The Hong Kong Polytechnic University

Department of Land Surveying and Geo-informatics

LSGI3322 Satellite Positioning Systems

GPS Positioning Project

(Final Phase – Calculating GPS Receiver Position)

Subject Lecturer: Prof. George Liu

Student Name: Waterdragen

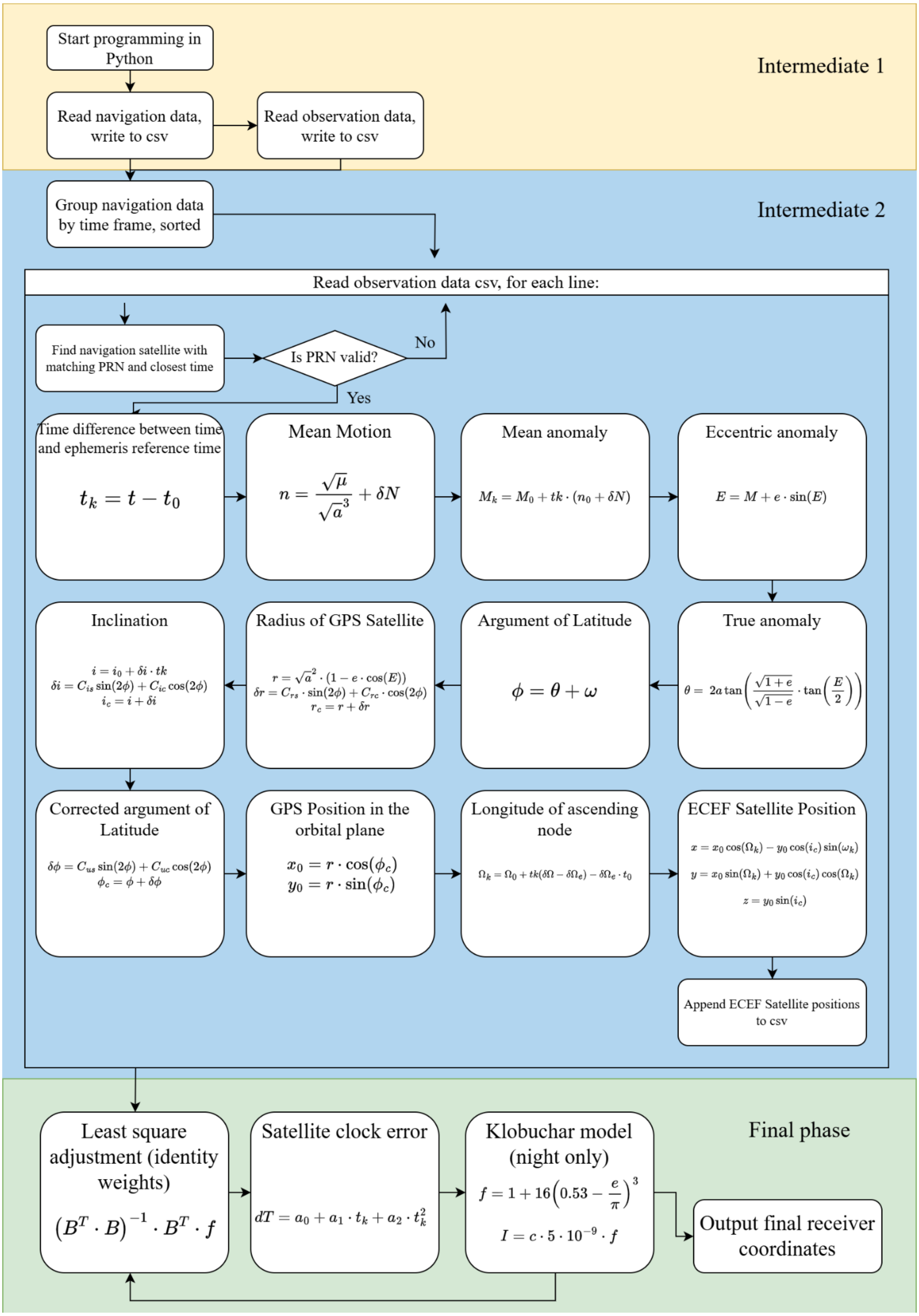
Student ID: 24[redacted]D

Year 3 student of Land Surveying and Geo-informatics

Final Objective

Python is chosen in the development of my computer program (the Program). Given the navigation and observation data files, calculate the GPS receiver position.

Workflow



Modifications to Intermediate 2

1. Calculation of the satellite clock error

```python
def satellite_clock_error(a0, a1, a2, tk) -> float:
    return Consts.c * (a0 + a1 * tk + a2 * tk * tk)
```

The summation of terms of satellite clock offset, drift, and drift rate, with respect to the time difference. Then the time is converted the distance light has travelled in the interval.

2. Adjust the satellite positions with earth’s rotation speed

```python
def correct_earth_rotation(pos: tuple[float, float, float], time_transmission) -> tuple[float, float, float]:
    # Rotated angle ωτ (rad) = rotation speed Ω_e (rad/s) * time (s)
    rotated_angle = Consts.omega_e * time_transmission
    x, y, z = pos
    new_x = x * cos(rotated_angle) + y * sin(rotated_angle)
    new_y = x * -sin(rotated_angle) + y * cos(rotated_angle)
    return new_x, new_y, z
```

The x, y, z offsets in ECEF system, the earth has rotated within the time difference.

