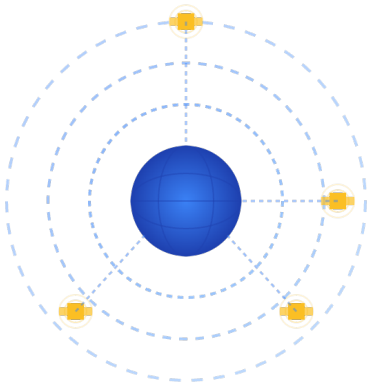


GNX-py User Manual

GNSS data processing and modelling toolkit

Hubert Pierzchała

January 8, 2026

Space Research Centre
Polish Academy of Sciences

Contents

GNX-py User Manual

This document provides an overview of the **GNX-py** library and its main functional blocks: **Precise Point Positioning (PPP)**, **orbit analysis** and **ionosphere modelling**.

The manual is structured in parallel to the Jupyter notebooks shipped with the project:

- **Introduction** – installation, high-level concepts, simple processing example
- **Precise Point Positioning (PPP)** – example of PPP with GPS and Galileo in IF and UDUC modes
- **Orbit module** – broadcast / precise orbits comparison, SISRE analysis
- **Ionosphere** – TEC processing, ionosphere mapping and monitoring

Each chapter in this PDF corresponds to one or more tutorial notebooks. For users who prefer an interactive environment, the same content is available as `.ipynb` files in the `tutorials/` folder of the project.

```
import gps_lib as gnx
import numpy as np
from utils import df_head
```

GNX-py library

GNX-py is a Python library designed to be a convenient tool for processing GNSS data. This project will undergo ongoing development in the coming months, expanding individual segments of the library and adding new ones. Currently, the library contains three main modules: PPP, Orbits, and Ionosphere. A description and example of how to use these modules can be found in dedicated Jupyter notebooks. To simplify the use of the program for users who do not necessarily want to build their own Python scripts, along with the library, you have received ready-made scripts that allow you to perform tasks covering the entire spectrum of currently available functionalities. We will describe them at the end of this chapter.

In this chapter, I will introduce you to the functionalities of the GNX-py library using the example of preprocessing observation data. We will go through the main tools of the library, from data loading to obtaining dataset fully prepared for positioning. At each step, I will explain the most important information about the current activities, but, unsurprisingly, I will not comment on every line of code, as this would be too time-consuming and pointless. If you are not familiar with Python and are only interested in using scripts for your studies or research, you may be more interested in using the ready-made scripts I mentioned. However, if you want to know exactly how certain steps were implemented, I encourage you to review the code. If you have any questions, please feel free to contact me on GitHub or by email.

Installation

The best way to install the software is to clone the repository directly from GitHub using git clone. Next, it is recommended to create and activate a virtual Python environment, and then execute the pip install -e command within it, which will install the library in developer mode based on the included setup.py script. This installation method allows you to easily modify the source code and immediately use the changes without reinstalling.

```
# Clone the repository
git clone https://github.com/HJpierzchala/GNX.git
cd GNX

# Create and activate a virtual environment (Python  3.12)
python3 -m venv venv
source venv/bin/activate    # Linux / macOS

# Windows
py -3.12 -m venv .venv
.venv\Scripts\activate
python -m pip install -U pip

Option A (recommended, reproducible)
pip install -e . -c constraints-win.txt

Option B (lets pip choose versions inside allowed ranges)
pip install -e .

#MacOS
python3.12 -m venv .venv
source .venv/bin/activate
python -m pip install -U pip

Option A (recommended, reproducible)
pip install -e . -c constraints-macos.txt

Option B (lets pip choose versions inside allowed ranges)
pip install -e .
```

> **Note:** Installing in editable mode (`-e`) allows you to modify the source code and immediately use the changes without reinstalling the package.

Example of processing in GNX-py

Preparing observations for positioning requires a series of steps. In general, the task can be summarized as follows:

We want to obtain:

- satellite coordinates (more precisely, satellite antenna phase centre (APC) or centre of mass (COM, with the appropriate correction) in the ECEF system at the time of signal transmission from the satellite;
- corrections for each observation, such as ionospheric, tropospheric, and relativistic correction;
- optionally, a sanity check of the observations: elevation mask and cycle slips detection for phase observations

Let's use the simplest example of preparing observations for single point positioning (SPP) using GPS with a broadcast orbit. The main class in GNX-py that allows you to load and parse most of the file formats used by the library is GNSSDataProcessor. First, let's gather the files we need: observation and navigation RINEX files, and an ATX file from which we will obtain the receiver's PCO.

```
import os
OBS = '../data/BOR100POL_R_20240350000_01D_30S_MO.rnx'
NAV='../data/BRDC00IGS_R_20240350000_01D_MN.rnx'
ATX='../data/ngs20.atx'
```

GNSSDataProcessor takes as arguments the paths to your input files and information about the system and signals you want to use. The arguments *obs_path*, *nav_path*, and *atx_path* are, of course, the aforementioned file paths. The argument *sys* is the code of the system you want to use: G-GPS, E-Galileo. Argument *mode* is the code of the signal you want to use. The available options are GPS: L1, L2, L5, L1L2, for Galileo: E1, E5a, E5b, E1E5a, E1E5b. Currently, this set of signals and their combinations has been implemented and tested. Work is underway on the implementation of triple frequency processing and positioning.

```
processor = gnx.GNSSDataProcessor2(obs_path=OBS,nav_path=NAV,atx_path=ATX
    ,sys={'G'},mode='L1L2', use_gfz=True)
```

Let's pause for a moment on the use_gfz argument. The main method for opening RINEX files in GNX-py is to use the georinex library ((Hirsch et al. 2023)). This is a well-maintained and reliable library. It allows for efficient loading of navigation and observation files into the xarray.dataset format, from which it is easy to switch to other Python structures. However, RINEX 3 files, which are currently the primary format supported by GNX-py, are processed quite slowly by georinex, as the author of the library himself informs on the project's GitHub page. By default, when you install all dependencies for GNX-py, you will also have access to georinex. However, if you have GFZRNX installed on your computer ((Nischan 2016)) you can integrate it with GNX-py processing by setting $use\_gfz = True$. Just make sure that your GFZRNX is aliased to "gfzrnx". Using this parameter, GNSSDataProcessor will try to load the file by first converting it to a .tab file using GFZRNX. This approach allows you to

speed up the loading of observation data slightly. If your observation Rinex is in .crx format and you want to select the use_gfz=True option, make sure you also have the RNXCMP tool ((Hatanaka 2008))on your computer to convert the file to .rnx, as this is the input format for GFZRNX. Georinex supports both .rnx and .crx files, as well as compressed .crx.gz files.

```
obs_data = processor.load_obs_data()
```

Using the lead_obs_data() method, we load observational data and additional information. Let's take a look at them now.

```
from dataclasses import fields
for f in fields(obs_data):
    print(f.name, f.type)
```

```
gps <class 'pandas.core.frame.DataFrame'>
gal <class 'pandas.core.frame.DataFrame'>
sat_pco <class 'collections.defaultdict'>
rec_pco <class 'collections.defaultdict'>
meta <class 'tuple'>
interval <class 'float'>
```

We obtained the obs_data structure containing the following fields: - gps - gal - sat_pco - rec_pco - meta - interval.

The *gps* and *gal* attributes are pandas dataframes with the *sv* and *time* pandas.MultIindex, where *sv* is the satellite PRN and *time* is the observation time from the RINEX file. We selected the L1L2 signals, so we see the L, C, and S columns. GNX-py inspects all observations contained in the loaded file and selects those to be loaded based on the mode, the number of observations, and their type according to the RINEX convention. For example, for GPS, the observation selection priority is: "W", "C", "P", "Y", 'L', "X". In practice, with an equal number of C1C and C1W observations, the program will load C1W, similarly in the case of phase observations and at other frequencies.

```
gps_obs = obs_data.gps.copy()
```

```
df_head(gps_obs, floatfmt=".2f",nrows=3,ncols=6, truncate_str=15)
```

sv	time	S1C	L1C	C1C	S2W
G02	2024-02-04 00:00:00	48.00	115266861.04	21934554.87	37.20
G02	2024-02-04 00:00:30	47.90	115336484.01	21947803.49	37.10
G02	2024-02-04 00:01:00	47.50	115406468.59	21961121.43	37.10

Another element of interest is the *sat_pco* attribute. It contains PCO corrections for satellites read from the ATX file. They are used in PPP and optionally

in orbit analysis. They are stored as a defaultdict structure with keys (PRN, SIGNAL) and values in a list, where PRN is the satellite's PRN code, Signal is the frequency code according to the ANTEX convention, e.g., G01-L1 frequency, G02-L2 frequency, etc. We will not use these values in our example, as we will be using broadcast orbits, in which the interpolated coordinates already refer to the satellite's APC.

```
sat_pco = obs_data.sat_pco

for i, (k,v) in enumerate(sat_pco.items()):
    if i == 3: break
    print(k, v)
```

```
('E01', 'E05') [122.14, -9.38, 792.04]
('E01', 'E07') [122.84, -8.98, 831.06]
('E01', 'E06') [122.04, -9.21, 845.56]
```

You can view the receiver's PCO in a similar structure. Information about the name and antenna cover is read from the header of the rinex file. The PCO is stored in defaultdict, where the key is a tuple (NAME, COVER, SIGNAL), where the signal, as in the case of satellite PCO, is a frequency code compliant with the RINEX convention. It is important to note that for some receivers, the ATX file used may not contain PCO information for a given frequency. This does not happen often, but if it does, we simply assume a zero offset. This may affect positioning accuracy, especially in the vertical plane.

```
rec_pco = obs_data.rec_pco
for i, (k, v) in enumerate(rec_pco.items()):
    if i == 3: break
    print(k, v)
```

```
('TRM59800.00', 'NONE', 'G01') [0.65, 1.35, 89.35]
('TRM59800.00', 'NONE', 'E01') [0.65, 1.35, 89.35]
('TRM59800.00', 'NONE', 'J01') [0.65, 1.35, 89.35]
```

Meta field contains a tuple with additional, but no less important, information about the observation data and the receiver. It includes, in order: - station name - receiver name - receiver cover type - antenna height in order $dUEN$ - Up, East, North [m] - approximate coordinates in ECEF - approximate coordinates in the geodetic system - interval in seconds - GPS observation types - Galileo observation types - first observation epoch - last observation epoch - phase shifts for observations

Important elements of the list are antenna height, approximate coordinates, and phase shifts. The first two elements are used directly in the estimation process as antenna height correction and a-priori coordinates, while phase shifts are necessary for preprocessing phase observations in PPP. Under the interval

argument, we find the interval in minutes. The code searches for it in the header, but in case it's missing, it also retrieves it from the file name.

```
meta = obs_data.meta
for i in meta:
    print(i)
print(obs_data.interval)
```

```
BOR1
TRM59800.00
NONE
[0.0622 0.     0.    ]
[3738358.5958 1148173.5785 5021815.7483]
[ 52.27695597  17.07345397 124.40446512]
30.0
['C1C', 'C2W', 'C2X', 'C5X', 'L1C', 'L2W', 'L2X', 'L5X', 'S1C', 'S2W', '
    S2X', 'S5X']
['C1X', 'C5X', 'C7X', 'C8X', 'L1X', 'L5X', 'L7X', 'L8X', 'S1X', 'S5X', '
    S7X', 'S8X']
2024-02-04 00:00:00
2024-02-05 00:00:00
{'R L1P': 0.25, 'R L2C': -0.25, 'R L2P': 0.0, 'G L2X': -0.25, 'J L2X':
    -0.25}
0.5
```

In this way, we have loaded the observation data and all the necessary additional information and parameters. Now let's load the navigation data. We use *load_broadcast_orbit*() for this.The resulting structure contains: - gps_orb - broadcast orbit for GPS - gal_orb - broadcast orbit for Galileo, if used - gpsa, gpsb, gala - ionospheric model parameters for GPS and Galileo

```
nav_data = processor.load_broadcast_orbit()
```

```
for f in fields(nav_data):
    print(f.name, f.type)
```

```
gps_orb typing.Optional[pandas.core.frame.DataFrame]
gal_orb typing.Optional[pandas.core.frame.DataFrame]
gpsa float | None
gpsb float | None
gala float | None
```

The GPS orbit, which is our source of information about the position of satellites, contains elements of the Keplerian orbit and satellite clock parameters, as well as some additional information, including the "health" of the satellite, i.e., its current status. A healthy satellite has a value of 0 in the health column of our dataframe. You can filter the data using:

yourorb[yourorb['health']==0]

Typically, orbits are updated every two hours and are useful within a range of 2-4 hours from the reference epoch of the message. In GNX-py, for each observation, we search for the message closest to its time with a tolerance of 2 hours.

```
gps_orb = nav_data.gps_orb.copy()
df_head(gps_orb, floatfmt=".2f",nrows=3,ncols=6, truncate_str=10)
```

sv	time	SVclockBias	SVclockDrift	SVclockDriftRate	IODE
G08	2024-02-03 23:59:44	-0.00	-0.00	0.00	15.00
G29	2024-02-03 23:59:44	-0.00	0.00	0.00	23.00
G31	2024-02-03 23:59:44	-0.00	-0.00	0.00	0.00

Satellite coordinates are obtained by converting Keplerian orbital elements to ECEF coordinates. The complete algorithm and explanation of symbols are presented below.

