v)_max = {r_at_90:.3f} Ã\227 {v_r_max:.6f}")
print(f"  (rÂ•v)_max = {r_dot_v_max:.3f} kmÂ²/s")
print(f"\nRelativistic Time Effect:")
print(f"  Î´t_rel = 2 Ã\227 |rÂ•v| / cÂ²")
print(f"  Î´t_rel_max = 2 Ã\227 {r_dot_v_max:.3f} / ({c})Â²")
print(f"  Î´t_rel_max = {delta_t_rel_max:.6e} s = {delta_t_rel_max*1e9:.3f} ns")
print(f"\nRange Error:")
print(f"  Î\224Ï\201_rel = c Ã\227 Î´t_rel")
print(f"  Î\224Ï\201_rel_max = {c} Ã\227 {delta_t_rel_max:.6e}")
print(f"  Î\224Ï\201_rel_max = {Delta_rho_rel_max*1e3:.6f} m")
print(f"\n>>> ANSWER:")
print(f"    Minimum relativistic effect: {Delta_rho_rel_min} m")
print(f"    Maximum relativistic effect: {Delta_rho_rel_max*1e3:.6f} m")

# ============================================================================
# SUMMARY OF ANSWERS
# ============================================================================
print("\n" + "="*70)
print("SUMMARY OF ANSWERS")
print("="*70)
```

```
print(f"\n1. CLOCK OFFSET EFFECT:")
print(f"   Range error = {Delta_rho_clock:.2g} km")
print(f"\n2. LIGHT TIME (SAGNAC) EFFECT:")
print(f"   Minimum effect = {Delta_rho_sagnac_min*1e3:.2g} m")
print(f"   Maximum effect = {Delta_rho_sagnac_max*1e3:.2g} m")
print(f"\n3. RELATIVISTIC EFFECT (ECCENTRICITY):")
print(f"   Minimum effect = {Delta_rho_rel_min:.2g} m")
print(f"   Maximum effect = {Delta_rho_rel_max*1e3:.2g} m")
print("="*70)
```

```
"""
GNSS Point Positioning using Iterative Least Squares (ILS)

This script implements an ILS solver for GNSS positioning using L1/L2 data.
It estimates the receiver's state vector [x_r, y_r, z_r, c*dt_r] at each epoch.
"""

import numpy as np
import matplotlib.pyplot as plt
from pathlib import Path

# Physical constants
c = 299792.458  # Speed of light [km/s]
omega_e = 7.2921151467e-5  # Earth rotation rate [rad/s]

# Data directory (relative to project root)
data_dir = Path(__file__).parent.parent / 'data' / 'provided_assignment'


def load_data():
    """Load all GNSS data from text files."""
    print("Loading data...")

    # Load time data
    t = np.loadtxt(data_dir / 't.txt')  # Time [s]

    # Load PRN IDs and CA range
    PRN_ID = np.loadtxt(data_dir / 'PRN_ID.txt')  # Satellite PRN IDs
    CA_range = np.loadtxt(data_dir / 'CA_range.txt')  # Pseudorange measurements [km]

    # Load GPS satellite positions (ECEF) [km]
    rx_gps = np.loadtxt(data_dir / 'rx_gps.txt')
    ry_gps = np.loadtxt(data_dir / 'ry_gps.txt')
    rz_gps = np.loadtxt(data_dir / 'rz_gps.txt')

    # Load GPS satellite velocities (ECEF) [km/s]
    vx_gps = np.loadtxt(data_dir / 'vx_gps.txt')
    vy_gps = np.loadtxt(data_dir / 'vy_gps.txt')
    vz_gps = np.loadtxt(data_dir / 'vz_gps.txt')

    # Load satellite clock corrections [s]
    clk_gps = np.loadtxt(data_dir / 'clk_gps.txt')

    # Load true receiver positions (for comparison) [km]
    rx_true = np.loadtxt(data_dir / 'rx.txt')
    ry_true = np.loadtxt(data_dir / 'ry.txt')
    rz_true = np.loadtxt(data_dir / 'rz.txt')

    # Load true receiver velocities [km/s]
    vx_true = np.loadtxt(data_dir / 'vx.txt')
    vy_true = np.loadtxt(data_dir / 'vy.txt')
    vz_true = np.loadtxt(data_dir / 'vz.txt')

    print(f"Data loaded successfully!")
    print(f"Number of epochs: {len(t)}")
    print(f"Data shape: {PRN_ID.shape}")

    return {
        't': t,
        'PRN_ID': PRN_ID,
        'CA_range': CA_range,
        'rx_gps': rx_gps,
        'ry_gps': ry_gps,
        'rz_gps': rz_gps,
        'vx_gps': vx_gps,
        'vy_gps': vy_gps,
        'vz_gps': vz_gps,
        'clk_gps': clk_gps,
        'rx_true': rx_true,
```

```
        'ry_true': ry_true,
        'rz_true': rz_true,
        'vx_true': vx_true,
        'vy_true': vy_true,
        'vz_true': vz_true
    }


def rotation_matrix_z(angle):
    """
    Create a rotation matrix around the z-axis.

    Parameters:
    -----------
    angle : float or ndarray
        Rotation angle in radians

    Returns:
    --------
    R : ndarray
        Rotation matrix (3x3) or array of rotation matrices
    """
    cos_a = np.cos(angle)
    sin_a = np.sin(angle)

    if np.isscalar(angle):
        R = np.array([
            [cos_a, sin_a, 0],
            [-sin_a, cos_a, 0],
            [0, 0, 1]
        ])
    else:
        # Handle multiple angles at once
        n = len(angle)
        R = np.zeros((n, 3, 3))
        R[:, 0, 0] = cos_a
        R[:, 0, 1] = sin_a
        R[:, 1, 0] = -sin_a
        R[:, 1, 1] = cos_a
        R[:, 2, 2] = 1

    return R


def ils_solver(data, max_iter=10, tol=1e-3):
    """
    Iterative Least Squares solver for GNSS point positioning.

    Parameters:
    -----------
    data : dict
        Dictionary containing all loaded GNSS data
    max_iter : int
        Maximum number of iterations for the ILS
    tol : float
        Convergence tolerance [km]

    Returns:
    --------
    results : ndarray
        Estimated receiver states [x_r, y_r, z_r, c*dt_r] for each epoch
    """
    # Extract data
    t = data['t']
    PRN_ID = data['PRN_ID']
    CA_range = data['CA_range']
    rx_gps = data['rx_gps']
    ry_gps = data['ry_gps']
    rz_gps = data['rz_gps']
```

```
vx_gps = data['vx_gps']
vy_gps = data['vy_gps']
vz_gps = data['vz_gps']
clk_gps = data['clk_gps']
rx_true = data['rx_true']
ry_true = data['ry_true']
rz_true = data['rz_true']

N_epochs = len(t)

# Initialize results array
results = np.zeros((N_epochs, 4))

# Initial guess for the first epoch
x_est = np.array([rx_true[0], ry_true[0], rz_true[0], 0.0])

print("\nStarting ILS processing...")