3. No longer write intermediate satellite positions to csv

4. Requires `numpy` library installed

Module structure review

`main.py`	main script for running the Python program
`core.py`	data structures for `NavData` and `ObsData`
`consts.py`	define constants
`helper_fn.py`	helper functions related to RINEX file formats
`math_fn.py`	helper functions related to math
`util.py`	miscellaneous helper functions related to data type manipulation (group by duplicates, list iterator, find first)

Detailed workflow and corresponding Python code

<table>
<tr><th>Workflow</th><th>Formula / Constant</th><th>Python Code</th></tr>
<tr><td>Define the necessary constants for the upcoming formulas for orbit calculation.</td><td>$c = 299792458\ m/s$
$g = 9.80665\ m/s^2$
$G = 6.6725 * 10^{11}\ m^3/kg\ s^2$
$M = 5.972 * 10^{24}\ kg$
$\mu = 3.986004418 * 10^{14}\ m^3/s^2$
$\delta\Omega = 7.2921151467 * 10^{-5}\ rad/s$</td><td><pre><code>class Consts:
    c = 299792458   # Speed of light (m/s)
    g = 9.80665   # Acceleration of earth's gravity (m/s^2)
    G = 6.67259e-11   # Universal gravitational constant (m^3/kg s^2)
    M = 5.972e24   # Mass of the Earth (kg)
    mu = 3.986004418e14 # Standard gravitational parameter (μ = G * M) (m^3/s^2)
    omega_e = 7.2921151467e-5 # Earth rotation rate (rad/s)

    half_week = 3.5 * 60 * 60 * 24</code></pre></td></tr>
</table>

Time difference between time and ephemeris reference time, then normalized within [-3.5 day, +3.5 day] range	$t_k = t - t_0$	`def time_diff_k(t: float, t0: float) -> float:`
		`    tk = t - t0`
		`    if tk > Consts.half_week:`
		`        tk -= 2 * Consts.half_week`
		`    elif tk < - Consts.half_week:`
		`        tk += 2 * Consts.half_week`
		`    return tk`

Mean motion (n)	$n = \frac{\sqrt{u}}{\left(\sqrt{a}\right)^3} + \delta n$	`def mean_motion(sqrt_a: float, delta_n: float) -> float:`
		`    # n = sqrt(mu / a^3)`
		`    #   = sqrt(mu) / sqrt(a^3)`
		`    #   = sqrt(mu) / sqrt(a) ^ 3`
		`    # n = n0 + Δn`
		`    return sqrt(Consts.mu) / sqrt_a ** 3 + delta_n`

Mean anomaly (M)	$M_k = M_0 + t_k(n_0 + \delta n)$	`def mean_anomaly(m0: float, n: float, tk: float) -> float:`
		`    # Mk = m0 + tk * (n0 + Δn)`
		`    return m0 + tk * n`

Eccentric anomaly (E)	$E_k = M_k + e \sin(E_k)$	`def ecc_anomaly(m: float, ecc: float) -> float:`
		`    # E = M + e * sin(E)`
		`    # firstly, assume e * sin(M) is very small ~= 0`
		`    # E = M`
		`    # then we can work bottom-up by substituting E to get the new E`
		`    iterations = 30`
		`    e = m`
		`    for _ in range(iterations):`
		`        e = m + ecc * sin(e)`
		
		`    return e`

True anomaly (θ)	$\theta_k = 2 * atan\left(\frac{sqrt(1 + e)}{sqrt(1 - e)} tan\left(\frac{E}{2}\right)\right)$	`def true_anomaly(E: float, ecc: float) -> float:`
		`    # θ = 2 * atan(sqrt(1 + e) / sqrt(1 - e) * tan(E / 2))`
		`    return 2 * atan(sqrt(1 + ecc) / sqrt(1 - ecc) * tan(E / 2))`

Argument of latitude	$\varphi_k = \theta_k + \omega$	`def argument_of_latitude(theta: float, omega: float) -> float:` `    return theta + omega`

Orbit radius of the satellite position (r)	$r = a(1 - e\cos E_k) + \delta r_k$
	$\delta r_k = C_{rs}\sin(2\varphi_k) + C_{ic}\cos(2\varphi_k)$
	$r_c = r_k + \delta r_k$
		`def orbit_radius(sqrt_a, ecc, E, crs, crc, phi) -> float:`
		`    r = sqrt_a * sqrt_a * (1 - ecc * cos(E))`
		`    delta_r = crs * sin(2 * phi) + crc * cos(2 * phi)`
		`    return r + delta_r`

Inclination (i)	$i_k = i_0 + \delta i + \delta i_k * t_k$
	$\delta i_k = C_{is}\sin(2\varphi_k) + C_{ic}\cos(2\varphi_k)$
	$i_c = i_k + \delta i_k$
		`def inclination(i0, d_i, tk, cis, cic, phi) -> float:`
		`    i = i0 + d_i * tk`
		`    delta_i = cis * sin(2 * phi) + cic * cos(2 * phi)`
		`    return i + delta_i`