Broadcast coordinates interpolation algorithm:

$$
\begin{aligned}
t_k &= t - t_{oe} \quad \text{(apply GPS week crossover)} \\
M_k &= M_0 + \left(\sqrt{\mu/a^3} + \Delta n\right) t_k \\
\text{solve } M_k &= E_k - e \sin E_k \\
v_k &= 2 \tan^{-1}\left(\sqrt{\frac{1+e}{1-e}} \tan \frac{E_k}{2}\right) \\
u_k &= \omega + v_k + c_{uc} \cos 2(\omega+v_k) + c_{us} \sin 2(\omega+v_k) \\
r_k &= a(1 - e \cos E_k) + c_{rc} \cos 2(\omega+v_k) + c_{rs} \sin 2(\omega+v_k) \\
i_k &= i_0 + \dot{i}\, t_k + c_{ic} \cos 2(\omega+v_k) + c_{is} \sin 2(\omega+v_k) \\
\lambda_k &= \Omega_0 + (\dot{\Omega} - \omega_E) t_k - \omega_E t_{oe} \\
\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} &= \mathbf{R}_3(-\lambda_k)\, \mathbf{R}_1(-i_k)\, \mathbf{R}_3(-u_k) \begin{bmatrix} r_k \\ 0 \\ 0 \end{bmatrix}
\end{aligned}
$$

$$
\begin{aligned}
dt^{sat} &= a_0 + a_1(t - t_0) + a_2(t - t_0)^2 + \Delta t_{rel} \\
\Delta t_{rel} &= -2 \cdot \frac{r_{sat} \cdot v_{sat}}{c^2}
\end{aligned}
$$

Where:

t : epoch of interest (GNSS system time)

t_0 : satellite clock reference epoch

t_{oe} : ephemeris reference epoch

$t_k = t - t_{oe}$ (apply GPS/Galileo week crossover, e.g., $t_k \in [-302400, 302400]$)

M_0 : mean anomaly at t_{oe}

Δn : correction to computed mean motion (rad/s)

μ : Earth GM (m^3/s^2); GPS uses 3.986005×10^{14}

a : semi-major axis

e : orbital eccentricity

E_k : eccentric anomaly solving $M_k = E_k - e \sin E_k$

v_k : true anomaly

ω : argument of perigee

c_{uc}, c_{us} : corrections to argument of latitude (rad)

c_{rc}, c_{rs} : corrections to orbital radius (m)

c_{ic}, c_{is} : corrections to inclination (rad)

$u_k = \omega + v_k + c_{uc} \cos 2(\omega + v_k) + c_{us} \sin 2(\omega + v_k)$ (corrected arg. of latitude)

$r_k = a(1 - e \cos E_k) + c_{rc} \cos 2(\omega + v_k) + c_{rs} \sin 2(\omega + v_k)$ (corrected radius)

i_0 : inclination at t_{oe} (rad)

$\dot{i}$: inclination rate (rad/s)

$i_k = i_0 + \dot{i}\, t_k + c_{ic} \cos 2(\omega + v_k) + c_{is} \sin 2(\omega + v_k)$ (corrected inclination)

Ω_0 : LAN at t_{oe} (Longitude of ascending node, rad)

$\dot{\Omega}$: LAN rate (rad/s)

ω_E : Earth rotation rate $\approx 7.2921151467 \times 10^{-5}$ rad/s

$\lambda_k = \Omega_0 + (\dot{\Omega} - \omega_E)t_k - \omega_E t_{oe}$ (node longitude used in ECEF rotation)

r_{sat} : satellite position vector (m)

v_{sat} : satellite velocity vector(m/s)

c : speed of light

In GNX-py, the above algorithm is used twice. First, we interpolate the coordinates at the time of signal reception, i.e., at the observation timestamp, to obtain the satellite clock correction along with the relativistic correction. Then, using the formula below, we calculate the signal transmission time, i.e., the time it took for the signal to travel from the satellite to the receiver.

$$
\begin{aligned}
R &= c(t_{\text{rcv}}[\text{reception}] - t^{sat}[\text{emission}]) \\
t^{sat}[\text{emission}] &= t_{\text{rcv}}[\text{reception}] - R/c \\
T[\text{emission}] &= t_{\text{rcv}}[\text{reception}] - R/c - dt^{sat}
\end{aligned}
$$

Where R, dt^{sat} and c are the measured pseudorange , satellite clock offset and speeed of light, respectively. Then, the coordinates are calculated again at the time of signal emission, substituting the signal emission epoch for t. We do this using the *BroadcastInterp* class from the coordinates module. This class takes as arguments an observation dataframe - *obs*, a navigation dataframe - *nav*, *mode*, system code - *sys*, and information on whether we want to interpolate at broadcast time or not.

```
interp = gnx.BroadcastInterp(obs=gps_obs, nav=gps_orb,sys='G',mode='L1',
    emission_time=True)
obs_crd = interp.interpolate()
```

Let's see what we get. After executing the interpolate() method, we obtain an observational dataframe enriched with a navigation message assigned to each observation, as well as the x, y, z coordinates of the satellites, clk, i.e., the satellite clock offset, and the velocities vx, vy, vz.

```
df_head(obs_crd[['x','y','z','C1C','C2W']], floatfmt=".2f",nrows=3,ncols
    =7, truncate_str=10)
```

sv	time	x	y	z	C1C	C2W
G02	2024-02-04 00:00:00	21576412.43	12769386.38	9640929.69	21934554.87	21934557.32
G02	2024-02-04 00:00:30	21595795.21	12798191.22	9555185.53	21947803.49	21947806.25
G02	2024-02-04 00:01:00	21615007.01	12826735.96	9469264.45	21961121.43	21961123.92

However, that is not all there is to satellite coordinates. Since we are talking about coordinates in the ECEF system, we need to transform the satellite coordinates to the system during signal transmission. In other words: during signal transmission, the satellite has moved, so we calculate the signal propagation time and coordinates during transmission, and at the same time, during the aforementioned propagation, the system has also rotated, so we correct it. The point is to recreate the receiver-satellite system as accurately as possible, in terms of time and in the system related to the satellite's signal transmission. The equation looks like this:

$$\tilde{\mathbf{r}}^{\,sat} \equiv \text{satellite ECEF at emission (tied to Earth at } T_{\text{em}})$$

$$\Delta t = \frac{\|\tilde{\mathbf{r}}^{\,sat} - \mathbf{r}_{rcv}\|}{c}$$

$$\mathbf{r}^{sat} = \mathbf{R}_3(\omega_E\,\Delta t)\,\tilde{\mathbf{r}}^{\,sat}, \quad \mathbf{R}_3(\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Where:

r_{rcv} - station position vector\

c - speed of light\

ω_E - Earth rotation rate\

Δt - propagation time\

$\mathbf{r}^{sat}$ - satellite ECEF coordinates at emission time\

$\mathbf{R}_3(\theta)$ - rotation matrix

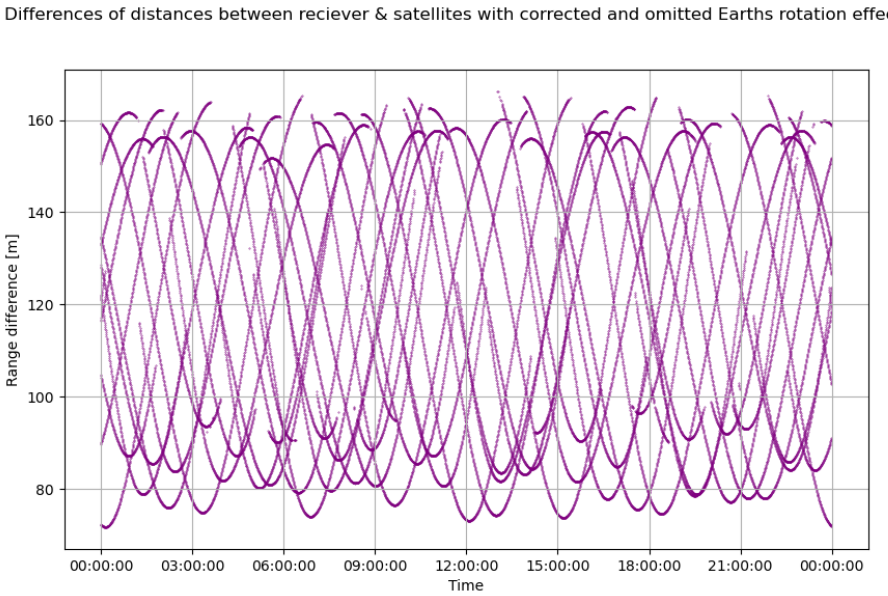
The most convenient way to do this is to use the wrapper from the coordinates.py module - CrdWrapper. This class will apply the above correction and calculate the elevation and azimuth to the satellites.

```
wrapper = gnx.CustomWrapper(obs=obs_crd,flh=obs_data.meta[5],xyz_a=
    obs_data.meta[4],mode='L1L2',epochs=None)
obs_crd=wrapper.run()
```

We obtain additional columns: xe, ye, ze, az, and ev, i.e., corrected coordinates for signal emission time, azimuth, and elevation to satellites.

```
dPos = (obs_crd[['xe','ye','ze']] - obs_crd[['xe_ur','ye_ur','ze_ur']].
    values).rename(columns={'xe':'dx','ye':'dy','ze':'dz'})

dPos['dr'] = np.sqrt(dPos['dx']**2 + dPos['dy']**2 + dPos['dz']**2)
```

```
from matplotlib import pyplot as plt
from matplotlib import dates as mdates
fig, ax = plt.subplots(1,1,figsize=(10,6))
fig.suptitle('Differences of distances between reciever & satellites with
     corrected and omitted Earths rotation effect')
for ind, gr in dPos.groupby('sv'):
    gr = gr.reset_index()
    ax.scatter(gr['time'], gr['dr'], label=ind,s=0.1,c='purple')
ax.set_ylabel('Range difference [m]')
ax.set_xlabel('Time')
ax.grid()
plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%H:%M:%S'))
```




```
df_head(obs_crd[['xe','ye','ze','ev','az']],nrows=3,ncols=10,
    truncate_str=10,floatfmt='.1f')
```

sv	time	xe	ye	ze	ev	az
G02	2024-02-04 00:00:00	21576480.1	12769272.0	9640929.7	47.9	156.3
G02	2024-02-04 00:00:30	21595863.1	12798076.7	9555185.5	47.7	156.3
G02	2024-02-04 00:01:00	21615075.1	12826621.3	9469264.5	47.5	156.4

At this point, we have already developed most of the elements needed for positioning: satellite coordinates and clocks, elevation and azimuth. Now let's filter out low-elevation satellites and calculate the ionospheric and tropospheric corrections. Three ionosphere models have been implemented in GNX-py: Klobuchar (Klobuchar 1987), NTCM (German Aerospace Center (DLR) 2023), and interpolation from GIM products (Schaer et al. 1998). Let's use the Klobuchar model using the previously obtained GPSA and GPSB coefficients. The equations for the Klobuchar model are presented below:

```
obs_crd = obs_crd[obs_crd['ev']>7]
```

Inputs in semicircles unless noted; E = elevation (semicircles), A = azimuth (rad).

$$\psi = \frac{0.0137}{E + 0.11} - 0.022 \qquad \text{Earth-centered angle}$$

$$\phi_I = \phi_u + \psi \cos A, \quad \phi_I \in [-0.416,\, 0.416] \qquad \text{IPP latitude}$$

$$\lambda_I = \lambda_u + \frac{\psi \sin A}{\cos \phi_I} \qquad \text{IPP longitude}$$

$$\phi_m = \phi_I + 0.064 \cos(\lambda_I - 1.617) \qquad \text{Geomagnetic latitude at IPP}$$

$$t' = (43200\, \lambda_I + t_{\mathrm{GPS}}) \bmod 86400 \qquad \text{Local time (GPS SOW)}$$

$$A_I = \sum_{n=0}^{3} \alpha_n \phi_m^n, \quad A_I \geq 0 \qquad \text{Amplitude of ionospheric delay}$$

$$P_I = \sum_{n=0}^{3} \beta_n \phi_m^n, \quad P_I \geq 72000 \qquad \text{Period of ionospheric delay (s)}$$

$$X_I = \frac{2\pi\,(t' - 50400)}{P_I} \qquad \text{Phase of ionospheric delay}$$

$$F = 1 + 16(0.53 - E)^3 \qquad \text{Slant factor}$$

$$I_{L1} = \begin{cases} [5{\cdot}10^{-9} + A_I(1 - \frac{X_I^2}{2} + \frac{X_I^4}{24})]\, F, & |X_I| \leq 1.57, \\ 5{\cdot}10^{-9}\, F, & |X_I| > 1.57, \end{cases} \qquad \text{Ionospheric delay (seconds) at L1}$$

$$\Delta R = c\, I_{L1}, \qquad I_f = \left(\frac{f_{L1}}{f}\right)^2 I_{L1} \qquad \text{Range effect and frequency scaling}$$

```
obs_crd=obs_crd.copy()
obs_crd.loc[:,'ion'] = obs_crd.apply(lambda row: gnx.klobuchar(azimuth=
    row['az'], elev=row['ev'],
                                                        fi=obs_data.meta
    [5][0],
                                                        lambda_=obs_data
    .meta[5][1],
                                                        tow=row['toc'],
                                                        alfa=nav_data.
    gpsa,
                                                        beta=nav_data.
    gpsb),axis=1)
```

Two different troposphere models are implemented: Saastamoinen (Saastamoinen 1972) and the Collins model (Collins 1999). In the case of PPP, however, Niell's mapping with ZTD estimation is used (Niell 1996). For the simple SPP discussed here, the Saastamoinen model will be entirely sufficient. The model equations are presented below. The correction is calculated by calling the *saastamoinen*() function.