# Loop through all epochs
for i in range(N_epochs):
    if i % 100 == 0:
        print(f"Processing epoch {i+1}/{N_epochs}...")

    # 1. Select valid satellites (non-zero PRN_ID and non-zero CA_range)
    valid_mask = (PRN_ID[i] != 0) & (CA_range[i] != 0)

    # Filter data for valid satellites
    prn_valid = PRN_ID[i][valid_mask]
    rho_obs = CA_range[i][valid_mask]  # Observed pseudorange
    r_gps = np.column_stack([
        rx_gps[i][valid_mask],
        ry_gps[i][valid_mask],
        rz_gps[i][valid_mask]
    ])
    v_gps = np.column_stack([
        vx_gps[i][valid_mask],
        vy_gps[i][valid_mask],
        vz_gps[i][valid_mask]
    ])
    dt_clk = clk_gps[i][valid_mask]  # Satellite clock correction [s]

    N_valid = len(prn_valid)

    if N_valid < 4:
        print(f"Warning: Only {N_valid} valid satellites at epoch {i}")
        results[i] = x_est
        continue

    # 2. Iterative Least Squares
    for iteration in range(max_iter):
        # Current receiver position estimate
        r_rx = x_est[0:3]
        dt_rx = x_est[3] / c  # Receiver clock offset [s]

        # a. Calculate geometric range
        dr = r_gps - r_rx  # Vector from receiver to satellite
        rho_geo = np.linalg.norm(dr, axis=1)  # Geometric distance [km]

        # b. Light-time (Sagnac) correction
        tau = rho_geo / c  # Signal travel time [s]
        angle = -omega_e * tau  # Rotation angle [rad]

        # Apply rotation to each satellite position
        r_corr = np.zeros_like(r_gps)
        for j in range(N_valid):
            R_z = rotation_matrix_z(angle[j])
            r_corr[j] = R_z @ r_gps[j]

        # Recalculate geometric range with corrected positions
```

```
            dr_corr = r_corr - r_rx
            rho_geo = np.linalg.norm(dr_corr, axis=1)

            # c. Relativistic correction
            # delta_rel = -2 * (r_gps Â• v_gps) / c^2
            r_dot_v = np.sum(r_gps * v_gps, axis=1)
            delta_rel = -2.0 * r_dot_v / (c**2)  # [s]

            # d. Modeled range
            # rho_model = rho_geo + c * (dt_rx - dt_clk - delta_rel)
            rho_model = rho_geo + c * (dt_rx - dt_clk - delta_rel)

            # e. Design matrix H
            H = np.zeros((N_valid, 4))
            # First three columns: unit line-of-sight vector (negative)
            H[:, 0:3] = -dr_corr / rho_geo[:, np.newaxis]
            # Last column: all ones (for clock offset)
            H[:, 3] = 1.0

            # f. Residual vector
            dy = rho_obs - rho_model

            # g. Solve normal equations: dx = (H^T H)^-1 H^T dy
            HTH = H.T @ H
            HTdy = H.T @ dy
            dx = np.linalg.solve(HTH, HTdy)

            # h. Update state
            x_est = x_est + dx

            # i. Check convergence
            if np.linalg.norm(dx) < tol:
                break

        # Store result for this epoch
        results[i] = x_est

        # Use current result as initial guess for next epoch
        # (already in x_est, no need to reassign)

    print("ILS processing completed!")
    return results


def plot_results(data, results):
    """
    Plot the estimated receiver position versus true position.

    Parameters:
    -----------
    data : dict
        Dictionary containing all loaded GNSS data
    results : ndarray
        Estimated receiver states from ILS
    """
    t = data['t']
    rx_true = data['rx_true']
    ry_true = data['ry_true']
    rz_true = data['rz_true']

    # Extract estimated positions
    rx_est = results[:, 0]
    ry_est = results[:, 1]
    rz_est = results[:, 2]
    c_dt_r = results[:, 3]

    # Calculate position errors
    error_x = rx_est - rx_true
    error_y = ry_est - ry_true
```

```
    error_z = rz_est - rz_true
    error_3d = np.sqrt(error_x**2 + error_y**2 + error_z**2)

    # Create plots
    fig, axes = plt.subplots(2, 2, figsize=(14, 10))

    # Plot 1: X-position
    axes[0, 0].plot(t, rx_true, 'b-', label='True', linewidth=2)
    axes[0, 0].plot(t, rx_est, 'r--', label='Estimated', linewidth=1.5)
    axes[0, 0].set_xlabel('Time [s]')
    axes[0, 0].set_ylabel('X Position [km]')
    axes[0, 0].set_title('Receiver X-Position')
    axes[0, 0].legend()
    axes[0, 0].grid(True, alpha=0.3)

    # Plot 2: Y-position
    axes[0, 1].plot(t, ry_true, 'b-', label='True', linewidth=2)
    axes[0, 1].plot(t, ry_est, 'r--', label='Estimated', linewidth=1.5)
    axes[0, 1].set_xlabel('Time [s]')
    axes[0, 1].set_ylabel('Y Position [km]')
    axes[0, 1].set_title('Receiver Y-Position')
    axes[0, 1].legend()
    axes[0, 1].grid(True, alpha=0.3)

    # Plot 3: Z-position
    axes[1, 0].plot(t, rz_true, 'b-', label='True', linewidth=2)
    axes[1, 0].plot(t, rz_est, 'r--', label='Estimated', linewidth=1.5)
    axes[1, 0].set_xlabel('Time [s]')
    axes[1, 0].set_ylabel('Z Position [km]')
    axes[1, 0].set_title('Receiver Z-Position')
    axes[1, 0].legend()
    axes[1, 0].grid(True, alpha=0.3)

    # Plot 4: 3D position error
    axes[1, 1].plot(t, error_3d * 1000, 'g-', linewidth=2)  # Convert to meters
    axes[1, 1].set_xlabel('Time [s]')
    axes[1, 1].set_ylabel('3D Position Error [m]')
    axes[1, 1].set_title('3D Position Error')
    axes[1, 1].grid(True, alpha=0.3)

    plt.tight_layout()

    # Save plot to plots directory
    plot_path = Path(__file__).parent.parent / 'plots' / 'gnss_positioning_results.png'
    plt.savefig(plot_path, dpi=300, bbox_inches='tight')
    print(f"\nPlot saved as '{plot_path}'")
    plt.show()