Corrected Argument of latitude	$\delta\varphi_k = C_{us}\sin(2\varphi_k) + C_{uc}\cos(2\varphi_k)$
	$\varphi_c = \varphi_k + \delta\varphi_k$
		`def argument_of_latitude_corrected(cus, cuc, phi) -> float:`
		`    delta_phi = cus * sin(2 * phi) + cuc * cos(2 * phi)`
		`    return phi + delta_phi`

GPS Position in orbital plane	$x_0 = r_c \cos(\varphi_c)$
	$y_0 = r_c \sin(\varphi_c)$
		`def gps_position_orbital_plane(r, phi_c) -> tuple[float, float]:`
		`    x0 = r * cos(phi_c)`
		`    y0 = r * sin(phi_c)`
		`    return x0, y0`

Longitude of ascending node	$\Omega_k = \Omega_0 + t_k(\delta\Omega - \delta\Omega_e) - \delta\Omega_e * t_0$	`def longitude_of_ascending_node(omega_0, d_omega, tk, t0) -> float:` `    return omega_0 + (d_omega - Consts.omega_e) * tk - Consts.omega_e * t0`

<table>
<tr>
<td>Earth-centered Earth-fixed (ECEF) frame in orbital terrestrial coordinate system</td>
<td>

$x = x_0 \cos(\Omega_k) - y_0 \cos(i_c) \sin(\Omega_k)$

$y = x_0 \sin(\Omega_k) + y_0 \cos(i_c) \cos(\Omega_k)$

$z = y_0 \sin(i_c)$

</td>
<td>

```python
def gps_position_ecef(x0, y0, omega_k, i) -> tuple[float, float, float]:
    x = x0 * cos(omega_k) - y0 * cos(i) * sin(omega_k)
    y = x0 * sin(omega_k) + y0 * cos(i) * cos(omega_k)
    z = y0 * sin(i)
    return x, y, z
```

</td>
</tr>
<tr>
<td>Satellite clock error (dT)</td>
<td>

$dT = c * a_0 + a_1 * t_k + a_2 * t_k^2$

</td>
<td>

```python
def satellite_clock_error(a0, a1, a2, tk) -> float:
    return Consts.c * (a0 + a1 * tk + a2 * tk * tk)
```

</td>
</tr>
<tr>
<td>Earth rotational adjustment</td>
<td>

$\omega\tau = \Omega_e - t_k$

$\begin{bmatrix} \cos(\omega\tau) & \sin(\omega\tau) & 0 \\ -\sin(\omega\tau) & \cos(\omega\tau) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{matrix} X \\ Y \\ Z \end{matrix}$

</td>
<td>

```python
def correct_earth_rotation(pos: tuple[float, float, float], time_transmission) -> tuple[float, float, float]:
    # Rotated angle ωτ (rad) = rotation speed Ω_e (rad/s) * time (s)
    rotated_angle = Consts.omega_e * time_transmission
    x, y, z = pos
    new_x = x * cos(rotated_angle) + y * sin(rotated_angle)
    new_y = x * -sin(rotated_angle) + y * cos(rotated_angle)
    return new_x, new_y, z
```

</td>
</tr>
<tr>
<td>Geometric distance in 3D space (ρ)</td>
<td>

$\rho = \sqrt{\delta x^2 + \delta y^2 + \delta z^2}$

</td>
<td>

```python
def geometric_distance(base_pos: np.array, gps_pos_list: np.ndarray) -> np.ndarray:
    return np.sqrt((base_pos[0] - gps_pos_list[:, 0]) ** 2 +
                   (base_pos[1] - gps_pos_list[:, 1]) ** 2 +
                   (base_pos[2] - gps_pos_list[:, 2]) ** 2)
```

</td>
</tr>
<tr>
<td>Design matrix for least square adjustments</td>
<td>

$\begin{bmatrix} \frac{x_0 - x}{\rho} & \frac{y_0 - y}{\rho} & \frac{z_0 - z}{\rho} & 1 \\ \ldots & \ldots & \ldots & \ldots \end{bmatrix}$

</td>
<td>

```python
def design_matrix(base_pos: np.array, gps_pos_list: np.ndarray, rho: np.ndarray) -> np.ndarray:
    # Or the "unit vector", â = a / |a| where a is dx, dy, dz, 1, and |a| is rho
    rows, _ = gps_pos_list.shape
    b = np.empty((rows, 4), dtype=float)
    b[:, 0] = (base_pos[0] - gps_pos_list[:, 0]) / rho
    b[:, 1] = (base_pos[1] - gps_pos_list[:, 1]) / rho
    b[:, 2] = (base_pos[2] - gps_pos_list[:, 2]) / rho
    b[:, 3] = np.ones_like(rho)
    return b
```