Saastamoinen tropospheric delay

$$\epsilon = |\text{ev}| \cdot \frac{\pi}{180}$$

$$P(h) = Pr\,(1 - 2.26 \times 10^{-5} h)^{5.225}$$

$$T(h) = Tr - 0.0065\,h$$

$$H(h) = Hr\,\exp(-6.396 \times 10^{-4} h)$$

$$e(h) = 0.01\,H(h)\,\exp(-37.2465 + 0.213166\,T(h) - 2.56908{\times}10^{-4} T(h)^2)$$

$$d_{\text{trop}}(\epsilon, h) = \underbrace{\frac{0.002277}{\sin\epsilon}\left[P(h) - \frac{B(h)}{\tan^2\epsilon}\right]}_{\text{hydrostatic (dry) component}} + \underbrace{\frac{0.002277}{\sin\epsilon}\left(\frac{1255}{T(h)} + 0.05\right) e(h)}_{\text{wet component}} \quad [\text{m}]$$

Parameter definitions

- **Elevation angle:** $\epsilon = |\text{ev}| \cdot \pi/180$
- **Sea-level reference constants:** $Pr = 1013.25$ mbar, $Tr = 291.15$ K, $Hr = 50\%$
- **Pressure model:** $P(h) = Pr(1 - 2.26 \times 10^{-5} h)^{5.225}$
- **Temperature profile:** $T(h) = Tr - 0.0065\,h$
- **Relative humidity vs. height:** $H(h) = Hr\,\exp(-6.396 \times 10^{-4} h)$
- **Water vapor partial pressure:** $e(h) = 0.01\,H(h)\,\exp(-37.2465 + 0.213166T - 2.56908{\times}10^{-4} T^2)$
- **Dry mapping coefficient:** $B(h)$ obtained by linear interpolation between:

  heights $h_a = [0, 500, 1000, 1500, 2000, 2500, 3000, 4000, 5000]$

  values $B_a = [1.156, 1.079, 1.006, 0.938, 0.874, 0.813, 0.757, 0.654, 0.563]$

```
obs_crd=obs_crd.copy()
obs_crd.loc[:,'tro'] = gnx.saastamoinen(ev=obs_crd['ev'].values, h=
    obs_data.meta[5][-1])
```

Relativistic path range correction is not necessary in SPP, as its effect is small and significantly below the accuracy of orbits and code observations, but nevertheless, for presentation purposes, we will calculate it below using the appropriate function. In PPP, it is worth introducing it, as it will interfere with positioning accuracy.

Relativistic Path Range Correction (Ashby 2003):

$$
\begin{aligned}
\rho_{\text{geom}} &= \left\| \mathbf{r}^{sat} - \mathbf{r}_{rcv} \right\|, \\
\Delta\rho_{\text{rel}} &= \frac{2\,\mu}{c^2}\,\ln\left(\frac{r^{sat} + r_{rcv} + \rho_{\text{geom}}}{r^{sat} + r_{rcv} - \rho_{\text{geom}}}\right), \\
\rho_{\text{corr}} &= \rho_{\text{geom}} + \Delta\rho_{\text{rel}}.
\end{aligned}
$$

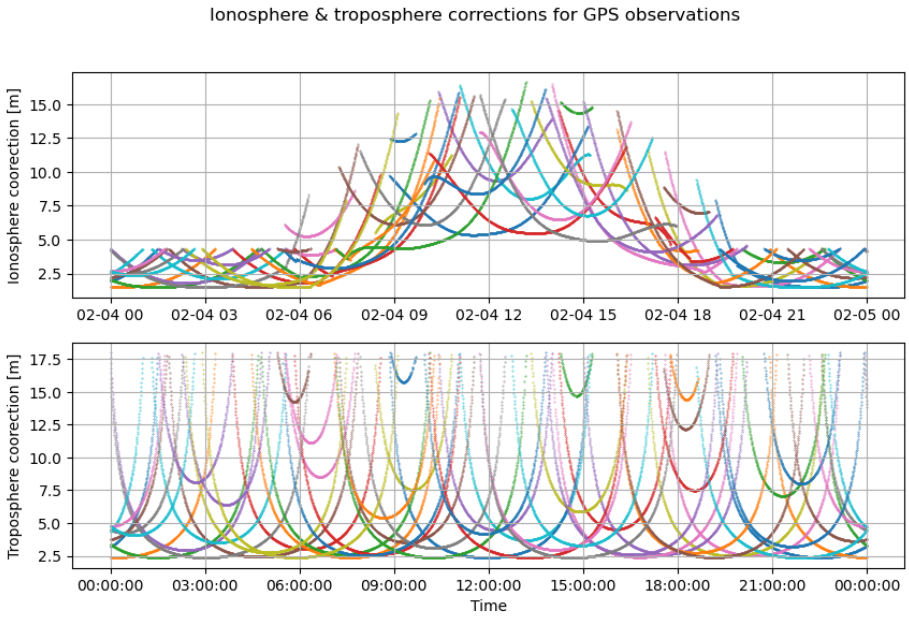
where:

$\mathbf{r}^{sat} \in \mathbb{R}^3$	satellite ECEF position (m),
$\mathbf{r}_{rcv} \in \mathbb{R}^3$	receiver ECEF position (m),
ρ_{geom}	Euclidean geometric range (m),
$c = 299{,}792{,}458$ m/s	speed of light,
$\mu = GM_{\oplus} \approx 3.986004418 \times 10^{14}$ m^3/s^2	Earth's gravitational constant.

```
obs_crd=obs_crd.copy()
obs_crd['dprel'] = gnx.rel_path_corr(rsat=obs_crd[['xe','ye','ze']], rrcv
    =obs_data.meta[4])
```

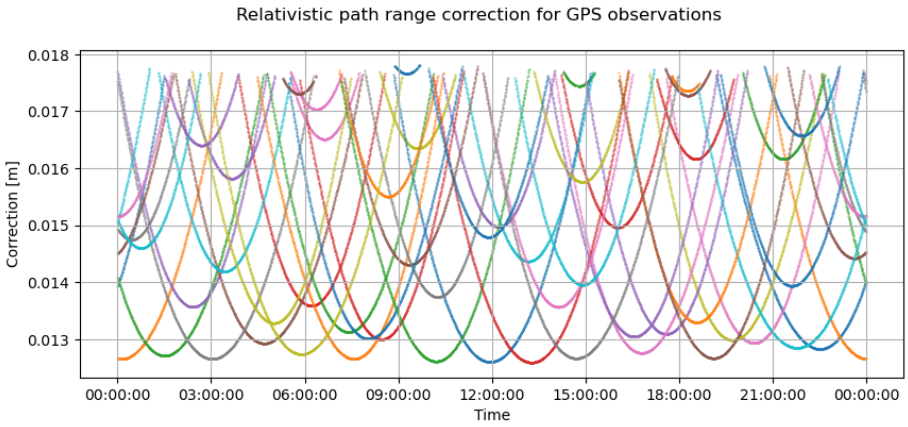
Magnitude of ionosphere and troposphere correction can be seen at the figure below:

```

fig, ax = plt.subplots(2,1,figsize=(10,6))
fig.suptitle('Ionosphere & troposphere corrections for GPS observations')
for ind, gr in obs_crd.groupby('sv'):
    gr = gr.reset_index()
    ax[0].scatter(gr['time'], gr['ion'], label=ind,s=0.1)
    ax[1].scatter(gr['time'], gr['tro'], label=ind,s=0.1)
ax[0].set_ylabel('Ionosphere coorection [m]')
ax[1].set_ylabel('Troposphere coorection [m]')
ax[1].set_xlabel('Time')
ax[0].grid()
ax[1].grid()
plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%H:%M:%S'))
```



Below you can see the scale of relativistic corrections for our observations.

```
from matplotlib import pyplot as plt
from matplotlib import dates as mdates
fig, ax = plt.subplots(1,1,figsize=(10,4))
fig.suptitle('Relativistic path range correction for GPS observations')
for ind, gr in obs_crd.groupby('sv'):
    gr = gr.reset_index()
    ax.scatter(gr['time'], gr['dprel'], label=ind,s=0.1)
ax.set_ylabel('Correction [m]')
ax.set_xlabel('Time')
ax.grid()
plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%H:%M:%S'))
```



The PCO of the receiver in the ANTEX file is given in the local topocentric coordinate system: East, North, Up. We can introduce them into the observation in two ways: either permanently correct the station position a-priori by the PCO vector converted to the ECEF system, or introduce the PCO correction as an amendment to the observation by projecting the vector onto the line-of-sight vector for the satellite-receiver pair. The default approach in GNX-py is the latter.

To calculate the line of sight vector, we need the approximate position of the station – we have obtained it from the RINEX file header.

```
xyz_apr = obs_data.meta[4].copy()
```

We will also write an auxiliary function that will 'extract' the PCO for a given system and signal. The PCO is given in millimetres, so we convert the value to metres.

```
def get_pco(rec_pco, sys):
    return np.array(next(v for k, v in rec_pco.items() if k[-1] == sys))
    /1000
```

```
rec_pco_l1 = gnx.enu_to_ecef(dENU=get_pco(rec_pco, "G01"),flh=obs_data.
    meta[5])
rec_pco_l2 = gnx.enu_to_ecef(dENU=get_pco(rec_pco, "G02"),flh=obs_data.
    meta[5])
print(f'Reciever PCO vectors in dENU:\n L1: {rec_pco_l1}\n L2: {
    rec_pco_l2}')
```

```
Reciever PCO vectors in dENU:
 L1: [0.05104749 0.01635834 0.07149983]
 L2: [0.06894601 0.02092454 0.09314551]
```

Our line-of-sight vector points from the receiver to the satellite. After projecting the PCO vector, we obtain a correction for a specific observation.

```
dxyz = -(obs_crd[['xe','ye','ze']].to_numpy() - xyz_apr)
dnorm = np.linalg.norm(dxyz,axis=1).reshape(-1,1)
los = dxyz/dnorm
pco_los_l1 = np.sum(los*rec_pco_l1,axis=1)
pco_los_l2 = np.sum(los*rec_pco_l2,axis=1)
obs_crd=obs_crd.copy()
obs_crd.loc[:,'pco_los_l1'] = pco_los_l1
obs_crd.loc[:,'pco_los_l2'] = pco_los_l2
```

```
df_head(obs_crd[['pco_los_l1','pco_los_l2']],nrows=3,ncols=4)
```

sv	time	pco_los_l1	pco_los_l2
G02	2024-02-04 00:00:00	-0.066	-0.087
G02	2024-02-04 00:00:30	-0.065	-0.087
G02	2024-02-04 00:01:00	-0.065	-0.087

The height of the antenna can be treated in the same way as PCO, taking into account that it is, of course, independent of the frequency of the signal used. Like PCO, it is given in a topocentric system. Therefore, we convert the antenna height offset to ECEF and project it onto the line-of-sight.

```
ant_h = obs_data.meta[3] # Up, East, North
ant_h = np.array([ant_h[1], ant_h[2], ant_h[0]])
ant_h_ecef = gnx.enu_to_ecef(dENU=ant_h,flh=obs_data.meta[5])
ah_los =  np.sum(los*ant_h_ecef,axis=1)
obs_crd = obs_crd.copy()
obs_crd.loc[:,'ah_los'] = ah_los
```

With the data prepared in this way, let's create a short script for SPP and perform positioning based on iono-free observations and uncombined observations with the Klobuchar model. First, let's read the SINEX file for a given day to obtain the reference position with *parse_sinex*() function. This will allow us to evaluate the positioning accuracy.

```
SINEX='/Users/hubertpierzchala/Desktop/GPSlib/art_data_package/
    IGS0OPSSNX_20240350000_01D_01D_CRD.SNX'
snx = gnx.parse_sinex(SINEX)
```

```
ref = snx.loc['BOR1'].to_numpy()
ref_flh = gnx.ecef2geodetic(ref)
```

Our a-priori position is the one we took from RINEX file header.

```
print(f'Reference coordinates: ECEF: {ref.round(3)}\nGeodetic: {ref_flh}\
    nA-priori coordinates: {xyz_apr}')
```

```
Reference coordinates: ECEF: [3738358.133 1148174.009 5021815.934]
Geodetic: [ 52.27695923  17.07346199 124.35793091]
A-priori coordinates: [3738358.5958 1148173.5785 5021815.7483]
```

Below you will find a simple script that performs SPP. It is not an advanced program, but it will allow us to verify the correctness of the above steps. You will find a description of the code in the comments in the cell below.

```
import pandas as pd
import numpy as np
# We define constants for ionofree combinations, speed of light, and list
     of results.
dtr = 0
f1=1575.42e06
f2= 1227.60e06
k1 = f1**2/(f1**2-f2**2)
k2 = f2**2/(f1**2-f2**2)
C=299792458
out = []



def hjacobian(x,sats):
    A = np.zeros(shape=(len(sats),4))
    dxyz = -(sats-x[:3])
    dnorm = np.linalg.norm(dxyz,axis=1)
    for i in range(len(sats)):
        A[i,:] = [dxyz[i,0]/dnorm[i],dxyz[i,1]/dnorm[i],dxyz[i,2]/dnorm[i
    ],1]
    return A

def hx(x,sats):
    dxyz = -(sats-x[:3])
    dist = np.sqrt(dxyz[:,0]**2 + dxyz[:,1]**2 + dxyz[:,2]**2)
    return dist + x[-1]


for num, (t, gr) in enumerate(obs_crd.groupby('time')):

    # We select observations above 10 degrees of elevation - a reasonable
     threshold.
    # gr=gr[gr['ev']>10]
    #We create a plan matrix - A, its dimensions are (N,4), where N is
    the number of observations,
    # and 4 corresponds to our state vector - the receiver position and
    clock.
```

```
    A = np.zeros(shape=(len(gr),4))
    #The elements of the state vector are the components of the line of
    sight vectors.
    dxyz = -(gr[['xe','ye','ze']].to_numpy() - xyz_apr)
    dnorm = np.linalg.norm(dxyz,axis=1)
    for i in range(len(gr)):
        A[i,:] = [dxyz[i,0]/dnorm[i],dxyz[i,1]/dnorm[i],dxyz[i,2]/dnorm[i
    ],1]
    # We determine the weight -matrixlet us assume an observation error
    of 0.5 at the zenith for both observation types.
    err = np.diag(0.5/np.sin(np.deg2rad(gr['ev']))**2)
    P = np.linalg.inv(err)
    Psf = P.copy()
    # We create our observation vector. Please note that for the ionofree
     combination, we do not introduce the ionosphere or TGD - the
    receiver clocks are referenced to the IF C1W-C2W combination,
    # we have C1C-C2W, but we do not have DCB between C1C-C1W.
    # However, we introduce TGD into the observation at a single
    frequency.  You can choose not to introduce it and see what happens.
    observed = np.asarray((k1*gr['C1C'] - k2*gr['C2W']) - gr['tro'] + gr[
    'clk']*C-gr['dprel'] + (k1*gr['pco_los_l1'] - k2*gr['pco_los_l2'])+
    gr['ah_los'])
    observed_sf =np.asarray(gr['C1C'] - gr['tro'] + (gr['clk']-gr['TGD'])
    *C-gr['dprel'] - gr['ion'] + gr['pco_los_l1'] + gr['ah_los'])
    # Our ""calculated value is simply the geometric distance.
    computed = gnx.calculate_distance(gr[['xe','ye','ze']].values,
    xyz_apr)

    L = observed-computed
    Lsf = observed_sf-computed
    # A very simple element of the observation sanity -checkwe filter out
     observations based on a 3-sigma filter on prefit residuals.
    # This will allow us to protect ourselves against clock jumps.
    md = np.median(L)
    mask = np.where(np.abs(L) > 3*md)
    if len(mask[0]) < len(gr)/2:
        for ind in mask:
            P[ind, ind]/=100
    # LSQ solution for IF
    Q = np.linalg.inv(A.T@P@A)
    X = Q.dot(A.T.dot(P).dot(L))

    # The same sanity check for SF observations.
    md = np.median(Lsf)
    mask = np.where(np.abs(Lsf) > 3*md)
    if len(mask[0]) < len(gr)/2:
        for ind in mask:
            Psf[ind, ind]/=100
    # LSQ for SF
    Q = np.linalg.inv(A.T@Psf@A)
```