    # Print statistics
    print("\n" + "="*60)
    print("POSITIONING STATISTICS")
    print("="*60)
    print(f"Mean 3D Error: {np.mean(error_3d)*1000:.3f} m")
    print(f"RMS 3D Error:  {np.sqrt(np.mean(error_3d**2))*1000:.3f} m")
    print(f"Max 3D Error:  {np.max(error_3d)*1000:.3f} m")
    print(f"Min 3D Error:  {np.min(error_3d)*1000:.3f} m")
    print(f"Std 3D Error:  {np.std(error_3d)*1000:.3f} m")
    print("="*60)
    print(f"Mean X Error: {np.mean(error_x)*1000:.3f} m")
    print(f"Mean Y Error: {np.mean(error_y)*1000:.3f} m")
    print(f"Mean Z Error: {np.mean(error_z)*1000:.3f} m")
    print("="*60)
    print(f"Mean Clock Offset: {np.mean(c_dt_r):.3f} km = {np.mean(c_dt_r)/c*1e9:.3f} ns")
    print("="*60)


def main():
    """Main execution function."""
    # Load data
```

```
    data = load_data()

    # Run ILS solver
    results = ils_solver(data, max_iter=10, tol=1e-3)

    # Plot results
    plot_results(data, results)


if __name__ == "__main__":
    main()
```

```
"""
Least-Squares Orbit Determination Calculations
GRACE-FO LEO Satellite using GPS Pseudo-range Observations

Task 1: Compute Observation Covariance Matrix (P_yy)
Task 2: Linearise and Evaluate a Single Pseudo-range Observation
"""

import numpy as np

# Set print options for better formatting
np.set_printoptions(precision=6, suppress=True)

print("="*80)
print("LEAST-SQUARES ORBIT DETERMINATION CALCULATIONS")
print("="*80)

# ============================================================================
# TASK 1: OBSERVATION COVARIANCE MATRIX (P_yy)
# ============================================================================
print("\n" + "="*80)
print("TASK 1: OBSERVATION COVARIANCE MATRIX (P_yy)")
print("="*80)

# Given parameters
n_obs = 4  # Number of observations
sigma = 3.0  # Standard deviation [m]
rho = 0.2  # Correlation coefficient

print(f"\nGiven:")
print(f"  Number of observations (n):   {n_obs}")
print(f"  Standard deviation (Ï\203):       {sigma} m")
print(f"  Correlation coefficient (Ï\201):  {rho}")

# Compute variance
variance = sigma**2  # [m^2]

# Construct the covariance matrix
# P_ii = Ï\203^2 (variance on diagonal)
# P_ij = Ï\201 * Ï\203_i * Ï\203_j (covariance off-diagonal)
P_yy = np.zeros((n_obs, n_obs))

for i in range(n_obs):
    for j in range(n_obs):
        if i == j:
            P_yy[i, j] = variance  # Diagonal: variance
        else:
            P_yy[i, j] = rho * sigma * sigma  # Off-diagonal: covariance

print(f"\nObservation Covariance Matrix P_yy [{n_obs}Ã\227{n_obs}] (mÂ²):")
print(P_yy)

print(f"\n>>> ANSWER (Question 9):")
print(f"    Diagonal elements (P_ii):     {variance:.2f} mÂ²")
print(f"    Off-diagonal elements (P_ij): {rho * sigma * sigma:.2f} mÂ²")

# ============================================================================
# TASK 2: LINEARISE A SINGLE PSEUDO-RANGE OBSERVATION
# ============================================================================
print("\n" + "="*80)
print("TASK 2: LINEARISE A SINGLE PSEUDO-RANGE OBSERVATION")
print("="*80)

# Constants
c = 299792.458  # Speed of light [km/s]

# Initial receiver state (GRACE-FO LEO satellite)
x_0 = 6878.0  # [km]
y_0 = 0.0     # [km]
```

```
z_0 = 0.0     # [km]
dt_r_0 = 0.001  # Receiver clock offset [s]

# PRN (GPS satellite) position
x_PRN = -371.9349049   # [km]
y_PRN = 19871.754959   # [km]
z_PRN = 17630.753853   # [km]
dt_t = 5.977963643e-5  # Transmitter clock offset [s]

print(f"\nGiven Constants:")
print(f"  Speed of light (c):           {c} km/s")

print(f"\nInitial Receiver State (GRACE-FO):")
print(f"  x_0:                          {x_0} km")
print(f"  y_0:                          {y_0} km")
print(f"  z_0:                          {z_0} km")
print(f"  Î´t_r,0:                       {dt_r_0} s")

print(f"\nPRN (GPS Satellite) State:")
print(f"  x_PRN:                        {x_PRN} km")
print(f"  y_PRN:                        {y_PRN} km")
print(f"  z_PRN:                        {z_PRN} km")
print(f"  Î´t_t:                         {dt_t} s")

# ============================================================================
# CALCULATION 1: Geometric Range (R_0)
# ============================================================================
print(f"\n" + "-"*80)
print("CALCULATION 1: Geometric Range (R_0)")
print("-"*80)

# Position difference vector
dx = x_0 - x_PRN
dy = y_0 - y_PRN
dz = z_0 - z_PRN

print(f"\nPosition difference vector:")
print(f"  Î\224x = x_0 - x_PRN = {x_0} - ({x_PRN}) = {dx:.6f} km")
print(f"  Î\224y = y_0 - y_PRN = {y_0} - ({y_PRN}) = {dy:.6f} km")
print(f"  Î\224z = z_0 - z_PRN = {z_0} - ({z_PRN}) = {dz:.6f} km")

# Geometric range
R_0 = np.sqrt(dx**2 + dy**2 + dz**2)

print(f"\nGeometric Range:")
print(f"  R_0 = â\210\232((Î\224x)Â² + (Î\224y)Â² + (Î\224z)Â²)")
print(f"  R_0 = â\210\232({dx:.6f}Â² + {dy:.6f}Â² + {dz:.6f}Â²)")
print(f"  R_0 = â\210\232{dx**2 + dy**2 + dz**2:.6f}")
print(f"  R_0 = {R_0:.6f} km")

print(f"\n>>> ANSWER: R_0 = {R_0:.6f} km = {R_0*1000:.3f} m")

# ============================================================================
# CALCULATION 2: Calculated Pseudo-range f(x_0)
# ============================================================================
print(f"\n" + "-"*80)
print("CALCULATION 2: Calculated Pseudo-range f(x_0)")
print("-"*80)

# Clock offset contribution
clock_offset = c * (dt_r_0 - dt_t)

print(f"\nClock offset contribution:")
print(f"  c(Î´t_r - Î´t_t) = {c} Ã\227 ({dt_r_0} - {dt_t})")
print(f"  c(Î´t_r - Î´t_t) = {c} Ã\227 {dt_r_0 - dt_t}")
print(f"  c(Î´t_r - Î´t_t) = {clock_offset:.6f} km")