</td>
</tr>
<tr>
<td>Conversion from ECEF to WGS84</td>
<td></td>
<td>

```python
def xyz_to_phi_lambda_h(pos: tuple[float, float, float]) -> tuple[float, float, float]:
    a, f, ecc_2 = WGS84.a, WGS84.f, WGS84.ecc_2
    x, y, z = pos
    p = sqrt(x * x + y * y)
    lam = atan2(y, x)

    r = sqrt(p * p + z * z)
    sin_phi = z / r
    phi = asin(sin_phi)
    h = r - a * (1 - sin_phi * sin_phi * f)
    tolerance = 1e-10

    while True:
        sin_phi = sin(phi)
        cos_phi = cos(phi)
        n = a / sqrt(1 - ecc_2 * sin_phi * sin_phi)  # prime vertical
        dp = p - (n + h) * cos_phi
        dz = z - (n * (1 - ecc_2) + h) * sin_phi
        h += sin_phi * dz + cos_phi * dp
        phi += (cos_phi * dz - sin_phi * dp) / (n + h)
        if (dp * dp + dz * dz) < tolerance:
            break

    return phi, lam, h  # phi and lambda are in radians
```

</td>
</tr>
</table>

<table>
<tr>
<td>Satellite positions in ENU system relative to the tentative receiver location</td>
<td>

$$\begin{bmatrix} E \\ N \\ U \end{bmatrix} = R_x(90^{\circ} - \phi0) R_z(90 + \lambda0) \begin{bmatrix} X - X_{p0} \\ Y - Y_{p0} \\ Z - Z_{p0} \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(90^{\circ} - \phi0) & sin(90^{\circ} - \phi0) \\ 0 & -sin(90^{\circ} - \phi0) & cos(90^{\circ} - \phi0) \end{bmatrix} \begin{bmatrix} cos(90 + \lambda0) & sin(90 + \lambda0) & 0 \\ -sin(90 + \lambda0) & cos(90 + \lambda0) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X - X_{p0} \\ Y - Y_{p0} \\ Z - Z_{p0} \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \sin(\phi0) & cos(\phi0) \\ 0 & -cos(\phi0) & \sin(\phi0) \end{bmatrix} \begin{bmatrix} -sin(\lambda0) & cos(\lambda0) & 0 \\ -cos(\lambda0) & -sin(\lambda0) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X - X_{p0} \\ Y - Y_{p0} \\ Z - Z_{p0} \end{bmatrix}$$

$$= \begin{bmatrix} -sin(\lambda0) & cos(\lambda0) & 0 \\ -\sin(\phi0)cos(\lambda0) & -\sin(\phi0)sin(\lambda0) & cos(\phi0) \\ cos(\phi0)cos(\lambda0) & cos(\phi0)sin(\lambda0) & \sin(\phi0) \end{bmatrix} \begin{bmatrix} X - X_{p0} \\ Y - Y_{p0} \\ Z - Z_{p0} \end{bmatrix}$$

</td>
<td>

```
def gps_satellites_in_enu(base_pos_xyz: np.array,
                          base_pos_phi_lambda_h: tuple[float, float, float],
                          gps_pos_list: np.ndarray) -> np.ndarray:
    # The transformation from global geodetic to local geodetic coordinate system
    # can be calculated as below:
    #
    # [ -sin(λ)        cos(λ)        0      ]   [X_sat - X0]
    # [ -sin(φ)cos(λ) -sin(φ)sin(λ)  cos(φ) ] * [Y_sat - Y0]
    # [  cos(φ)cos(λ)  cos(φ)sin(λ)  sin(φ) ]   [Z_sat - Z0]
    #
    # where X0, Y0, Z0, φ, λ are the base position parameters

    x, y, z = base_pos_xyz[0], base_pos_xyz[1], base_pos_xyz[2]
    phi, lam, _ = base_pos_phi_lambda_h
    sin_phi, cos_phi, sin_lam, cos_lam = sin(phi), cos(phi), sin(lam), cos(lam)
    rows, _ = gps_pos_list.shape
    gps_sat_enu = np.empty((rows, 3), dtype=float)
    gps_sat_enu[:, 0] = ((gps_pos_list[:, 0] - x) * -sin_lam +
                         (gps_pos_list[:, 1] - y) * cos_lam)
    gps_sat_enu[:, 1] = ((gps_pos_list[:, 0] - x) * -sin_phi * cos_lam +
                         (gps_pos_list[:, 1] - y) * -sin_phi * sin_lam +
                         (gps_pos_list[:, 2] - z) * cos_phi)
    gps_sat_enu[:, 2] = ((gps_pos_list[:, 0] - x) * cos_phi * cos_lam +
                         (gps_pos_list[:, 1] - y) * cos_phi * sin_lam +
                         (gps_pos_list[:, 2] - z) * sin_phi)
    return gps_sat_enu
```

</td>
</tr>
<tr>
<td>Elevation angle of satellites relative to the receiver</td>
<td>

$elev = \ atan2(U, \sqrt{E^2 + N^2})$

</td>
<td>

```
def elevation_angle_from_enu(gps_sat_enu: np.ndarray) -> np.ndarray:
    # the elevation angle (radians) can be calculated as follows:
    # d = sqrt(E ^ 2 + N ^ 2)
    # theta = atan(U / d)
    return np.arctan2(gps_sat_enu[:, 2],
                      np.sqrt(gps_sat_enu[:, 0] ** 2 + gps_sat_enu[:, 1] ** 2))
```