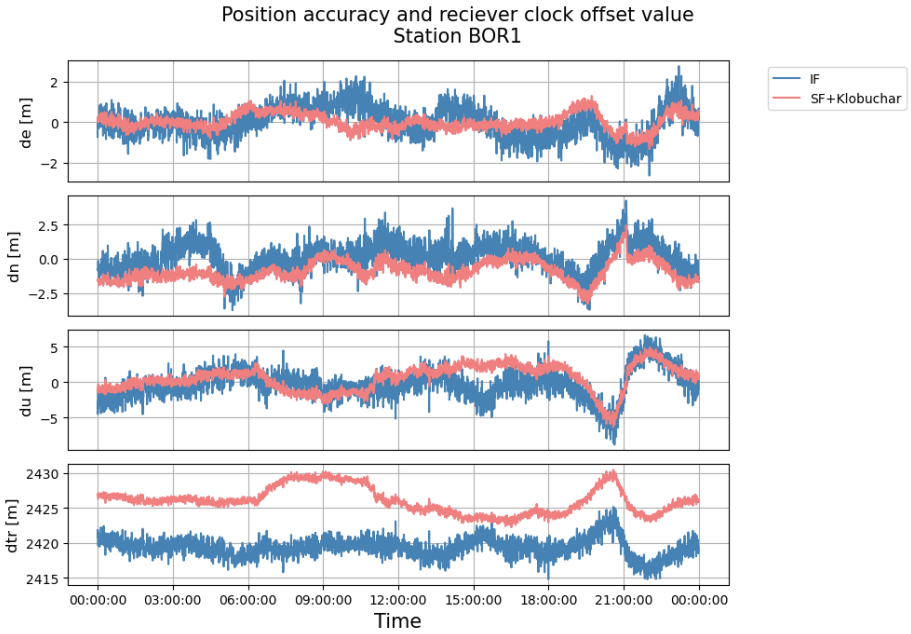
```
    Xsf = Q.dot(A.T.dot(Psf).dot(Lsf))
    # Corrections to observations
    V = A.dot(X)-L
    Vsf = A.dot(Xsf)-Lsf
    obs_crd=obs_crd.copy()
    obs_crd.loc[gr.index,'V']=V
    obs_crd.loc[gr.index,'Vsf']=Vsf
    # We collect results from both modes.
    dtr = X[3]
    obl_sf = xyz_apr+Xsf[0:3]
    obl = xyz_apr+X[0:3]

    dtrsf = Xsf[3]
    dif=ref-obl
    dif_sf=ref-obl_sf
    # Convert positioning errors to ENU coordinates.
    denu = gnx.ecef_to_enu(dif,flh=ref_flh)
    denu_sf = gnx.ecef_to_enu(dif_sf,flh=ref_flh)




    out.append(pd.DataFrame({'time':[t],'de':denu[0],'dn':denu[1],'du':
    denu[2],'dtr':[dtr],
                             'de_sf':denu_sf[0],'dn_sf':denu_sf[1],'du_sf
    ':denu_sf[2],'dtr_sf':dtrsf}
                            ))
    # print(f'Processing: {obs_data.meta[0]} epoch: {t} \n Error IF: {
    denu}\n Error SF: {denu_sf}\n')
    # print('-----'*10)
```

Let's visualise our results. Below, I will show you how to use completely ordinary Python code to visualise positioning results. This is a major advantage of GNX-py – the simplicity of integrating popular Python libraries such as pandas and matplotlib to manage results and data.

```
result = pd.concat(out)
df_head(result,nrows=3,ncols=10,floatfmt='.1f')
```

index	time	de	dn	du	dtr	de_sf	dn_sf	du_sf	dtr_sf
0.0	2024-02-04 00:00:00	-0.7	-0.8	-2.8	2420.8	0.0	-1.6	-1.2	2426.9
0.0	2024-02-04 00:00:30	0.0	-0.9	-4.5	2421.9	0.3	-1.6	-1.4	2427.0
0.0	2024-02-04 00:01:00	-0.3	-0.6	-2.6	2420.8	0.2	-1.6	-1.0	2426.8

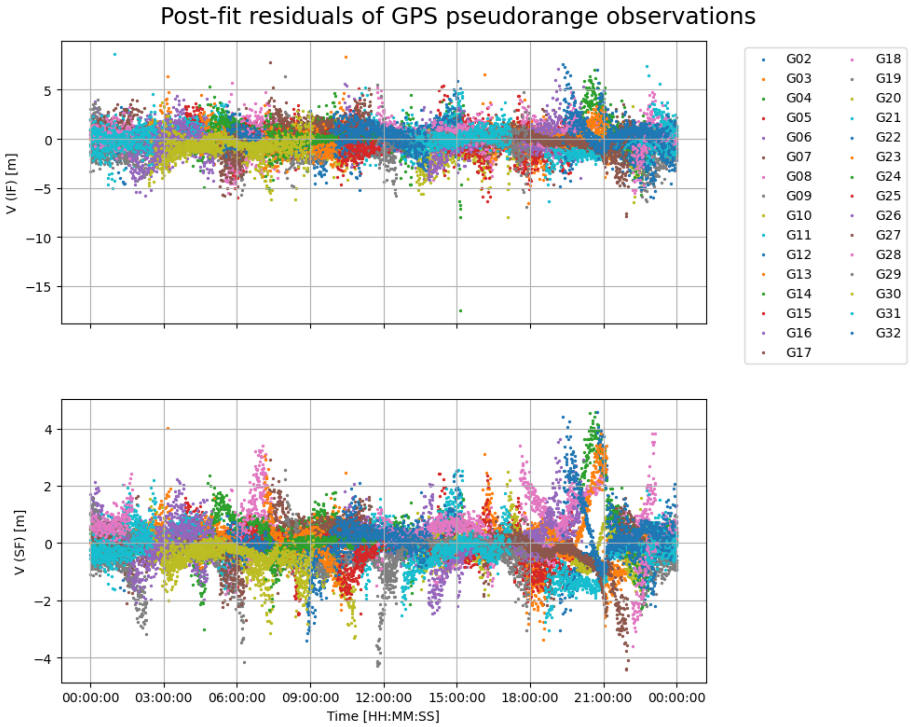
```python
import matplotlib.pyplot as plt
from matplotlib.dates import DateFormatter

fig, ax = plt.subplots(4, 1, figsize=(10, 7), sharex=True)
cols = ['de', 'dn', 'du', 'dtr']
fig.suptitle(f'Position accuracy and reciever clock offset value\n
    Station {obs_data.meta[0]} ',fontsize=15)
for i, col in enumerate(cols):
    ax[i].plot(result['time'], result[col], c='steelblue',label='IF')
    ax[i].plot(result['time'], result[f'{col}_sf'], c='lightcoral',label=
    'SF+Klobuchar')
    ax[i].set_ylabel(f'{col} [m]',fontsize=12)
    ax[i].grid()
    if i < len(cols) - 1:
        ax[i].tick_params(axis='x', which='both', labelbottom=False,
    bottom=False)
ax[0].legend(loc='best',bbox_to_anchor=(1.05, 1), ncol=1)

# format and show x-axis (czas) only on the last subplot
ax[-1].xaxis.set_major_formatter(DateFormatter('%H:%M:%S'))
ax[-1].set_xlabel('Time',fontsize=15)

plt.tight_layout()
```



What you see above is the accuracy of our positioning: errors in the East, North and Up planes, and clock offset. We have nothing to compare the latter with,

so we simply visualise it. The graph above illustrates what you can often read in textbooks on GNSS – iono-free observations are free from the influence of the ionosphere at the cost of increased noise caused by the linear combination of observations, while single-frequency observations are less noisy than iono-free ones, but require ionosphere modelling (Teunissen and Montenbruck 2017; HofmannWellenhof et al. 2008). Observing the differences in clock offset values, we see another characteristic feature: differences in clock delay for different frequencies. Let us also visualise the post-fit observation residuals. Once again, we see the effect I described above – IF residuals suffer from more noise than SF residuals.

```
import matplotlib.pyplot as plt
from matplotlib.dates import DateFormatter

fig, ax = plt.subplots(2, 1, figsize=(10, 8), sharex=True)

fig.suptitle('Post-fit residuals of GPS pseudorange observations',
    fontsize=18)

for sv, gr in obs_crd.groupby('sv'):
    ax[0].scatter(gr.index.get_level_values('time'), gr['V'], label=sv, s
    =2)
    ax[1].scatter(gr.index.get_level_values('time'), gr['Vsf'], label=sv,
     s=2)

ax[0].set_ylabel('V (IF) [m]', fontsize=10)
ax[1].set_ylabel('V (SF) [m]', fontsize=10)
ax[1].set_xlabel('Time [HH:MM:SS]', fontsize=10)

for i in range(2):
    ax[i].grid()
    ax[i].xaxis.set_major_formatter(DateFormatter('%H:%M:%S'))

# legenda tylko na górnym wykresie
ax[0].legend(loc='best', bbox_to_anchor=(1.05, 1), ncol=2)

plt.tight_layout()
plt.show()
```



Let us examine the RMS values broken down into components and in 3D.

```
cols = ['de', 'dn', 'du']
for i, col in enumerate(cols):
    pif = np.sqrt(np.mean(result[col]**2))
    sf = np.sqrt(np.mean(result[f'{col}_sf']**2))
    print(f'{col.upper()} RMS  IF: {pif:.2f} | SF: {sf:.2f}| meters')
```

```
DE RMS  IF: 0.78 | SF: 0.42| meters
DN RMS  IF: 1.10 | SF: 1.18| meters
DU RMS  IF: 2.11 | SF: 1.95| meters
```

```
result['3d_if'] = np.sqrt(result['de']**2 + result['dn']**2 + result['du']**2)
result['3d_sf'] = np.sqrt(result['de_sf']**2 + result['dn_sf']**2 + result['du_sf']**2)
```

```
for c in ['if','sf']:
    rms = np.sqrt(np.mean(result[f'3d_{c}']**2))
    print(f'{c.upper()} 3D RMS: {rms:.03f}' )
```

```
IF 3D RMS: 2.505
SF 3D RMS: 2.317
```

Interestingly, the SF solution with the klobuchar model proved to be slightly more accurate in terms of RMS than the IF solution.

SPP Pipeline in GNX-py

Above, we have implemented a series of steps to obtain positioning results. All of our preparations are contained in the SPPSession class, which takes an instance of the SPPConfig class as an argument, which is a class that stores the configuration parameters of our positioning. This is the basic way of implementing things in GNX-py: a configuration class for management and an execution class for the given configuration, which 'hides' all the complex code from us.

```
NAME ='BOR1'
DOY=35
config = gnx.SPPConfig(
                     obs_path=OBS,
                     nav_path=NAV,
                     sp3_path=None,
                     atx_path=ATX,
                     dcb_path=None,
                     gim_path=None,
                     sinex_path=SINEX,
                     use_gfz=True,
                     output_dir = './output',

                     sys='G',
                     gps_freq='L1',
                     gal_freq='E1',

                     screen=False,
                     orbit_type='broadcast',
                     broadcast_tolerance='2H',

                     ionosphere_model='klobuchar',
                     troposphere_model='saastamoinen',
                     sat_pco=False,
                     rec_pco=True,
                     windup=False,
                     rel_path=True,
                     solid_tides=False,
                     antenna_h =True,
                     ev_mask=7,


                     time_limit=None,
                     station_name=NAME,
                     day_of_year=DOY,

                     solver='LS' # in future: KF/ EKF
)
```

```
controller = gnx.SPPSession(config)
```

```
spp_results = controller.run()
```

```
for f in fields(spp_results):
    print(f.name, f.type)
```

```
solution typing.Optional[ForwardRef('pd.DataFrame')]
residuals_gps typing.Optional[ForwardRef('pd.DataFrame')]
residuals_gal typing.Optional[ForwardRef('pd.DataFrame')]
covariance_info typing.Optional[ForwardRef('pd.DataFrame')]
```

The result is a structure - a dataclass containing the results. *solution* - these are the estimated state vector parameters, i.e. the receiver's position and clock. *residuals_gps* and (optionally) *residuals_gal* are residual-enriched dataframe with observations, and $covariance\_info$ contains elements of the covariance matrix of the determined parameters in each epoch. Use pandas and matplotlib tools to compare the results from our tutorial with those from SPPSession – you should obtain the same results with a difference of 1^{-5} metres.

```
residuals = spp_results.residuals_gps
```

In this chapter, you learned about an example of data processing using the GNX-py library. You performed steps from loading data to visualising positioning results. Finally, you learned about the SPPConfig and SPPSession classes, which allow you to perform the same task without having to write your own code. In the following notebooks, I will discuss the use of similar functionalities for positioning in PPP mode, orbit analysis, and ionosphere monitoring and modelling. In each case, the user interface is similar: configure the configuration class and pass it to the execution class (SPPConfig -> SPPSession). In the Library folder, you will find .py scripts such as: SPPSession, PPPSession, STECSession, SISSession. These are scripts that allow you to run your own pipeline for multiple observation files or even data from multiple days, where you only need to replace the appropriate paths to get the results. After reading the subsequent notebooks in this tutorial, you will know what to enter and where to enter it in order to perform the desired task.