# Pseudo-range
f_x0 = R_0 + clock_offset
```

```
print(f"\nPseudo-range:")
print(f"  f(x_0) = R_0 + c(Î´t_r,0 - Î´t_t)")
print(f"  f(x_0) = {R_0:.6f} + {clock_offset:.6f}")
print(f"  f(x_0) = {f_x0:.6f} km")

print(f"\n>>> ANSWER: f(x_0) = {f_x0:.6f} km = {f_x0*1000:.3f} m")

# ============================================================================
# CALCULATION 3: Design Matrix H (1Ã\2274)
# ============================================================================
print(f"\n" + "-"*80)
print("CALCULATION 3: Design Matrix H (1Ã\2274)")
print("-"*80)

# Design matrix elements
H_11 = dx / R_0  # â\210\202Ï\201/â\210\202x = (x_0 - x_PRN) / R_0
H_12 = dy / R_0  # â\210\202Ï\201/â\210\202y = (y_0 - y_PRN) / R_0
H_13 = dz / R_0  # â\210\202Ï\201/â\210\202z = (z_0 - z_PRN) / R_0
H_14 = c         # â\210\202Ï\201/â\210\202(Î´t_r) = c

# Construct design matrix
H = np.array([[H_11, H_12, H_13, H_14]])

print(f"\nDesign matrix elements:")
print(f"  H_11 = (x_0 - x_PRN) / R_0 = {dx:.6f} / {R_0:.6f} = {H_11:.6f}")
print(f"  H_12 = (y_0 - y_PRN) / R_0 = {dy:.6f} / {R_0:.6f} = {H_12:.6f}")
print(f"  H_13 = (z_0 - z_PRN) / R_0 = {dz:.6f} / {R_0:.6f} = {H_13:.6f}")
print(f"  H_14 = c = {H_14:.6f} km/s")

print(f"\nDesign Matrix H [1Ã\2274]:")
print(H)

print(f"\n>>> ANSWER (Question 13):")
print(f"    H_11 (unitless):  {H_11:.6f}")
print(f"    H_12 (unitless):  {H_12:.6f}")
print(f"    H_13 (unitless):  {H_13:.6f}")
print(f"    H_14 (km/s):      {H_14:.6f}")

# ============================================================================
# SUMMARY OF RESULTS
# ============================================================================
print("\n" + "="*80)
print("SUMMARY OF RESULTS")
print("="*80)

print(f"\nTASK 1: Observation Covariance Matrix P_yy [4Ã\2274] (mÂ²):")
print(P_yy)
print(f"  Diagonal elements:     {variance:.2f} mÂ²")
print(f"  Off-diagonal elements: {rho * sigma * sigma:.2f} mÂ²")

print(f"\nTASK 2: Linearisation Results:")
print(f"  Geometric Range R_0:       {R_0:.6f} km")
print(f"  Pseudo-range f(x_0):       {f_x0:.6f} km")
print(f"  Design Matrix H [1Ã\2274]:")
print(f"    H_11 (unitless):         {H_11:.6f}")
print(f"    H_12 (unitless):         {H_12:.6f}")
print(f"    H_13 (unitless):         {H_13:.6f}")
print(f"    H_14 (km/s):             {H_14:.6f}")

print("="*80)
```

```
"""
Iterative Least-Squares (ILS) Orbit Determination for GRACE-FO Satellite
Using GPS Pseudo-range Measurements

This script implements:
- Basic model (clock corrections only)
- Advanced model (with Light-Time and Relativistic corrections)
- Error analysis with velocity correction hint
- Residual analysis
- PDOP calculation and plotting
"""

import numpy as np
import matplotlib.pyplot as plt
from pathlib import Path

# Physical constants
c = 299792.458  # Speed of light [km/s]
omega_e = 7.2921151467e-5  # Earth rotation rate [rad/s]

# Data directory (relative to project root)
data_dir = Path(__file__).parent.parent / 'data' / 'provided_assignment'


def load_grace_data():
    """Load all GRACE-FO orbit determination data from text files."""
    print("Loading GRACE-FO data...")

    # Load time data
    t = np.loadtxt(data_dir / 't.txt')  # Time [s]

    # Load PRN IDs
    PRN_ID = np.loadtxt(data_dir / 'PRN_ID.txt')  # Satellite PRN IDs

    # Load GPS satellite positions (ECEF) [km]
    x_GPS = np.loadtxt(data_dir / 'rx_gps.txt')
    y_GPS = np.loadtxt(data_dir / 'ry_gps.txt')
    z_GPS = np.loadtxt(data_dir / 'rz_gps.txt')

    # Load GPS satellite velocities (ECEF) [km/s]
    vx_GPS = np.loadtxt(data_dir / 'vx_gps.txt')
    vy_GPS = np.loadtxt(data_dir / 'vy_gps.txt')
    vz_GPS = np.loadtxt(data_dir / 'vz_gps.txt')

    # Load GPS satellite clock offsets [s]
    dt_GPS = np.loadtxt(data_dir / 'clk_gps.txt')

    # Load observed pseudo-ranges [km]
    rho_obs = np.loadtxt(data_dir / 'CA_range.txt')

    # Load GRACE-FO precise reference positions (ECEF) [km]
    x_GRACE = np.loadtxt(data_dir / 'rx.txt')
    y_GRACE = np.loadtxt(data_dir / 'ry.txt')
    z_GRACE = np.loadtxt(data_dir / 'rz.txt')

    # Load GRACE-FO precise velocities (ECEF) [km/s]
    vx_GRACE = np.loadtxt(data_dir / 'vx.txt')
    vy_GRACE = np.loadtxt(data_dir / 'vy.txt')
    vz_GRACE = np.loadtxt(data_dir / 'vz.txt')

    print(f"Data loaded successfully!")
    print(f"Number of epochs: {len(t)}")
    print(f"Data shape: {PRN_ID.shape}")

    return {
        't': t,
        'PRN_ID': PRN_ID,
        'x_GPS': x_GPS,
        'y_GPS': y_GPS,
```

```
        'z_GPS': z_GPS,
        'vx_GPS': vx_GPS,
        'vy_GPS': vy_GPS,
        'vz_GPS': vz_GPS,
        'dt_GPS': dt_GPS,
        'rho_obs': rho_obs,
        'x_GRACE': x_GRACE,
        'y_GRACE': y_GRACE,
        'z_GRACE': z_GRACE,
        'vx_GRACE': vx_GRACE,
        'vy_GRACE': vy_GRACE,
        'vz_GRACE': vz_GRACE
    }


def rotation_matrix_z(angle):
    """
    Create a rotation matrix around the z-axis for Earth rotation correction.

    Parameters:
    -----------
    angle : float
        Rotation angle in radians

    Returns:
    --------
    R : ndarray (3x3)
        Rotation matrix
    """
    cos_a = np.cos(angle)
    sin_a = np.sin(angle)

    R = np.array([
        [cos_a, sin_a, 0],
        [-sin_a, cos_a, 0],
        [0, 0, 1]
    ])

    return R


def light_time_correction(r_GPS, v_GPS, r_rx_est):
    """
    Apply light-time (Sagnac) correction for Earth rotation during signal travel.