</td>
</tr>
<tr>
<td>Klobuchar model for night (Klobuchar coefficients and time of day not needed)</td>
<td>

$f \ = 1 + 16\left(0.53 - \dfrac{elev}{\pi}\right)^3$

$I \ = \ c * f * 5 * 10^{-9}$

</td>
<td>

```
def klobuchar_adjustment(base_pos_xyz: np.array,
                         base_pos_phi_lambda_h: tuple[float, float, float],
                         gps_pos_list: np.ndarray) -> np.ndarray:
    # This is a coarse klobuchar adjustments, which does not account for the time of the day
    # And we use the night model
    # therefore, klobuchar coefficients are not used here
    gps_sat_enu: np.ndarray = gps_satellites_in_enu(base_pos_xyz,
                                                    base_pos_phi_lambda_h,
                                                    gps_pos_list)
    elev: np.ndarray = elevation_angle_from_enu(gps_sat_enu)  # in radians

    # slant factor
    # (radians / pi) is the number of half-turns
    f = 1.0 + 16.0 * (0.53 - elev / pi) ** 3

    # 5 light-nanoseconds slant distance (m)
    return Consts.c * f * 5e-9
```

</td>
</tr>
</table>

GPS Receiver positioning

According to Liu (2025), the formula for pseudorange measurement can be expressed as below:

$$P \ = \ \rho + d\rho + c(dt - dT) + I + T + e_{mp} + n_p$$

Since we assume each satellite has the same weight, the P is an identity matrix and is omitted from the calculation.

$$P \ \approx \ \rho + c(dt - dT) + I$$

The procedure for least square adjustments are described as below:

$$B \ = \begin{bmatrix} \dfrac{x_0 - x}{\rho} & \dfrac{y_0 - y}{\rho} & \dfrac{z_0 - z}{\rho} & 1 \\ \dots & \dots & \dots & \dots \end{bmatrix}$$

$$f \ = \begin{bmatrix} c1 - & \rho + dT \\ & \dots \end{bmatrix}$$

$$adjustments \ = \ (B^T * P * B)^{-1} * B^T * P * f \approx (B^T * B)^{-1} * B^T * f$$

```python
def least_square_adjustment(gps_pos_list: np.ndarray,
                            nav_list,
                            obs_list,
                            c1_list: np.ndarray,
                            sat_clock_err_list: np.ndarray) -> tuple[tuple[float, float, float],
                                                                     tuple[float, float, float]]:
    # We will progressively use more satellites as the accuracy increases
    num_sats = 4  # Minimum satellites required
    total_sats, _ = gps_pos_list.shape
    total_reached = 0

    # Initial guess
    approx_pos = np.array([1.0, 1.0, 1.0], dtype=float)

    rx_clock_dist_prev = 0.0
    rx_clock_dist_adjusted = inf

    iono_adjustments = None  # be aware this must be initialized before use
    adjustments = np.array([inf, inf, inf, 0.0], dtype=float)

    # dx, dy, dz, dt all converge to < 0.1mm, and have used all the least square parameters at least once
    while abs(rx_clock_dist_adjusted) > 1e-4 or total_reached < 2 or np.sum(np.abs(adjustments[:3])) > 1e-4:
        # Recalculate with new clock error
        rx_clock_error = math_fn.transmission_sec(adjustments[3])
        for i in range(4):
            x, y, z, sat_clock_err = calculate_gps_position(nav_list[i], obs_list[i], rx_clock_error)
            gps_pos_list[i, 0] = x
            gps_pos_list[i, 1] = y
            gps_pos_list[i, 2] = z
            sat_clock_err_list[i] = sat_clock_err

        # Actual geometric distance to the satellites
        rho = math_fn.geometric_distance(approx_pos, gps_pos_list[:num_sats])

        # Design matrix
        b = math_fn.design_matrix(approx_pos, gps_pos_list[:num_sats], rho)

        # Error function
        f = c1_list[:num_sats] - rho + sat_clock_err_list[:num_sats]
        if total_reached:
            f += iono_adjustments

        # Least squares 4 unknowns: dx, dy, dz, dt
        # (B^T * B) ^ -1 * B^T * f
        adjustments = np.linalg.inv(b.T @ b) @ b.T @ f
        approx_pos += adjustments[:3]
        rx_clock_dist_adjusted = adjustments[3] - rx_clock_dist_prev
        rx_clock_dist_prev = adjustments[3]

        num_sats *= 4
        # Limit the number of satellites, or jump to total satellites if the error is below 1 meter
        if num_sats > total_sats or abs(rx_clock_dist_adjusted) < 1:
            num_sats = total_sats
            base_pos = approx_pos[0], approx_pos[1], approx_pos[2]
            base_pos_wgs = math_fn.xyz_to_phi_lambda_h(base_pos)
            iono_adjustments = math_fn.klobuchar_adjustment(base_pos,
                                                            base_pos_wgs,
                                                            gps_pos_list)
            total_reached += 1

    x, y, z = approx_pos[0], approx_pos[1], approx_pos[2]
    phi, lam, h = math_fn.xyz_to_phi_lambda_h((x, y, z))

    return (x, y, z), (phi, lam, h)
```

The least squares approach first starts with an approximate position and no receiver clock error. After calculating the GPS satellite positions and their clock errors, we can obtain the design matrix and the error function. Then applying the least squares matrix formula and solve for the adjustments for the approximated position and the receiver clock error. In the last iteration, also account for the ionospheric delay from the Klobuchar model.