Let me give you some additional information about the library that will help you understand its current capabilities. The GNX-py library is part of a larger project that I want to develop. Its goal is to create a convenient tool for obtaining data for scientific research, but also, as far as possible, to popularise and provide scientific assistance to students and GNSS enthusiasts. The current version of the library is the first 'operational' version and is not yet a fully comprehensive tool. Currently, the library only supports RINEX 3.0 files for observation and navigation files. The supported systems are GPS and Galileo.

In terms of positioning, within these systems, the library processes a maximum of two signals at a time, in any configuration (e.g. L1L2, L1L5, E1E5a, E1E5b, etc.). Some elements are still waiting their turn in the queue for optimisation, primarily data loading and functions that calculate certain corrections. If you encounter any bugs, please do not hesitate to start a thread on GitHub.

I invite you to read the following chapters of the tutorial. I recommend the following order: - Introduction (since you are reading this, you have already finished it) - PPP - Orbits - Ionosphere

In some stages, we will use data from previous notebooks, hence this order.

PPP in GNX-py

In this tutorial, I will discuss the PPP module of the GNX-py library. Since the core of the PPP technique is the use of the Kalman filter (Kouba and Héroux 2001) , I will first discuss the basic information about this algorithm and its application in PPP. For detailed information about the Kalman filter, its implementation methods, and its variants, I encourage you to refer to the guide (Labbe 2020). The FilterPy library discussed there has been integrated into GNX-py, and one of the classes of this library - ExtendedKalmanFilter, is used for PPP implementation in GNX-py. I will also discuss the corrections I use, such as ionosphere models, troposphere models, satellite clocks, etc. This will help you understand how to process GNSS observations and what happens during parameter estimation. Next, I will discuss the structure of the GNX-py module and provide several examples of the use of positioning algorithms implemented in the library.

Parameters estimation in PPP

Observation model:

For each satellite s and frequency i code and phase observations at a single frequency can be written as (Teunissen and Montenbruck 2017):

$$
\begin{aligned}
C_i &= \rho_r^s + (\Delta_{PCO,i}^{rcv} - \Delta_{PCO,i}^{s}) - c(dt^s - dt_r) + \gamma_i I \\
&\quad + T + me_{wet} ZTD + \Delta_{ah} + \rho_{rel} + e_r^s \, \delta_{tides} + OSB_{C_i} + \varepsilon_{C_i} \\
L_i &= \rho_r^s + (\Delta_{PCO,i}^{rcv} - \Delta_{PCO,i}^{s}) - c(dt^s - dt_r) - \gamma_i I \\
&\quad + T + me_{wet} ZTD + \Delta_{ah} + \rho_{rel} + e_r^s \, \delta_{tides} + OSB_{L_i} + \lambda_i N_i + \lambda_i phw + \varepsilon_{L_i}
\end{aligned}
$$

where:

C_i — code pseudorange observation

L_i — carrier phase observation

ρ_i — geometric distance between satellite s and receiver r

$\Delta^r_{PCO,i}$ - reciever phase center offset

$\Delta^s_{PCO,i}$ - satellite phase center offset

c — speed of light

$d_{ts,i}$ — satellite clock offset

dt_r - reciever clock offset

T_i — slant tropospheric delay

me_{wet} - mapping function

ZTD - zenith tropospheric delay

Δ_{ah} - antenna height

ρ_{rel} - relativistic path range correction

e^s_r - line-of-sight unit vector

δ_{tides} solid tides displacement correction

OSB_{C_i} / OSB_{L_i} - observable specific satellite hardware bias

λ_i — carrier phase wavelength at frequency i

N_i — ambiguity for satellite i

phw - carrier phase wind-up correction

$\varepsilon^C_i, \varepsilon^L_i$ — measurement noise

Ionosphere-free observations, that are free from the influence of the ionosphere are recorded as (Teunissen and Montenbruck 2017) :

$$\begin{aligned} C_{IF} &= \frac{f_1^2}{(f_1^2 - f_2^2)} C1 - \frac{f_2^2}{(f_1^2 - f_2^2)} C2 \\ &= \rho^s_r + (\Delta^{rcv}_{PCO,IF} - \Delta^s_{PCO,IF}) - c(dt^s - dt_r) + T + me_{wet}\, ZTD \\ &+ \Delta_{ah} + \rho_{rel} + e^s_r\, \delta_{tides} + \varepsilon_{C_{IF}} \end{aligned}$$

$$\begin{aligned} L_{IF} &= \frac{f_1^2}{(f_1^2 - f_2^2)} L1 - \frac{f_2^2}{(f_1^2 - f_2^2)} L2 \\ &= \rho^s_r + (\Delta^{rcv}_{PCO,IF} - \Delta^s_{PCO,IF}) - c(dt^s - dt_r) + T + me_{wet}\, ZTD \\ &+ \Delta_{ah} + \rho_{rel} + e^s_r\, \delta_{tides} + \varepsilon_{L_{IF}} + \lambda_{IF} N_{IF} \end{aligned}$$

The PPP technique enables the positioning of a single receiver using phase and code observations with sub-centimeter accuracy (Kouba and Héroux 2001;

Glaner 2022; Schönemann 2014). The accuracy of the final position depends on the quality of the orbit models, clocks, and corrections used. PPP positioning involves determining a number of unknown parameters. For the purposes of this introduction, we will consider the case of a conventional model, in which code and phase observations are combined into ionosphere-free observations in which first-order ionospheric delay is eliminated. In this situation, our state vector contains the following parameters:

$$\mathbf{x} = \begin{bmatrix} X & Y & Z & d_{tr} & isb & ztd & N_1 & N_2 & ... & N_n \end{bmatrix}^{\mathrm{T}}$$

where

X, Y, Z — receiver coordinates (ECEF) dt_r — receiver clock offset ztd — zenith tropospheric delay isb - Inter system bias
N_i — carrier phase ambiguity for satellite i

X, Y, Z, and dt_r are the reciever coordinates and offset of the receiver clock, isb is the inter-system bias—in the case of processing data from two GNSS constellations at once, it reflects the difference in the time reference system and receiver delays between the processed signals from different GNSS. N_i are phase ambiguities for a $i-th$ satellite. This is the total number of phase cycles on the satellite-receiver path. During estimation, it is determined as a floating point number.

EKF prediction:

Let's discuss the process of parameter estimation using the Extended Kalman Filter. We assume that we have access to a time series of GNSS observations from a given receiver. We call what we measure a state: our state is the estimated parameters. The Kalman filter, in very simple terms, combines the prediction of the state of our system with the state obtained from measurements. Filtration is performed sequentially: in each epoch, we process the measurements available in that epoch and perform a prediction and update step. Let's first discuss the prediction step and look at the equations:

$$\hat{\mathbf{x}}_k = \mathbf{F}\hat{\mathbf{x}}_{k-1}$$

$$\mathbf{P}_{k|k-1} = \mathbf{F}_{k-1}\mathbf{P}_{k-1|k-1}\mathbf{F}_{k-1}^{\mathrm{T}} + \mathbf{Q}_{k-1}$$

$\hat{\mathbf{x}}_k$ and $\hat{\mathbf{x}}_{k-1}$ are our state vectors in epochs k and $k-1$, respectively. This vector contains the parameters discussed at the beginning of this section, i.e., the receiver and satellite parameters.

Matrix F is the so-called state transition matrix. The $F\,x$ operation is intended to map the predicted evolution of the state between adjacent epochs. Matrix F is an element of the process model, i.e., it is a mathematical representation of our

knowledge or assumptions about the behavior of the system. Let's illustrate this with an example. Assuming the state vector discussed above and a stationary GNSS receiver, we can make the following assumptions about our system:

- the position of the receiver does not change
- we do not know the characteristics of the clock - we assume white noise with high variance
- we assume that isb is constant over time
- we assume that ZTD can be described as a random walk
- satellite uncertainties are constant, assuming that cycle slips have already been eliminated
- In this case, the matrix F will look like this:

$$\mathbf{F} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

Each column of the matrix corresponds to a pramater in vector X. Try multiplying an example vector, even a 3 or 4 element vector, by a similar matrix on a piece of paper—you will see that after multiplication you will get the same vector (except for the clock, but we will get to that in a moment). This is the task of the matrix F - it reflects our predicted behavior of the system during the passage of one epoch.

The P matrix is an a-priori error matrix of the state vector elements. In the prediction step, the variance is propagated to the next epoch. The Q matrix, or process noise matrix, reflects our uncertainty about the process model. In other words, the Q matrix reflects our confidence in the accuracy of the description of the system's behavior by the F matrix. Assuming that we are certain that the receiver is stationary and modeling the clock as white noise, the first three diagonal elements of the Q matrix will be equal to zero (for coordinates), while for the clock we can set the variance to a very large value (e.g., 1e8, 1e9, etc.). We assume that the process model accurately describes the ambiguities and ISB as constant over time, so the corresponding elements of the Q matrix are also equal to zero. We model ZTD as a random walk—the variance could be, for example, 2-3 mm/h.

$$\mathbf{Q} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1^9 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & 0 & & 2e^{-3}\,\frac{\Delta_t}{3600} & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \end{bmatrix}$$

EKF update:

In the update step, we apply our measurements into the model and combine information about the measurements and their accuracy with information about the predicted state of the system and the accuracy of this prediction.

$$\begin{aligned} \mathbf{y}_k &= \mathbf{z}_k - \mathbf{h}(\hat{\mathbf{x}}_{k|k-1}) \\ \mathbf{S}_k &= \mathbf{H}_k \mathbf{P}_{k|k-1} \mathbf{H}_k^{\mathrm{T}} + \mathbf{R}_k \\ \mathbf{K}_k &= \mathbf{P}_{k|k-1} \mathbf{H}_k^{\mathrm{T}} \mathbf{S}_k^{-1} \\ \hat{\mathbf{x}}_{k|k} &= \hat{\mathbf{x}}_{k|k-1} + \mathbf{K}_k \mathbf{y}_k \\ \mathbf{P}_{k|k} &= (\mathbf{I} - \mathbf{K}_k \mathbf{H}_k) \mathbf{P}_{k|k-1} \end{aligned}$$

where
$\mathbf{y_k}$ - prefit residuals
$\mathbf{z}_k$ — vector of observed measurements at epoch k
$\mathbf{h}(\cdot)$ — nonlinear measurement function
$\mathbf{H}_k$ — Jacobian of $\mathbf{h}$ with respect to $\mathbf{x}$
$\mathbf{R}_k$ — measurement noise covariance
$\mathbf{S_k}$ - prefit residuals covariance matrix
$\mathbf{K_k}$ - Kalman gain
$\mathbf{Q}_{k-1}$ — process noise covariance
$\hat{\mathbf{x}}_\mathbf{k}$ - updated state vector
$\mathbf{P_{k|k}}$ - updated a-priori covariance matrix

Prefit residuals are the difference between measurements and predicted measurements. In our case, this is the difference between GNSS observations, $\mathbf{z_k}$, corrected for all available corrections, and calculated (predicted) observations. The function $\mathbf{h}(\cdot)$ is a function that maps the elements of the state vector to observations. In our case, predicted pseudorange and phase observation, denoted by $\widehat{z_{C_i}}$ and $\widehat{z_{L_i}}$ are computed as:

$$\begin{aligned} \widehat{z_{C_i}} &= \rho - (dt_r + isb) - me_{wet}\, ZTD \\ \widehat{z_{L_i}} &= \rho - (dt_r + isb) - me_{wet}\, ZTD - \lambda_i\, N_i \end{aligned}$$

Where dt_r, isb, ZTD and N_i are predicted elements of state vector

The Jacobian matrix is a design matrix—as in standard LSQ, each column contains the partial derivatives of the observation equation with respect to a given element of the state vector.

$$\mathbf{H}_k = \begin{bmatrix} \frac{\partial h_1}{\partial X} & \frac{\partial h_1}{\partial Y} & \frac{\partial h_1}{\partial Z} & \frac{\partial h_1}{\partial d_{tr}} & \frac{\partial h_1}{\partial \mathrm{ISB}} & \frac{\partial h_1}{\partial ztd} & \frac{\partial h_1}{\partial N_1} & 0 & \cdots & 0 \\ \frac{\partial h_2}{\partial X} & \frac{\partial h_2}{\partial Y} & \frac{\partial h_2}{\partial Z} & \frac{\partial h_2}{\partial d_{tr}} & \frac{\partial h_2}{\partial \mathrm{ISB}} & \frac{\partial h_2}{\partial ztd} & 0 & \frac{\partial h_2}{\partial N_2} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial h_m}{\partial X} & \frac{\partial h_m}{\partial Y} & \frac{\partial h_m}{\partial Z} & \frac{\partial h_m}{\partial d_{tr}} & \frac{\partial h_m}{\partial \mathrm{ISB}} & \frac{\partial h_m}{\partial ztd} & 0 & 0 & \cdots & \frac{\partial h_m}{\partial N_m} \end{bmatrix}$$

Assuming that we are processing code and phase observations from two GNSS systems, each row of the $\mathbf{H_k}$ matrix corresponds to a given observation from the $\mathbf{z}$ vector. The partial derivatives with respect to ambiguities are equal to zero for code observations, and similarly, derivative with respect to ISB is equal to zero for the reference constellation for the receiver clock (GPS by default). Assuming that for each satellite we process one IF code observation and one IF phase observation, the vector $\mathbf{z}$ and the matrix $\mathbf{h_k}$ will have dimensions of $N \cdot 2$ and ($N \cdot 2$, 5+N), respectively, where N is the number of satellites visible at a given epoch and 5 stands for 5 default parameters: station position vector elements, clock offset, ZTD and ISB.