    Parameters:
    -----------
    r_GPS : ndarray (3,)
        GPS satellite position [km]
    v_GPS : ndarray (3,)
        GPS satellite velocity [km/s]
    r_rx_est : ndarray (3,)
        Estimated receiver position [km]

    Returns:
    --------
    r_GPS_corr : ndarray (3,)
        Corrected GPS position [km]
    """
    # Calculate geometric range
    rho_geo = np.linalg.norm(r_GPS - r_rx_est)

    # Signal travel time
    tau = rho_geo / c

    # Rotation angle due to Earth rotation
    angle = omega_e * tau

    # Apply rotation to GPS position
```

```
    R_z = rotation_matrix_z(angle)
    r_GPS_corr = R_z @ (r_GPS - tau*v_GPS)

    return r_GPS_corr


def relativistic_correction(r_GPS, v_GPS):
    """
    Calculate relativistic correction due to GPS orbit eccentricity.

    Parameters:
    -----------
    r_GPS : ndarray (3,)
        GPS satellite position [km]
    v_GPS : ndarray (3,)
        GPS satellite velocity [km/s]

    Returns:
    --------
    delta_t_rel : float
        Relativistic time correction [s]
    """
    # Relativistic effect: delta_t = -2 * (r Â• v) / c^2
    r_dot_v = np.dot(r_GPS, v_GPS)
    delta_t_rel = 2.0 * r_dot_v / (c**2)

    return delta_t_rel


def run_orbit_determination(data, corrections=False, max_iter=10, tol=1e-6):
    """
    Run Iterative Least-Squares orbit determination.

    Parameters:
    -----------
    data : dict
        Dictionary containing all loaded data
    corrections : bool
        If True, apply Light-Time and Relativistic corrections
    max_iter : int
        Maximum number of ILS iterations (Q17: stopping criterion #1)
    tol : float
        Convergence tolerance [km] (Q17: stopping criterion #2)
        Iteration stops when position update norm < tol

    Returns:
    --------
    results : dict
        Dictionary containing estimated states, errors, residuals, etc.
    """
    # Extract data
    t = data['t']
    PRN_ID = data['PRN_ID']
    x_GPS = data['x_GPS']
    y_GPS = data['y_GPS']
    z_GPS = data['z_GPS']
    vx_GPS = data['vx_GPS']
    vy_GPS = data['vy_GPS']
    vz_GPS = data['vz_GPS']
    dt_GPS = data['dt_GPS']
    rho_obs = data['rho_obs']
    x_GRACE = data['x_GRACE']
    y_GRACE = data['y_GRACE']
    z_GRACE = data['z_GRACE']
    vx_GRACE = data['vx_GRACE']
    vy_GRACE = data['vy_GRACE']
    vz_GRACE = data['vz_GRACE']

    N_epochs = len(t)
```

```
# Initialize result arrays
x_est = np.zeros(N_epochs)
y_est = np.zeros(N_epochs)
z_est = np.zeros(N_epochs)
dt_r_est = np.zeros(N_epochs)  # Receiver clock offset [s]
residuals = []  # Store residuals for each epoch
PDOP = np.zeros(N_epochs)
num_iterations = np.zeros(N_epochs, dtype=int)  # Track iterations per epoch

model_name = "Advanced" if corrections else "Basic"
print(f"\nRunning {model_name} Model ILS orbit determination...")

# Loop through all epochs
for i in range(N_epochs):
    if i % 50 == 0:
        print(f"Processing epoch {i+1}/{N_epochs}...")

    # Get valid satellites for this epoch (non-zero PRN_ID and rho_obs)
    valid_mask = (PRN_ID[i] != 0) & (rho_obs[i] != 0)

    if not np.any(valid_mask):
        # No valid observations, use previous estimate or NaN
        if i > 0:
            x_est[i] = x_est[i-1]
            y_est[i] = y_est[i-1]
            z_est[i] = z_est[i-1]
            dt_r_est[i] = dt_r_est[i-1]
        else:
            x_est[i] = np.nan
            y_est[i] = np.nan
            z_est[i] = np.nan
            dt_r_est[i] = np.nan
        PDOP[i] = np.nan
        residuals.append(np.array([]))
        continue

    # Filter data for valid satellites
    r_GPS_valid = np.column_stack([
        x_GPS[i][valid_mask],
        y_GPS[i][valid_mask],
        z_GPS[i][valid_mask]
    ])
    v_GPS_valid = np.column_stack([
        vx_GPS[i][valid_mask],
        vy_GPS[i][valid_mask],
        vz_GPS[i][valid_mask]
    ])
    dt_GPS_valid = dt_GPS[i][valid_mask]
    rho_obs_valid = rho_obs[i][valid_mask]

    N_valid = len(rho_obs_valid)

    # Initial guess: center of Earth or previous estimate
    if i == 0:
        x_state = np.array([0.0, 0.0, 0.0, 0.0])  # [x, y, z, c*dt_r]
    else:
        x_state = np.array([x_est[i-1], y_est[i-1], z_est[i-1], c*dt_r_est[i-1]])

    # Iterative Least Squares
    for iteration in range(max_iter):
        # Current estimate
        r_rx = x_state[0:3]
        c_dt_r = x_state[3]
        dt_r = c_dt_r / c

        # Initialize arrays for this iteration
        rho_calc = np.zeros(N_valid)
        H = np.zeros((N_valid, 4))
```

```
            # Loop through valid satellites
            for j in range(N_valid):
                r_GPS_j = r_GPS_valid[j]
                v_GPS_j = v_GPS_valid[j]
                dt_GPS_j = dt_GPS_valid[j]

                # Apply corrections if enabled
                if corrections:
                    # Light-time correction
                    r_GPS_j = light_time_correction(r_GPS_j, v_GPS_j, r_rx)

                    # Relativistic correction
                    delta_t_rel = relativistic_correction(r_GPS_valid[j], v_GPS_j)
                    # print(f"Relativistic correction for satellite {j}: {delta_t_rel:.6e}
s")
                    # delta_t_rel = 0
                else:
                    delta_t_rel = 0.0

                # Calculate geometric range
                dr = r_GPS_j - r_rx
                rho_geo = np.linalg.norm(dr)

                # Calculate modeled pseudo-range
                rho_calc[j] = rho_geo + c * (dt_r - dt_GPS_j + delta_t_rel)

                # Design matrix row
                H[j, 0:3] = -dr / rho_geo  # Partial derivatives w.r.t. position
                H[j, 3] = 1.0  # Partial derivative w.r.t. c*dt_r