How to run

run `python main.py` in the terminal in the same working directory as the python files. This program requires at least Python 3.7 to run. Ensure that site0900.01n and site0900.01o are also in the working directory, and the `numpy` library is installed.

Results

```
PS C:\[redacted]\Desktop\Satellite Positioning> python main.py
Successfully wrote navigation data to site0900-01n.csv
First observation time: 2001-03-31 00:00:00
Successfully wrote observation data to site0900-01o.csv
Successfully calculated the GPS receiver position!

ECEF Coordinates
(-2341338.210, -3539062.216, 4745807.234)

WGS84 Datum
Latitude: 48.389785496
Longitude: -123.487434965
Height: 52.602

Paste these coordinates in Google map:
(48°23'23.22779", -123°29'14.76587")
Time elapsed: 0.722s
PS C:\[redacted]\Desktop\Satellite Positioning>
```

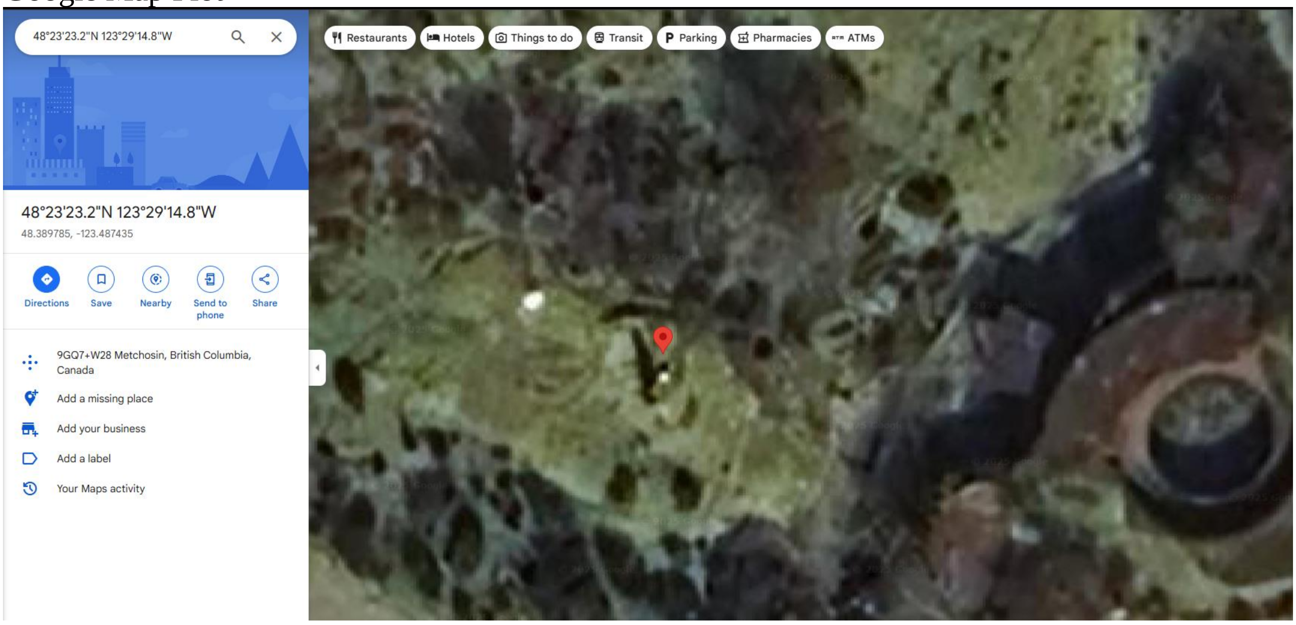
The Program should write the navigation and observation data csv to files, and calculate the coordinates as shown in the console, with sub-second performance.

Information of the calculated location

9GQ7+W28 Metchosin, British Columbia, Canada

ECEF	WGS84
(-2341388.210, -3539062.216, 4745807.234)	(48 °23’23.22779”, -123°29’14.76587”, 52.602)

Google Map Plot



USGS Aerial photo

Improvements made

Instead of iteratively use all the satellites for least-squares adjustments, only 4 are used initially, and as the accuracy increases, more satellites are used. This approach has saved the computation time for matrix multiplications and yields the same results.

Here are the steps of this gradual approach:

- start with 4 satellites
- assume ionospheric delay is 0
- adjust the approximate location and receiver clock error with the least square solutions
- multiply the satellite count by 4, within the total satellite count
- use all the satellites if the improved accuracy of the receiver clock error is less than 1 meter
- recalculate, if all the satellite is used, calculate one last time with the Klobuchar model.