The R matrix is a diagonal matrix of measurement variances. In GNX-py you can customize code variances and phases, although I can only vouch for the default values I used for positioning classes. For code and phase observations, we adopt the following error model:

$$\sigma = a + \frac{b}{sin(elev)}$$

Where a is equal to 1 and 1^{-4}, and b is equal to 3^{-3} and 2.5^{-4} meters for code and phase observations, respectively.

$\mathbf{S_k}$, as shown above, is the covariance matrix of observation residuals. The $\mathbf{K_k}$ - Kalman gain we can intuitively represent as a measure of our "confidence" in the state vector estimate relative to the prediction. A small gain means little "confidence" in the measurements. In the equation for the updated state vector $\hat{x_k}$, in the action $\mathbf{K}_k\mathbf{y}_k$ we weight the difference of measurements and predicted measurements by a measure of our confidence in measurements/uncertainty about predictions.

In GNX-py the approach to matrix composition is as follows: it stacks matrices in GPS and Galileo order, and within systems in code, phase order. This isn't particularly significant, but I mentioned it here for clarity. Notice the matrix H; I've listed the derivatives with respect to all components of the estimated state vector. Naturally, the code observations have a 0 in the derivative over the ambiguity, just as Galileo phase observations have zeros in the column designated for GPS phase ambiguities, and vice versa.

Overview of PPP module

We have already discussed the basics of the Kalman filter, here we will proceed to demonstrate the use of GNX-py to perform precise point positioning. We will start by discussing the module responsible for PPP positioning and then present two examples of use.

The multi-step preprocessing of the observations and the positioning itself is hidden in the library in the PPPSession class. The only required argument of this class is an instance of the configuration class - PPPConfig. This class contains a number of parameters that define the processing strategy. I will discuss the most important of them and recommend that you see the class in person in the /config module. There each parameter is briefly presented. Let's import the library and use the data in the ./data directory, and then see how to set the positioning strategy.

```
import gps_lib as gnx
import numpy as np
```

Below we define the path to our observation file, the path to the broadcast file, .ATX, .BIA, and a list of paths to the three SP3 files. We use the previous day's and next day's files by default, so that the interpolation functions can find the right window to interpolate the coordinates to the specified epochs. We also create a set with the designations of the systems we want to use : G stands for GPS, and E for Galileo. These abbreviations are used in all gnx-py modules. In addition, we also specify the day of measurement as DOY. The SINEX file is optional, but when provided, if it contains the coordinates of the station we are processing, its coordinates from the file will serve as a reference.

```
OBS_PATH='../data/BOR100POL_R_20240350000_01D_30S_MO.rnx'
NAME = OBS_PATH.split('/')[-1][:4]
NAV = '../data/BRDC00IGS_R_20240350000_01D_MN.rnx'
SP3_LIST=[
'../data/COD0OPSFIN_20240340000_01D_05M_ORB.SP3',
'../data/COD0OPSFIN_20240350000_01D_05M_ORB.SP3',
'../data/COD0OPSFIN_20240360000_01D_05M_ORB.SP3']
ATX_PATH='../data/ngs20.atx'
DCB_PATH= '../data/COD0OPSFIN_20240350000_01D_01D_OSB.BIA'
GIM='../data/COD0OPSFIN_20240350000_01D_01H_GIM.INX'
SINEX='../data/IGS0OPSSNX_20240350000_01D_01D_CRD.SNX'
OUT='../output'
SYS= {'G','E'}
DOY = 35
```

As I mentioned, the class "controlling" our processing is the PPPConfig class. In it we define the parameters that determine the various steps of processing. I will now discuss the most important arguments of the class and how they are used.

- *obs_path* - path to the observation file
- *nav_path* - the path to the broadcast file - it can be used to screen data based on the satellite health code or to interpolate coordinates. when using ionosphere models such as klobuchar or ntcm, this file is necessary to obtain model coefficients
- *screen* - do we want to apply the above data screening
- *atx/dcb_path* - paths to atx and dcb files. If they are omitted, the associated corrections will not be made, which will degrade the quality of the solution.
- *sp3_path* - list of paths to sp3 files
- *sys* - set with the constellation codes that I discussed above. Note, if you set {“G” , “E”}, i.e. GPS +. Galileo, and Galileo observations are not found in the observation file, because the receiver did not register them, processing will be aborted. In this situation, change to: {“G”}
- The designation of the signals used in the form F1F2, e.g. for GPS: L1L2, L1L5, and for Galileo E1E5a, E1E5b. By default, the program always tries to load two frequencies (L1L2 and E1E5a) in order to detect cycle slips using geometry free combinations.
- *positioning_mode* - you can choose “combined”, which is a conventional PPP using ionofree observations, or ‘uncombined’, which is an undifferenced - uncombined approach in which “raw” observations are processed (with two frequencies 4 observations per satellite: two code and two carrier phase), or ‘single’ - in which single frequency code and phase observations are processed
- *orbit_type* - source of satellite coordinates, “broadcast” or “precise”
- *time_limit* - datetime list with $[t_0, t_1]$ allows you to cut observations from the given time interval from t_0 to t_1
- *sinex_path* - the path to the sinex file, if the station coordinates are present in it, they will serve as a reference.
- *ev_mask* - cutoff angle of satellite elevation
- *station_name* - station code in the form of 4 characters, e.g. BRUX, ZIM2, P051.
- *dcb_path* - path to the DCB file
- *sat_pco* - whether to use satellite PCO correction, and in which way to apply it: ‘los’ (line-of-sight projection) or ‘crd’ - direct satellite coordinates modification
- *reset_every* - every how many minutes to reset the filter - useful for testing accuracy. Default is 0 - no resets

- *ionosophere_model* - which ionosphere model to use, available = [“gim”, “ntcm”, “klobuchar”] or False - no ionosphere model
- *troposphere_model*: which model to use, available: [“niell”, “collins”, “saastamoinen”] or False - no troposphere model
- *min_arc_len*: maximum length of satellite arc pass. This parameter is used for cycle slips detection. For given satellite, if time between two cycle slips is shorter than this value, this arc is rejected. Default value is 10 minutes.

You can control the entire estimation process by modifying the elements of matrices P and Q. For example, *p_amb*, *p_crd*, *p_isb*, *p_dt*, etc. are diagonal elements of the P matrix corresponding to phase ambiguity, coordinates, ISB, and receiver clock, respectively. Parameters with the prefix *q_* are corresponding elements of the Q matrix.

Example 1 - Conventional PPP

```
config = gnx.PPPConfig(
    obs_path=OBS_PATH,
    nav_path=NAV,
    sp3_path=SP3_LIST,
    atx_path=ATX_PATH,
    dcb_path=DCB_PATH,
    sinex_path=SINEX,
    gim_path=GIM,
    use_gfz=True,
    output_dir=OUT,
    sys=SYS,
    gps_freq='L1L2',
    gal_freq='E1E5a',

    screen=True,
    orbit_type='precise',

    ionosphere_model=False,
    troposphere_model='niell',
    sat_pco='los',
    rec_pco=True,
    windup=True,
    rel_path=True,
    solid_tides=True,
    antenna_h=True,
    ev_mask=7,
    cycle_slips_detection=True,
    min_arc_len=10,

    station_name=NAME,
    day_of_year=DOY,
```

```
    time_limit=None,

    # PPP parameters
    positioning_mode = 'combined',
    trace_filter=False,
    reset_every=0,


)
```

The class in which all processing and positioning takes place is PPPSession. The only thing we need to pass to it is our configuration file. We also create a logger that will capture information about the steps currently being performed.

```
controller = gnx.PPPSession(config=config)
```

To start processing, call the .run() method and wait – it may take a moment. In jupyter notebook, everything usually runs slower than in a script.

```
results = controller.run()
```

```
results.__annotations__
```

```
{'solution': "Union['pd.DataFrame', None]",
 'residuals_gps': "Union['pd.DataFrame', None]",
 'residuals_gal': "Union['pd.DataFrame', None]",
 'convergence': 'Union[float, int, None]'}
```

Our results are a PPPResult class instance, which contains:

- *solution* - a dataframe with estimated parameters for the station (position, clock, ztd, isb)
- *residuals_gps*, *residuals_gal* - a dataframe containing residuals of code and phase observations, as well as all observation data used in processing, such as raw observations, corrections, coordinates, and satellite clocks
- *convergence* - this is the measured convergence time. By default, GNX-py measures convergence as the time after which the change in the receiver's position from epoch to epoch does not exceed 5 mm. Although the filter uses a reference position to generate results, I did not implement convergence measurement based on position accuracy there, because I wanted to have a different default indicator, independent of the reference position. In fact, it is currently a measure of the stability or precision of the filter rather than convergence. We will use an external function to measure convergence based on accuracy.

```
sol = results.solution
print(sol.columns.tolist())
```

```
['de', 'dn', 'du', 'dtr', 'isb', 'ztd', 'x', 'y', 'z', 'mx', 'my', 'mz',
    'ct_min']
```

As you can see, in *solutions* dataframe we have access to the receiver position, ISB, clock offset, and ZTD. In addition, because we used the SINEX file, we also have errors in the topocentric cover in relation to the position from the file. Additionally, we see the columns mx, my, mz, and ct_min - these are, respectively: errors of estimated coordinates (elements of the covariance matrix of the state vector) and convergence time in minutes.

```
from plotting_tools import df_tail
df_tail(sol,nrows=5,ncols=7)
```

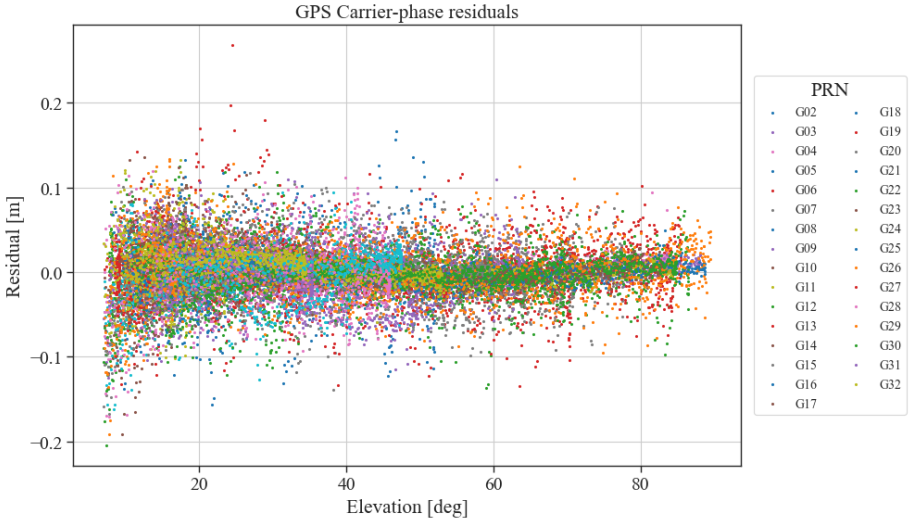
time	de	dn	du	dtr	isb	ztd
2024-02-04 23:57:30	-0.002	0.000	0.044	2419.515	1.770	0.005
2024-02-04 23:58:00	-0.002	0.000	0.044	2419.515	1.770	0.005
2024-02-04 23:58:30	-0.002	0.000	0.044	2419.507	1.770	0.005
2024-02-04 23:59:00	-0.002	0.000	0.044	2419.512	1.770	0.006
2024-02-04 23:59:30	-0.002	0.000	0.044	2419.505	1.770	0.014

Let's select data with GPS and Galileo residuals and use the function from the *plotting_tools.py* file for visualization. You can use *plot_residuals* function to automatically create a graph of phase and code residuals depending on time or satellite elevation. The function has several additional parameters that improve readability, which are useful in visualizing phase residuals in particular. First, you can skip the first N epochs on the graph (*cut_init_time* parameter). During the first few minutes, the filter is in the initialization phase and the residuals are very large because they contain initial estimates of the clock and other parameters that are subject to large errors, which makes it difficult to see the rest of the residuals. The *skip_obs* parameter allows you to skip the first N observations for a given satellite that becomes visible during the filter run - again, for the first few epochs (typically 0-10), the uncertainty is not yet close to its final value, and this also distorts the scale of the graph. Of course, for illustrative purposes, you can disable these options by setting both parameters to 0 - what appears to you as outliers is nothing to worry about. To select a specific observation type, choose $obs_type = 'code' or $'phase'. You can plot in relation to time or elevation by selecting $x\_axis =$ '*time*' or '*elev*'.