            # Residual vector
            dy = rho_obs_valid - rho_calc

            # Solve normal equations: dx = (H^T H)^-1 H^T dy
            HTH = H.T @ H
            HTdy = H.T @ dy

            try:
                dx = np.linalg.solve(HTH, HTdy)
            except np.linalg.LinAlgError:
                # Singular matrix, use pseudoinverse
                dx = np.linalg.lstsq(HTH, HTdy, rcond=None)[0]

            # Update state
            x_state = x_state + dx

            # Check convergence
            if np.linalg.norm(dx[0:3]) < tol:
                num_iterations[i] = iteration + 1
                break
        else:
            num_iterations[i] = max_iter

        # Store results
        x_est[i] = x_state[0]
        y_est[i] = x_state[1]
        z_est[i] = x_state[2]
        dt_r_est[i] = x_state[3] / c
        residuals.append(dy)

        # Calculate PDOP
        # PDOP = sqrt(Q_11 + Q_22 + Q_33) where Q = (H^T H)^-1
        try:
            Q = np.linalg.inv(HTH)
            PDOP[i] = np.sqrt(Q[0,0] + Q[1,1] + Q[2,2])
        except np.linalg.LinAlgError:
            PDOP[i] = np.nan
```

```
    print(f"{model_name} Model ILS completed!")

    # Calculate errors with velocity correction hint
    # x_ref_corr = x_ref + v_ref * dt_r
    x_ref_corr = x_GRACE + vx_GRACE * dt_r_est
    y_ref_corr = y_GRACE + vy_GRACE * dt_r_est
    z_ref_corr = z_GRACE + vz_GRACE * dt_r_est

    # Absolute position error (L2 norm)
    pos_error = np.sqrt(
        (x_est - x_ref_corr)**2 +
        (y_est - y_ref_corr)**2 +
        (z_est - z_ref_corr)**2
    )

    return {
        't': t,
        'x_est': x_est,
        'y_est': y_est,
        'z_est': z_est,
        'dt_r_est': dt_r_est,
        'x_ref_corr': x_ref_corr,
        'y_ref_corr': y_ref_corr,
        'z_ref_corr': z_ref_corr,
        'pos_error': pos_error,
        'residuals': residuals,
        'PDOP': PDOP,
        'num_iterations': num_iterations
    }


def print_first_epochs_table(results, model_name, n_epochs=4):
    """Print table of estimated positions for first n epochs (Q16/Q20)."""
    print(f"{'Epoch':<8} {'Component':<12} {'Estimated [km]':<18} {'Reference (corr) [km]':
<23} {'Error [m]':<12}")
    print("-"*80)

    for i in range(min(n_epochs, len(results['t']))):
        x_est = results['x_est'][i]
        y_est = results['y_est'][i]
        z_est = results['z_est'][i]
        x_ref = results['x_ref_corr'][i]
        y_ref = results['y_ref_corr'][i]
        z_ref = results['z_ref_corr'][i]

        # Print with cm precision for positions, mm for errors
        print(f"{i+1:<8} {'X':<12} {x_est:>17.5f}  {x_ref:>22.5f}  {(x_est-x_ref)*1000:>11.
3f}")
        print(f"{'':8} {'Y':<12} {y_est:>17.5f}  {y_ref:>22.5f}  {(y_est-y_ref)*1000:>11.3f
}")
        print(f"{'':8} {'Z':<12} {z_est:>17.5f}  {z_ref:>22.5f}  {(z_est-z_ref)*1000:>11.3f
}")
        print(f"{'':8} {'|Position|':<12} {'':<18} {'':<23} {results['pos_error'][i]*1000:>
11.3f}")
        if i < n_epochs - 1:
            print("-"*80)

    print("="*80)


def compute_residual_statistics(residuals):
    """Compute RMS and other statistics from residuals list."""
    all_residuals = []
    for res_epoch in residuals:
        if len(res_epoch) > 0:
            all_residuals.extend(res_epoch)

    all_residuals = np.array(all_residuals)
```

```
    if len(all_residuals) == 0:
        return {'rms': np.nan, 'mean': np.nan, 'std': np.nan, 'max': np.nan}

    return {
        'rms': np.sqrt(np.mean(all_residuals**2)),
        'mean': np.mean(all_residuals),
        'std': np.std(all_residuals),
        'max': np.max(np.abs(all_residuals)),
        'count': len(all_residuals)
    }


def plot_position_error(results_basic, results_advanced):
    """Plot absolute position error vs time for both models."""
    fig, ax = plt.subplots(figsize=(12, 6))

    t = results_basic['t']

    ax.plot(t, results_basic['pos_error'] * 1000, 'b-',
            label='Basic Model', linewidth=1.5, alpha=0.7)
    ax.plot(t, results_advanced['pos_error'] * 1000, 'r-',
            label='Advanced Model', linewidth=1.5, alpha=0.7)

    ax.set_xlabel('Time [s]', fontsize=12)
    ax.set_ylabel('Absolute Position Error [m]', fontsize=12)
    ax.set_title('GRACE-FO Orbit Determination Error', fontsize=14, fontweight='bold')
    ax.legend(fontsize=11)
    ax.grid(True, alpha=0.3)

    plt.tight_layout()
    return fig


def plot_residuals(results_basic, results_advanced):
    """Plot measurement residuals for both models (Q21)."""
    fig, ax = plt.subplots(figsize=(12, 6))

    t = results_basic['t']

    # Flatten residuals (convert list of arrays to single array per epoch)
    res_basic = []
    res_advanced = []
    t_res = []

    for i, (rb, ra) in enumerate(zip(results_basic['residuals'],
                                     results_advanced['residuals'])):
        if len(rb) > 0:
            for r in rb:
                res_basic.append(r * 1000)  # Convert to meters
                t_res.append(t[i])
        if len(ra) > 0:
            for r in ra:
                res_advanced.append(r * 1000)  # Convert to meters

    # Plot as scatter or line
    ax.scatter(t_res, res_basic, c='blue', s=1, alpha=0.3, label='Basic Model (No Correctio
ns)')
    ax.scatter(t_res, res_advanced, c='red', s=1, alpha=0.3, label='Advanced Model (with Co
rrections)')

    ax.set_xlabel('Time [s]', fontsize=12)
    ax.set_ylabel('Pseudo-range Residuals [m]', fontsize=12)
    ax.set_title('Pseudo-range Measurement Residuals Comparison', fontsize=14, fontweight='
bold')
    ax.legend(fontsize=11)
    ax.grid(True, alpha=0.3)
    ax.axhline(y=0, color='k', linestyle='--', linewidth=0.8)

    plt.tight_layout()
```

```
    return fig


def plot_clock_offset(results):
    """Plot receiver clock offset vs time."""
    fig, ax = plt.subplots(figsize=(12, 6))

    t = results['t']
    dt_r = results['dt_r_est'] * 1e6  # Convert to microseconds

    ax.plot(t, dt_r, 'g-', linewidth=1.5)

    ax.set_xlabel('Time [s]', fontsize=12)
    ax.set_ylabel('Receiver Clock Offset [Î¼s]', fontsize=12)
    ax.set_title('GRACE-FO Receiver Clock Offset (Advanced Model)',
                 fontsize=14, fontweight='bold')
    ax.grid(True, alpha=0.3)

    plt.tight_layout()
    return fig


def plot_pdop_and_error(results):
    """Plot PDOP and position error on dual y-axis."""
    fig, ax1 = plt.subplots(figsize=(12, 6))

    t = results['t']