Improvements that can be made

1. Error modeling

The Program only models 3 errors: satellite clock error, receiver clock error, and ionospheric delay. The ionospheric delay uses the simplified version of the Klobuchar model, which does not account for the time of the day. Although some calculation steps are omitted, the performance is at the cost of less vigorous adjustments applied to the ionospheric delay. It is also possible to apply dual frequency adjustments to cancel the ionospheric delay, or use the other models such as the Hopfield Model for mitigating the tropospheric delay. Due to the complexity of the algorithms, only the Klobuchar model is implemented, and only considering the night model.

2. Weight matrix

The Program can also give more weights to different satellites to enhance the least squares calculation, for example give more weight to the satellites with larger elevation angle.

3. I/O Tasks

Furthermore, the Program can be customized whether to write the intermediate results to the csv files.

4. Multiprocessing

The Program can leverage more CPU for computational power, especially to process more than 20000 entries.

Appendix – Source code excerpt for orbit calculation (4 pages)

```python
def calculate_gps_position(nav_data: NavData,
                           obs_data: ObsData,
                           rx_clock_error=0.0) -> tuple[float, float, float, float]:
    # Time from ephemeris reference epoch (tk)
    transmission_sec = math_fn.transmission_sec(obs_data.c1)
    t = obs_data.time_num - transmission_sec - rx_clock_error
    tk = math_fn.time_diff_k(t, nav_data.t0)
    satellite_clock_error = math_fn.satellite_clock_error(nav_data.sv_clock_bias,
                                                          nav_data.sv_clock_drift,
                                                          nav_data.sv_clock_drift_rate, tk)

    # Mean motion (n)
    n = math_fn.mean_motion(nav_data.sqrt_a, nav_data.delta_n)

    # Mean anomaly (M)
    M = math_fn.mean_anomaly(nav_data.m0, n, tk)

    # Eccentric anomaly (E)
    E = math_fn.ecc_anomaly(M, nav_data.ecc)

    # True anomaly (θ)
    theta = math_fn.true_anomaly(E, nav_data.ecc)

    # Argument of latitude (φ)
    phi = math_fn.argument_of_latitude(theta, nav_data.omega)

    # Orbit radius of GPS satellite position (r)
    r = math_fn.orbit_radius(nav_data.sqrt_a, nav_data.ecc, E,
                             nav_data.crs, nav_data.crc, phi)

    # Corrected GPS orbit inclination (i)
    i = math_fn.inclination(nav_data.i0, nav_data.d_i, tk,
                            nav_data.cis, nav_data.cic, phi)

    # Corrected argument of latitude
    phi_c = math_fn.argument_of_latitude_corrected(nav_data.cus, nav_data.cuc, phi)

    # GPS Positions in the orbital plane (φ)
    x0, y0 = math_fn.gps_position_orbital_plane(r, phi_c)

    # Longitude of ascending node (Ω_k)
    omega_k = math_fn.longitude_of_ascending_node(nav_data.omega_0, nav_data.d_omega, tk, nav_data.t0)

    # GPS Positions in ECEF
    pos = math_fn.gps_position_ecef(x0, y0, omega_k, i)

    # Correct GPS Satellite positions with earth's rotation
    x, y, z = math_fn.correct_earth_rotation(pos, transmission_sec)

    return x, y, z, satellite_clock_error
```

Calculation workflow function

```python
def process_gps_position(nav_data_list: list[NavData], obs_file, number_of_obs: int):
    # The navigation file is always sorted by time, record the ranges of timeframes with slices
    grouped_nav_data_list, nav_slice_list = group_by_duplicates(iter(nav_data_list),
                                                                lambda nav1, nav2: nav1.time() == nav2.time())
    nav_time_gen = (nav_data.time() for nav_data in grouped_nav_data_list)
    # Maps time to slice of original list
    time_map = [(t, s) for t, s in zip(nav_time_gen, nav_slice_list)]

    gps_pos_list = np.empty((number_of_obs, 3), dtype=float)
    nav_list = [None] * number_of_obs
    obs_list = [None] * number_of_obs
    c1_list = np.empty(number_of_obs, dtype=float)
    sat_clock_err_list = np.empty(number_of_obs, dtype=float)

    for index, obs_row in enumerate(obs_file):
        obs_data = ObsData.from_csv_row(obs_row)
        obs_time = obs_data.time()
        # Sort time-slice map by closest time first
        time_map.sort(key=lambda item: abs(item[0] - obs_time))
        match_nav_data = None
        for _, s in time_map:
            # Linear search within the same time frame
            match_nav_data = find_first(list_iter(nav_data_list, s), lambda nav_data: nav_data.prn == obs_data.prn)
            if match_nav_data is not None:
                break

        # No satellites with matching PRN found
        if match_nav_data is None:
            continue
        # Found a matching satellite with matching PRN, but expired
        if abs(match_nav_data.time() - obs_time) >= timedelta(hours=4):
            continue
```

```python
        x, y, z, sat_clock_err = calculate_gps_position(match_nav_data, obs_data)
        gps_pos_list[index, 0] = x
        gps_pos_list[index, 1] = y
        gps_pos_list[index, 2] = z
        nav_list[index] = match_nav_data
        obs_list[index] = obs_data
        c1_list[index] = obs_data.c1
        sat_clock_err_list[index] = sat_clock_err

    ecef_pos, wgs84_pos = least_square_adjustment(gps_pos_list, nav_list, obs_list, c1_list, sat_clock_err_list)
    pretty_print_results(ecef_pos, wgs84_pos)
```