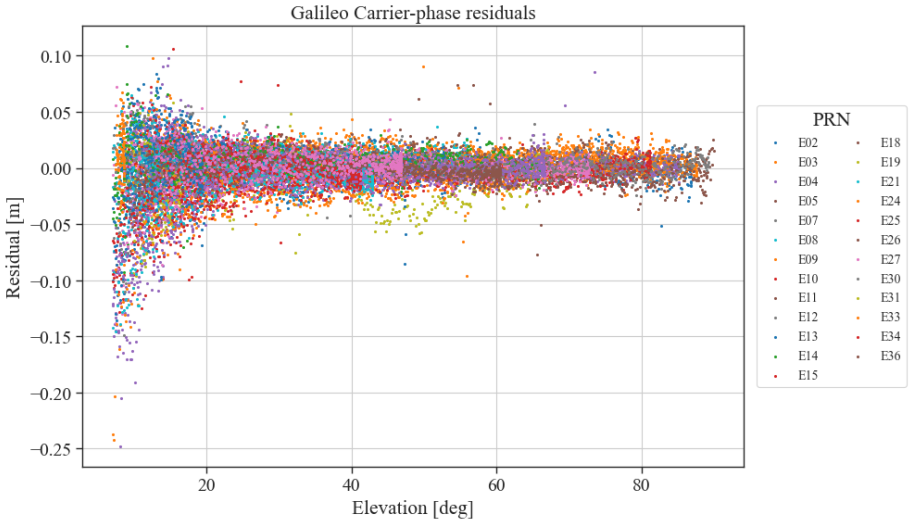
```
res = results.residuals_gps
res_gal = results.residuals_gal
```

```
from plotting_tools import plot_residuals, plot_positioning_series,
    plot_uduc_residuals
```

```
plot_residuals(df=res,obs_type='phase',x_axis='elev',skip_obs=10)
```




```
plot_residuals(df=res_gal,obs_type='phase',x_axis='elev',skip_obs=10)
```

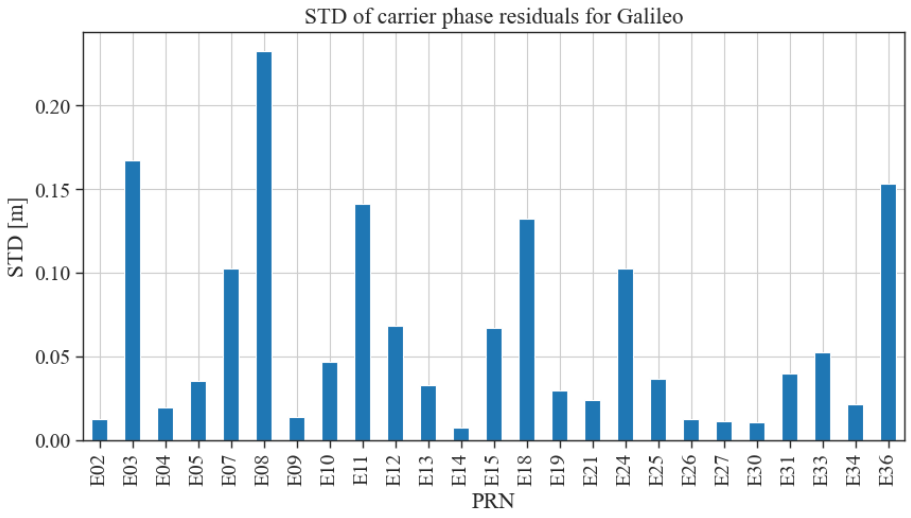


Note that Galileo observations have significantly less noise than GPS. You can process the data as it is — in pandas.DataFrame format, and obtain more statistics and information using standard pandas tools. You can see an example below:

```
res_gal['prn'] = res_gal.index.get_level_values('sv').str[:3]
```

```
res_gal['v_std'] = (
    res_gal
    .groupby("prn")["v"]
    .transform(
        lambda s: s.loc[
            # skip first 10 pbservations (ambiguity initialization time)
            s.index.get_level_values('time').isin(
                res_gal.index.get_level_values('time').unique()[30:]
            ) &
            (np.arange(len(s)) >= 5)
        ].std()
    )
).to_frame()
v_std_summary = res_gal.groupby('prn')['v_std'].first().sort_index()
v_std_summary.plot(kind='bar',title='STD of carrier phase residuals for
    Galileo',grid=True,figsize=(10,5), ylabel='STD [m]',xlabel='PRN')
```

```
<Axes: title={'center': 'STD of carrier phase residuals for Galileo'},
    xlabel='PRN', ylabel='STD [m]'>
```



I mentioned the convergence indicator earlier. In the *stat_tools.py* script, you will find a simple function that measures convergence for a time series based on accuracy. The function takes a convergence accuracy threshold, mode as 3D or 2D, and a minimum stability period, i.e., the number of epochs in which the error remains below the threshold.

```
from stat_tools import measure_convergence_time
```

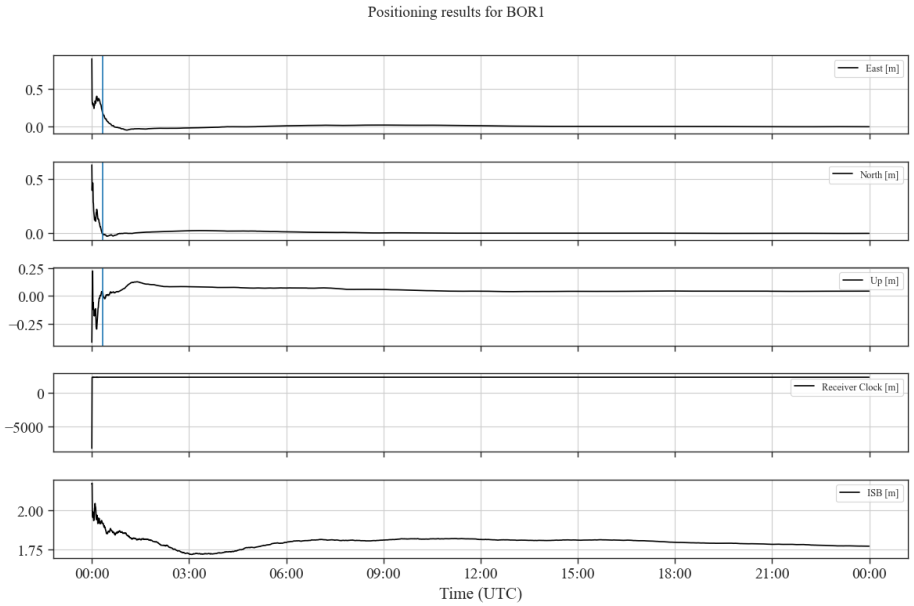
```
ct = measure_convergence_time(df=sol,threshold=0.1,min_stable_duration
    =30,mode='2D')
```

```
print(f'Convergence time : {ct} min')
```

```
Convergence time : 20.75 min
```

With the plot_position_series function from the plotting_tools file, you can create a default plot of the positioning results and estimated parameters. Additionally, you can mark the convergence time.

```
plot_positioning_series(sol=sol,title=f'Positioning results for {NAME}',
    color='black',ct=ct)
```



We can see that the filter converges quite quickly, especially in the dN and dU planes. Any inaccuracies in determining the ambiguities mainly affect the East component. Since we use float ambiguity determination, this effect is accentuated. Finally, we obtain a fairly accurate position at the sub-centimeter level horizontally and a few centimeters vertically, which should generally meet our expectations for PPP Float with final products.

Example 2 - Undifferenced & Uncombined PPP

Let's look at another example. This time, we will use positioning based on the undifferenced uncombined approach. In this approach, we do not eliminate the influence of the ionosphere by IF combinations, but instead introduce raw observations at two frequencies into our system and estimate the ionospheric delay for each observation (Glaner 2022; Schönemann 2014). GNX-py currently

supports processing two frequencies at a time. Additionally, we can enrich our system of equations by introducing ionospheric constants, which will "stiffen" our system in the initial phase of the filter's operation. These constants are ionospheric constraints derived from an ionosphere model, e.g., from GIM, in the form of ionospheric delays for each satellite. They act as pseudo-observations that help the filter adjust the estimated ionospheric delays. We introduce constants into the equation by setting the use_iono_constr argument to True in the PPPConfig class and selecting the ionosphere model. Since the modeled delays act as constraints, it is important to select the appropriate weights for the observations. In the case of GIM, we have three options:

- consistent use of modeled delay error information based on the RMS maps available in the ionex file;
- use of custom weights;
- a combination of the above

In GNX-py, the default approach for PPP-UDUC is the third option. Namely, by selecting *use_iono_constr* and *use_iono_rms* and *ionosphere_model* = *gim*, we obtain ionospheric delays and their errors. To allow the filter to gradually mitigate the impact of contraints and increasingly "trust" the measurements as the filter progresses, we scale the obtained errors by the parameters *sigma_iono_*0, *sigma_iono_end*, and *t_end*. These parameters are coefficients by which the ionospheric pseudo-observation errors are multiplied in each epoch. This approach means that we first treat pseudo-observations with a high degree of confidence (greater than the nominal error value) and at the end of a given period (*t_end*) with a lower degree of confidence. The default parameters for GIM are 0.15, 2.5, and 20 for *sigma_iono_*0, *sigma_iono_end*, and *t_end*, respectively, which represent the initial coefficient, the final coefficient, and the final epoch (in number of epochs, not minutes!).

In the UDUC approach in our system, we have slightly more parameters. First, for each satellite, we have two phase ambiguities: for frequencies $F1$ and $F2$. Second, for each code observation at a given frequency, we have one receiver hardware delay parameter, which is essentially the observable specific code bias of the receiver for a given observation. In addition, we have parameters as in conventional PPP, i.e., coordinates, clock offset, ISB in the case of more than one GNSS system, and ztd.

Currently, in GNX-py, you can use the PPP-UDUC approach in single or multi GNSS mode. We will perform positioning for the data from the previous example and compare the results.

```
config = gnx.PPPConfig(
    obs_path=OBS_PATH,
    nav_path=NAV,
    sp3_path=SP3_LIST,
    atx_path=ATX_PATH,
    dcb_path=DCB_PATH,
    sinex_path=SINEX,
```

```
    gim_path=GIM,
    use_gfz=True,
    output_dir=OUT,
    sys=SYS,
    gps_freq='L1L2',
    gal_freq='E1E5a',

    screen=True,
    orbit_type='precise',

    ionosphere_model='gim',
    use_iono_rms=True,
    troposphere_model='niell',
    sat_pco='los',
    rec_pco=True,
    windup=True,
    rel_path=True,
    solid_tides=True,
    antenna_h=True,
    ev_mask=7,
    cycle_slips_detection=True,
    min_arc_len=10,

    station_name=NAME,
    day_of_year=DOY,
    time_limit=None,

    # PPP parameters
    positioning_mode = 'uncombined',
    trace_filter=False,
    reset_every=0,


)
```

```
controller = gnx.PPPSession(config=config)
uncombined_results = controller.run()
```

```
unc_sol = uncombined_results.solution
```

```
unc_sol.to_csv('../output/unc_sol.csv')
```

```
df_tail(unc_sol,nrows=5,ncols=5)
```

time	de	dn	du	dtr
2024-02-04 23:57:30	-0.006	0.007	0.027	2425.699

time	de	dn	du	dtr
2024-02-04 23:58:00	-0.006	0.007	0.027	2425.592
2024-02-04 23:58:30	-0.006	0.007	0.027	2425.783
2024-02-04 23:59:00	-0.006	0.007	0.027	2425.646
2024-02-04 23:59:30	-0.006	0.007	0.027	2425.661

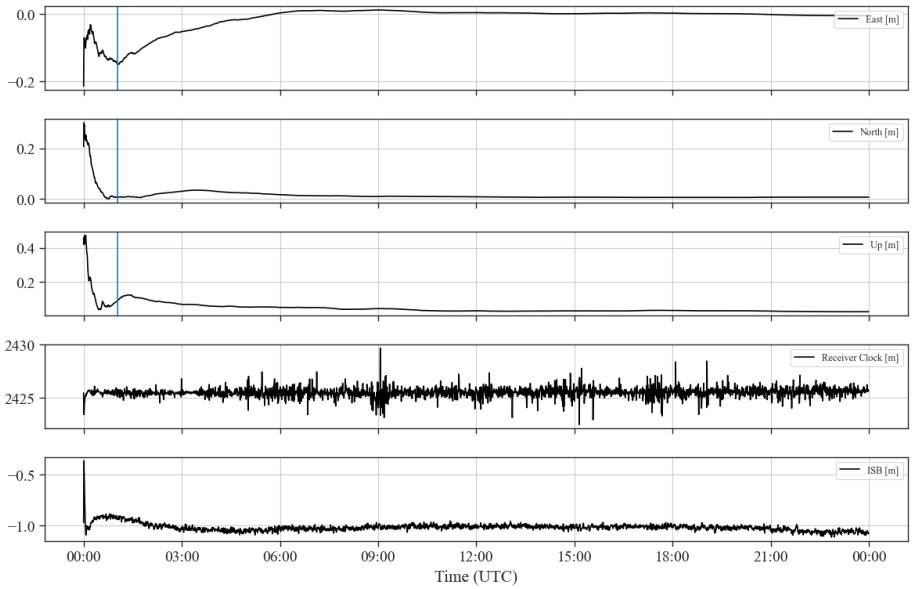
The results are slightly worse in the horizontal plane and nearly 2 cm better in the vertical component. As you can see, in addition to the values present in the state vector in PPP-IF, we also have columns starting with DCB. These are the receiver hardware delays I mentioned above. In the estimation process, in order to improve numerical efficiency, I used an approach in which OSB at the reference frequency (L1 and E1) is 'frozen', i.e. the value 0 is enforced by assigning this value to the low variance of the process model (matrix Q and P). OSB at the second frequency is determined relative to the reference frequency. Hence, what we obtain is in fact DSB bias - differential signal bias. The process model assumes that the bias affects code and phase observations equally. The OSB of phase observations is assumed to be absorbed by the float ambiguity.