    # Plot PDOP on left y-axis
    color1 = 'tab:blue'
    ax1.set_xlabel('Time [s]', fontsize=12)
    ax1.set_ylabel('PDOP', fontsize=12, color=color1)
    ax1.plot(t, results['PDOP'], color=color1, linewidth=1.5, label='PDOP')
    ax1.tick_params(axis='y', labelcolor=color1)
    ax1.grid(True, alpha=0.3)

    # Plot position error on right y-axis
    ax2 = ax1.twinx()
    color2 = 'tab:red'
    ax2.set_ylabel('Position Error [m]', fontsize=12, color=color2)
    ax2.plot(t, results['pos_error'] * 1000, color=color2, linewidth=1.5,
             label='Position Error', alpha=0.7)
    ax2.tick_params(axis='y', labelcolor=color2)

    ax1.set_title('PDOP and Position Error (Advanced Model)',
                  fontsize=14, fontweight='bold')

    # Add legends
    lines1, labels1 = ax1.get_legend_handles_labels()
    lines2, labels2 = ax2.get_legend_handles_labels()
    ax1.legend(lines1 + lines2, labels1 + labels2, loc='upper left', fontsize=11)

    plt.tight_layout()
    return fig


def print_question_answers(results_basic, results_advanced):
    """Print answers to all assignment questions."""
    print("\n" + "#"*80)
    print("#" + " "*78 + "#")
    print("#" + " "*20 + "ASSIGNMENT QUESTION RESPONSES" + " "*29 + "#")
    print("#" + " "*78 + "#")
    print("#"*80)

    # Q14: Basic model implementation
    print("\n" + "="*80)
    print("Q14: Basic Model (Clock Corrections Only) - Position Errors")
    print("="*80)
    valid_basic = ~np.isnan(results_basic['pos_error'])
```

```
    print(f"Mean absolute error:  {np.nanmean(results_basic['pos_error'][valid_basic])*1000
:.3f} m")
    print(f"RMS error:            {np.sqrt(np.nanmean(results_basic['pos_error'][valid_basi
c]**2))*1000:.3f} m")
    print(f"Max error:            {np.nanmax(results_basic['pos_error'][valid_basic])*1000:
.3f} m")
    print(f"Min error:            {np.nanmin(results_basic['pos_error'][valid_basic])*1000:
.3f} m")
    print("See plot: grace_position_error_basic.png")

    # Q15: Discussion
    print("\n" + "="*80)
    print("Q15: Figure Discussion (Basic Model)")
    print("="*80)
    print("The absolute position errors show the accuracy of orbit determination using")
    print("only clock corrections. The errors are larger than the advanced model due to")
    print("unmodeled effects (light-time and relativistic corrections).")
    print("Typical errors are in the range of tens to hundreds of meters.")

    # Q16: First 4 epochs table
    print("\n" + "="*80)
    print("Q16: Estimated Positions - First 4 Epochs (Basic Model)")
    print("="*80)
    print_first_epochs_table(results_basic, "Basic", n_epochs=4)

    # Q17: Convergence criteria
    print("\n" + "="*80)
    print("Q17: Iterative Least Squares Convergence Criteria")
    print("="*80)
    print("The ILS iterations stop when ONE of the following conditions is met:")
    print("")
    print("1. CONVERGENCE: The position update norm falls below the tolerance:")
    print("   ||Î\224r|| < tol = 1e-6 km = 1 mm")
    print("   where Î\224r = [Î\224x, Î\224y, Î\224z] is the position correction vector.")
    print("")
    print("2. MAX ITERATIONS: The maximum number of iterations is reached:")
    print("   iteration > max_iter = 10")
    print("")
    print("This ensures both accuracy (convergence criterion) and computational")
    print("efficiency (iteration limit). In practice, most epochs converge within 3-5")
    print("iterations when starting from a reasonable initial guess.")
    print("")
    avg_iter_basic = np.mean(results_basic['num_iterations'][results_basic['num_iterations'
] > 0])
    avg_iter_adv = np.mean(results_advanced['num_iterations'][results_advanced['num_iterati
ons'] > 0])
    print(f"Average iterations (Basic Model):    {avg_iter_basic:.2f}")
    print(f"Average iterations (Advanced Model): {avg_iter_adv:.2f}")

    # Q18: Advanced model implementation
    print("\n" + "="*80)
    print("Q18: Advanced Model (with Light-Time & Relativistic Corrections) - Errors")
    print("="*80)
    valid_advanced = ~np.isnan(results_advanced['pos_error'])
    print(f"Mean absolute error:  {np.nanmean(results_advanced['pos_error'][valid_advanced]
)*1000:.3f} m")
    print(f"RMS error:            {np.sqrt(np.nanmean(results_advanced['pos_error'][valid_a
dvanced]**2))*1000:.3f} m")
    print(f"Max error:            {np.nanmax(results_advanced['pos_error'][valid_advanced])
*1000:.3f} m")
    print(f"Min error:            {np.nanmin(results_advanced['pos_error'][valid_advanced])
*1000:.3f} m")
    print("See plot: grace_position_error_advanced.png")

    # Q19: Discussion
    print("\n" + "="*80)
    print("Q19: Figure Discussion (Advanced Model)")
    print("="*80)
    print("The advanced model shows significantly improved accuracy compared to the basic")
```

```
    print("model. By including light-time (Sagnac) and relativistic corrections, the")
    print("systematic errors are reduced. The remaining errors are primarily due to:")
    print("- Measurement noise in pseudo-range observations")
    print("- Unmodeled effects (ionospheric delays, tropospheric delays, multipath)")
    print("- Clock modeling errors")
    improvement = (np.nanmean(results_basic['pos_error'][valid_basic]) -
                   np.nanmean(results_advanced['pos_error'][valid_advanced])) * 1000
    print(f"\nImprovement: {improvement:.3f} m (mean error reduction)")

    # Q20: First 4 epochs table
    print("\n" + "="*80)
    print("Q20: Estimated Positions - First 4 Epochs (Advanced Model)")
    print("="*80)
    print_first_epochs_table(results_advanced, "Advanced", n_epochs=4)