The function that processes the GPS Positions

```python
def group_by_duplicates(iterator, equals_fn):
    result_list = []
    result_slices = []
    start_idx = 0
    first_item = next(iterator)  # Get the first item from the iterator
    result_list.append(first_item)  # Always append the first item
    prev_item = first_item  # Set prev_item to the first item

    for i, item in enumerate(iterator, start=1):  # Start enumeration from 1
        if not equals_fn(item, prev_item):
            result_slices.append(slice(start_idx, i))
            start_idx = i
            result_list.append(item)
            prev_item = item

    # Append the last slice for the final group
    result_slices.append(slice(start_idx, len(result_list)))

    return result_list, result_slices

def list_iter(ref_list, slice_obj):
    return (ref_list[index] for index in range(slice_obj.start, slice_obj.stop))

def find_first(iterable, predicate):
    for item in iterable:
        if predicate(item):
            return item
    return None
```

```python
def pretty_format_degree(deg) -> str:
    positive = deg >= 0
    sign = '' if positive else '-'
    deg = abs(deg)
    degrees = int(deg)
    minutes = int((deg - degrees) * 60)
    seconds = ((deg - degrees) * 60 - minutes) * 60

    return f"{sign}{degrees}°{minutes}'{seconds:.5f}\""
```

Utility functions used

```python
def pretty_print_results(ecef_pos: tuple[float, float, float],
                         wgs84_pos: tuple[float, float, float]):
    print("Successfully calculated the GPS receiver position!")
    print()
    x, y, z = ecef_pos
    print(f"ECEF Coordinates\n"
          f"({x:.3f}, {y:.3f}, {z:.3f})")
    print()
    phi, lam, h = wgs84_pos
    phi = degrees(phi)
    lam = degrees(lam)
    print(f"WGS84 Datum\n"
          f"Latitude: {phi:.9f}\n"
          f"Longitude: {lam:.9f}\n"
          f"Height: {h:.3f}\n"
          f"\n"
          f"Paste these coordinates in Google map:\n"
          f"({pretty_format_degree(phi)}, {pretty_format_degree(lam)})")
```

Pretty print results

```python
def main():
    read_nav()
    read_obs()
    with open(OBS_CSV_FILE, newline='') as obs_file:
        obs_file = csv.reader(obs_file)
        next(obs_file)  # exclude header from csv
        process_gps_position(obs_file)

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

Updated main function

References

Calais E. *GPS Geodesy – Lab 5 From GPS ephemerides to ECEF satellite positions*. Retrieved from https://www.geologie.ens.fr/~ecalais/teaching/gps-geodesy/lab_5.pdf

Crocetto N., Gatti M. & Perfetti N. (1997) *The GPS Single Point Positioning: A Data Processing Program for Tutorial Purposes*. ISPRS Commision VI, Working Group 3. 1-5 Retrieved from https://www.isprs.org/proceedings/XXXII/6-W1/131_XXXII-6-W1.pdf

Gurtner W. & Mader G. (1990) *RINEX: The Receiver Independent Exchange Format Version 2*. CSTG GPS Bulletin. Vol. 3. Retrieved from https://www.ngs.noaa.gov/CORS/RINEX-2.txt

Li Y., Chang, X., Yu, K., Wang, S., & Li, J. (2019). *Estimation of snow depth using pseudorange and carrier phase observations of GNSS single-frequency signal*. GPS Solutions, 23(4), 1–13.

Retrieved from https://doi.org/10.1007/s10291-019-0912-5

Liu, Z. Z. (2025) *Errors Modeling* Retrieved from https://learn.polyu.edu.hk/bbcswebdav/pid-7275710-dt-content-rid-63241790_1/xid-63241790_1

Liu, Z. Z. (2025) *Handout – V08 Satellite Positioning George Lec 5 2025 Feb.* 1-41. Retrieved from https://learn.polyu.edu.hk/bbcswebdav/pid-7319802-dt-content-rid-63999154_1/xid-63999154_1

Mussio L., Crippa B. & Vettore A. (1997) *International Society for Photogrammetry and Remote Sensing* ISPRS 32:1-11. Retrieved from https://www.researchgate.net/profile/Gabriella-Caroti/publication/260934448_Kinematic_GNSS_SurveyExperiences_to_be_Transferred/links/58f759d50f7e9be34b3426db/Kinematic-GNSS-SurveyExperiences-to-be-Transferred.pdf

Xu, G. & Xu, Y., *GPS: Theory, Algorithm and Applications*. Berlin, Heidelberg: Springer Berlin, 2016. Retrieved from https://link.springer.com/book/10.1007/978-3-662-50367-6

Zhao, S., Cui, X., & Lu, X., *Single point positioning using full and fractional pseudo range measurements*

from GPS and BDS, Survey review - Directorate of Overseas Surveys, vol. 53, no. 376, pp. 27–34, 2021, Retrieved from https://www.researchgate.net/publication/336977979_Single_point_positioning_using_full_and_fractional_pseudorange_measurements_from_GPS_and_BDS