    # Q21 & Q22: Residual analysis
    print("\n" + "="*80)
    print("Q21-Q22: Residual Analysis and Comparison")
    print("="*80)
    print("\nChosen Residual Measure: RMS of Pseudo-range Residuals")
    print("-" * 80)
    print("")
    print("MOTIVATION:")
    print("The RMS (Root Mean Square) of pseudo-range residuals is chosen because:")
    print("1. It quantifies the average magnitude of observation-model mismatch")
    print("2. It is sensitive to both systematic and random errors")
    print("3. It provides a single scalar metric for model quality assessment")
    print("4. RMS is standard practice in GNSS analysis and least-squares estimation")
    print("")

    # Compute residual statistics
    stats_basic = compute_residual_statistics(results_basic['residuals'])
    stats_advanced = compute_residual_statistics(results_advanced['residuals'])

    print("RESIDUAL STATISTICS:")
    print("-" * 80)
    print(f"{'Metric':<25} {'Basic Model':<20} {'Advanced Model':<20}")
    print("-" * 80)
    print(f"{'RMS Residual [m]':<25} {stats_basic['rms']*1000:>19.4f} {stats_advanced['rms'
]*1000:>19.4f}")
    print(f"{'Mean Residual [m]':<25} {stats_basic['mean']*1000:>19.4f} {stats_advanced['me
an']*1000:>19.4f}")
    print(f"{'Std Dev [m]':<25} {stats_basic['std']*1000:>19.4f} {stats_advanced['std']*100
0:>19.4f}")
    print(f"{'Max |Residual| [m]':<25} {stats_basic['max']*1000:>19.4f} {stats_advanced['ma
x']*1000:>19.4f}")
    print(f"{'Number of Observations':<25} {stats_basic['count']:>19d} {stats_advanced['cou
nt']:>19d}")
    print("-" * 80)

    residual_improvement = stats_basic['rms'] - stats_advanced['rms']
    percent_improvement = (residual_improvement / stats_basic['rms']) * 100

    print(f"\nRMS Residual Improvement: {residual_improvement*1000:.4f} m ({percent_improve
ment:.2f}%)")
    print("")
    print("INTERPRETATION:")
    print("The advanced model shows" + (" improved" if residual_improvement > 0 else " simi
lar") + " residuals compared to")
    print("the basic model. This" + (" aligns" if residual_improvement > 0 else " may not a
lign") + " with expectations because:")
    print("")
    if residual_improvement > 0:
        print("â\234\223 Light-time correction accounts for Earth rotation during signal tr
ansit")
        print("  (Sagnac effect), which can introduce ~20-30m range errors.")
        print("â\234\223 Relativistic correction accounts for satellite clock variations du
e to")
        print("  orbital eccentricity, contributing ~1-2m range errors.")
```

```
        print("â\234\223 Including these corrections makes the observation model more physi
cally")
        print("  accurate, reducing systematic residuals.")
    else:
        print("  Even if residuals are similar, the position estimates may still improve")
        print("  because corrections affect the geometry of the solution.")
    print("")
    print("See plot: grace_residuals.png")
    print("="*80)


def main():
    """Main execution function."""
    # Load data
    data = load_grace_data()

    # Run Basic Model (no corrections)
    print("\n" + "="*80)
    print("BASIC MODEL (Clock Corrections Only)")
    print("="*80)
    results_basic = run_orbit_determination(data, corrections=False, max_iter=10, tol=1e-6)

    # Run Advanced Model (with corrections)
    print("\n" + "="*80)
    print("ADVANCED MODEL (with Light-Time and Relativistic Corrections)")
    print("="*80)
    results_advanced = run_orbit_determination(data, corrections=True, max_iter=10, tol=1e-
6)

    # Create plots directory if it doesn't exist
    plots_dir = Path(__file__).parent.parent / 'plots'
    plots_dir.mkdir(exist_ok=True)

    # Print comprehensive question answers
    print_question_answers(results_basic, results_advanced)

    # Generate plots
    print("\n" + "="*80)
    print("GENERATING PLOTS")
    print("="*80)

    # Plot 1: Basic Model Position Error (Q14)
    fig1 = plt.figure(figsize=(12, 6))
    plt.plot(results_basic['t'], results_basic['pos_error'] * 1000, 'b-', linewidth=1.5)
    plt.xlabel('Time [s]', fontsize=12)
    plt.ylabel('Absolute Position Error [m]', fontsize=12)
    plt.title('Q14: GRACE-FO Position Error - Basic Model (Clock Corrections Only)',
              fontsize=13, fontweight='bold')
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    fig1.savefig(plots_dir / 'grace_position_error_basic.png', dpi=300, bbox_inches='tight'
)
    print(f"â\234\223 Saved Q14 plot: grace_position_error_basic.png")

    # Plot 2: Advanced Model Position Error (Q18)
    fig2 = plt.figure(figsize=(12, 6))
    plt.plot(results_advanced['t'], results_advanced['pos_error'] * 1000, 'r-', linewidth=1
.5)
    plt.xlabel('Time [s]', fontsize=12)
    plt.ylabel('Absolute Position Error [m]', fontsize=12)
    plt.title('Q18: GRACE-FO Position Error - Advanced Model (with Corrections)',
              fontsize=13, fontweight='bold')
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    fig2.savefig(plots_dir / 'grace_position_error_advanced.png', dpi=300, bbox_inches='tig
ht')
    print(f"â\234\223 Saved Q18 plot: grace_position_error_advanced.png")

    # Plot 3: Residuals Comparison (Q21)
```

```
    fig3 = plot_residuals(results_basic, results_advanced)
    fig3.savefig(plots_dir / 'grace_residuals.png', dpi=300, bbox_inches='tight')
    print(f"â\234\223 Saved Q21 plot: grace_residuals.png")

    # Plot 4: Combined Position Error Comparison
    fig4 = plot_position_error(results_basic, results_advanced)
    fig4.savefig(plots_dir / 'grace_position_error_comparison.png', dpi=300, bbox_inches='t
ight')
    print(f"â\234\223 Saved comparison plot: grace_position_error_comparison.png")

    # Plot 5: Receiver Clock Offset
    fig5 = plot_clock_offset(results_advanced)
    fig5.savefig(plots_dir / 'grace_clock_offset.png', dpi=300, bbox_inches='tight')
    print(f"â\234\223 Saved: grace_clock_offset.png")

    # Plot 6: PDOP and Error
    fig6 = plot_pdop_and_error(results_advanced)
    fig6.savefig(plots_dir / 'grace_pdop_error.png', dpi=300, bbox_inches='tight')
    print(f"â\234\223 Saved: grace_pdop_error.png")

    print("="*80)
    print("\nAll plots saved to:", plots_dir)
    print("\n" + "#"*80)
    print("#" + " "*78 + "#")
    print("#" + " "*25 + "PROCESSING COMPLETE" + " "*33 + "#")
    print("#" + " "*78 + "#")
    print("#"*80)

    plt.show()


if __name__ == "__main__":
    main()
```