Let us now measure the convergence time. It turns out to be almost three times longer for this station than for IF positioning. This can be explained by the number of unique parameters determined in the epoch, such as ionospheric delays.

```
ct = measure_convergence_time(df=unc_sol,threshold=0.1,
    min_stable_duration=30,mode='2D')
print(f'Convergence time : {ct} min')
```

```
Convergence time : 62.0 min
```

```
plot_positioning_series(sol=unc_sol,ct=ct,color='black')
```



Let's take a look at the post-fit residuals. Unlike the IF approach, we have 4 residuals for each satellite in our initial dataframe, corresponding to the number of observations processed. To use them in our function, replace the names according to the pattern in the cell below. The *plot_residuals* function is based on the presence of the *v* column for phase observations and *vc* for code observations.

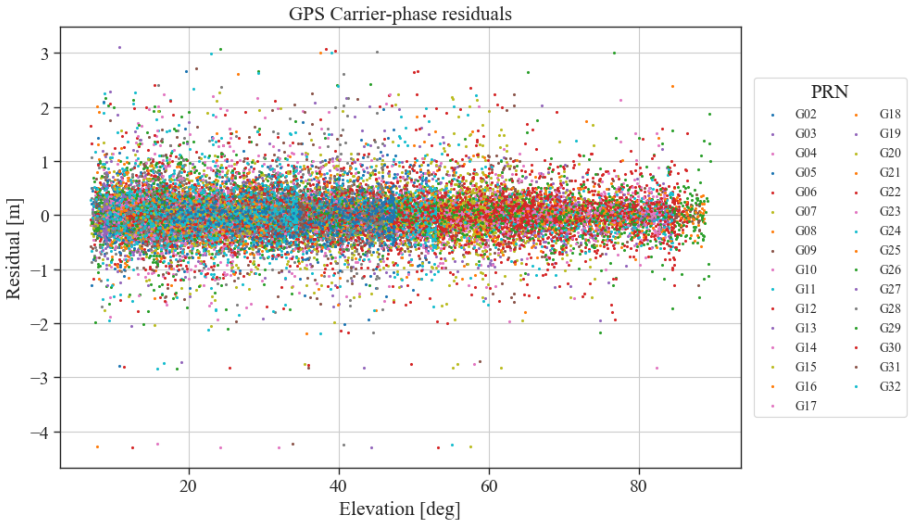
```
residuals = uncombined_results.residuals_gps.copy()
```

```
df_tail(residuals[[c for c in residuals.columns if c.startswith(('vc','vl
    '))]],nrows=5)
```

sv	time	vc1	vl1	vc2	vl2
G17_2_4	2024-02-04 23:59:30	-0.173	0.037	0.230	0.040
G19_3_6	2024-02-04 23:59:30	-0.501	0.035	0.676	0.038
G21_1_2	2024-02-04 23:59:30	-0.090	0.025	0.207	-0.007
G28_2_6	2024-02-04 23:59:30	-0.074	0.041	-0.149	0.050
G31_5_1	2024-02-04 23:59:30	-0.050	0.048	0.371	0.085

```
residuals['vc'] = residuals['vc1'] # or vc2
residuals['v'] = residuals['vl1'] # or vl2
```

```
plot_residuals(df=residuals,obs_type='phase',x_axis='elev',skip_obs=10,
    cut_init_time=10)
```



As I mentioned in the *Introduction* notebook, working in GNX-py mostly involves creating a configuration class and passing it to the management class. As you can see, this is also the case with PPP. I encourage you to try out other positioning modes and frequencies. If you are interested in how all this was implemented, I recommend that you study the scripts in the $gnx-py.ppp$ module. In the next notebook, we will discuss how to use GNX-py to monitor the quality of GNSS satellite orbits.

GNX Orbit module

The orbits module of the GNX-py library allows you to compare different GNSS orbit products. In this tutorial, I will discuss the components of the module and show you how to use the code to analyze GNSS products. The core of the orbits module is the SISController class, which integrates a number of GNX-py functions and allows us to compare two GNSS orbit products, both broadcast and precise. The class generates satellite coordinates and clocks from two specified sources and then presents the results in the form of coordinate differences, clocks, and coordinate differences converted to the local satellite system, i.e., in the form of radial, cross-track, and along-track differences. In addition to the above differences, the class also calculates the Signal in Space Error (European Union Agency for the Space Programme 2021), or in GPS nomenclature: User Range Error (National Executive Committee for Space-Based Positioning, Navigation, and Timing 2023). These values are the square root of the weighted sum of the squares of the individual components: clock and position differences. The user can adjust the weights for individual components or use the default values. The class also allows for the simultaneous analysis of DCB products assigned to each orbit. In this case, the DCB differences are included in the clock error in the resulting SISE/URE.

This notebook is used to present the use of the orbits module. In the folder, you will find the plotting_tools.py script, which provides default functions for visualizing the results. In the orbit_data folder, you will find the orbit files that I will use for the presentation. The algorithms and implementation details can be found in the next section.

Algorithms and methods

When implementing methods and algorithms, I relied on information described in (Montenbruck et al. 2018, 2015).

Calculations in the SISController class can be summarized in the following points.

- Generate satellite coordinates and clocks for the given epoch
- Determine the difference between the above
- Create a local satellite layout based on the reference orbit
- Convert errors ΔXYZ to ΔRAC
- Using ΔRAC, calculate the signal in space error
- Return an array with the calculated differences and SISE

Now I will discuss the approaches used for satellite coordinates and in the next section, methods of handling satellite clocks and DCB's. They require a special approach that is worth discussing separately.

Satellite coordinates

Let's consider a comparison between the broadcast and precise orbits. Suppose we want to investigate how accurate the broadcast orbit and clock are in relation to the precise final product. We want to obtain coordinate errors and errors in the local satellite system in order to better understand the distribution of errors in the orbital plane. The coordinates we obtain from the navigation message data are referenced to the satellite antenna phase center (APC), while the coordinates in SP3 products are referenced to the satellite center of mass (COM). Therefore, we cannot directly compare the two sets of coordinates because they are referenced to different points. A direct comparison of these two types of coordinates will lead to offsets of approximately 1-2m. Fortunately, analysis centers provide ANTEX files with the .atx extension, which contain PCO (phase center offsets) that we can use to refer our coordinates to a single location. If we apply PCO into the precise COM coordinates we obtain APC coordinates. The .ATX file is necessary for the correct execution of the task we are considering here. GNX-py will also perform the calculations if we do not provide it, but as I mentioned, the results obtained in this case will be subject to a large error. GNX-py enables reading ATX files. Let us now discuss the algorithms used.

In .ATX files, we find PCO for satellites in a local satellite system, created by

three vectors (Teunissen and Montenbruck 2017).

$$e_1 = \frac{r_{sat}}{|rsat|}$$
$$e_2 = e_1 \times \hat{e}$$
$$\hat{e} = \frac{r_{sun} - r_{sat}}{|r_{sun} - r_{sat}|}$$
$$e_3 = e_2 \times e_1$$

Where: - r_{sat}, $|r_{sat}|$ are satellite position vector and norm of that vector - r_{sun} is the Sun position vector - e_1 is the unit vector pointing to the geocentre - $\hat{e}$ is the unit vector pointing from tha satellite to sun - e_2 is the cross product of e_1 and $\hat{e}$ - e_3 completes the right handed system

Having satellite PCO in satellite fixed system, we can bring satellite COM coordinates into APC coordinates by creating rotation matrix:

$$rot = [e_3, e_2, e_1]$$

Denoting PCO as ΔAPC and satellite COM coordinates as XYZ_{COM} we obtain APC coordinates - XYZ_{APC} by applying:

$$XYZ_{APC} = XYZ_{COM} + rot \cdot \Delta APC$$

With the APC coordinates of the satellites, we can simply subtract them and convert the differences obtained into differences in the satellite's orbital plane. To do this, we will use the orbital reference system, created by the R, A, and C vectors:

$$R = \frac{r_{sat}}{|rsat|}$$
$$A = \frac{r_{sat} \times v_{sat}}{|r_{sat} \times v_{sat}|}$$
$$C = e_3 \times e_1$$

Where v_{sat} is the satellite velocity vector. The differences Δ_{RAC} are obtained by (HofmannWellenhof et al. 2008):

$$\Delta R = R \cdot \Delta XYZ$$
$$\Delta A = A \cdot \Delta XYZ$$
$$\Delta C = C \cdot \Delta XYZ$$

With the differences determined in this way, the software uses the formula for SISE/URE:

$$SISE = \sqrt{(w_R \cdot dR^2 + w_{dt} \cdot dT^2 + w_{AC} \cdot (dA^2 + dC^2) + w_{RdT} \cdot dR \cdot dT)}$$

Where w_R, w_{dT}, w_{AC}, w_{RdT} are the weights for individual components. w_{RdT} is used to introduce the correlation between the radial error and the clock error

into the equation. As I mentioned, we will move on to how to deal with satellite clocks in the next section. It is worth noting that the RTN system is created using the current position and velocity of the satellite. In the case of broadcast orbits, we have access to both of these values, because satellite velocities can be obtained using a classic algorithm for converting Keplerian orbit elements to ECEF coordinates. However, in the case of SP3 orbits, if the velocities are not specified in the file, which is often the case, we do not have access to the satellite velocities. Simply differentiating the positions between two epochs, separated by, for example, 30 seconds or even just 1 second, will not allow us to precisely determine the satellite's velocity from the SP3 coordinates. You can see for yourself by trying to generate and differentiate coordinates from the broadcast orbit, and then compare them with the velocities. The satellite velocity for SP3 data is needed both to create the RTN system and to determine the relativistic clock correction. GNX-py implements the following approach: assuming that we interpolate the satellite position at epochs t_0, t_1, t_2 ... t_n, for each time step we also interpolate the position t_n - 0.5μ s and t_n + 0.5μ s. Such a small time step means that the neighboring coordinates are very close to each other, and the difference between these positions gives the actual instantaneous velocity of the satellite at a given time step tn.

The above problem may not occur if we compare the precise and broadcast orbits, as we can create an RTN system based on the broadcast positions and velocities. However, when comparing precise products (e.g., Ultra Rapid and Final), we have to deal with this inconvenience. In GNX-py, in the SISController class, one of the orbits is "reference" and data from this orbit is used to create the RTN system - it is up to the user to decide whether it will be broadcast or precise orbit.

Satellite clocks and DCBs

I would like to discuss satellite clocks and DCBs separately, due to the special treatment they receive. Precise and broadcast offsets of satellite clocks are characterized by slight differences in relation to global time scales, such as TAI and UTC. Broadcast ephemerides are referenced to the time scale specific to the constellation, similarly, precise products are associated with a product-specific scale. Thus, clock values should not be compared directly, due to this difference in the implementation of the time scale specific to a given GNSS. This bias can be removed by correcting the "raw" differences in clock offset values between two products by the average difference in clock offset values in a given epoch for the entire constellation under analysis Montenbruck et al. (2015). That is: at each epoch, we calculate the raw clock difference for each satellite, and the average clock difference for the entire constellation at that epoch. In GNX-py, we can use the mean or median to calculate the correction value, or ignore this bias entirely.

In GNX-py, for orbit and clock error analysis, we can also enable comparison of transmitted TGD broadcasts and DCB from precise .BIA/.BSX products.

They can (and should) be treated in the same way as clocks, i.e., with the application of the average difference (TGD-DCB) per epoch. Currently, the algorithm supports the use of both DSB and OSB products for comparisons with TGD broadcasts, but in the latter case, OSB is converted to DSB. Since the transmitted DCBs are associated with specific signals for GPS and Galileo, i.e., combinations of IF C1W-C2W for GPS, C1C-C5Q, C1C-C7Q for F/NAV and I/NAV for Galileo, the comparison is made between the transmitted TGD and the corresponding bias from the precision products.

Satellite eclipse periods

GNX-py orbits module also provides the ability to determine the time perid when the satellite is in Earth's shadow. The entry of a satellite into the Earth's shadow can cause effects such as a disturbance in motion visible as an increase in the errors of orbital components, or a jump in the value of clock errors caused by a change in temperature. This functionality can be useful for users who want to use the module to study the stability of satellite orbits in these eclipse periods. In GNX-py, the eclipse period is determined by formulas (Mervart 1995):

$$cos(\phi) = \frac{r^{sat} \cdot r^{sun}}{r^{sat} \cdot r^{sun}} < 0$$

$$\sqrt{1 - cos^2(\phi)} < a_e$$

Where r^{sat} and r^{sun} are vectors from the geocentre to the satellite and Sun respectively, and r^{sat} and r^{sun} are norms of these vectors, and a_e is the mean equatorial radius of Earth. If these equations are satisfied, we are dealing with an eclipse period.

Let us conclude the theoretical introduction here and move on to examples of using GNX-py for orbit analysis. In the following sections, we will refer to the information discussed here.

Example 1 - Broadcast & Precise products comparison

As with SPP and PPP, we use a configuration and management class to analyse orbits. However, the configuration class differs slightly from those used in the two previous cases. In the $SISConfig$ class, we have the following parameters: - *orb_path_0* and *orb_path_1* - Broadcast or SP3 orbit files - *dcb_path_0* and *dcb_path_1* - DCB .BIA/.BSX or Broadcast files - *atx_path* - ANTEX file - *systems* - G or E - system to be used - *gps_mode* and *gal_mode* - GNSS signals, essentially a choice between PCO and, in the case of Galileo, INAV and FNAV orbits - *interval* - interval at which to calculate satellite coordinates - *apply_eclipse* - whether to calculate the time the satellite spends in the Earth's shadow - *extend_eclipse* - whether to extend the eclipse flag -$extension_time$ - time to extend the eclipse duration flag, default 30 min - *apply_satellite_pco* -

whether to enter satellite PCO - *clock_bias* - whether to correct the clock error by the average bias per epoch - *clock_bias_function* - which function to use to calculate the bias: choose between "meand" or 'median' - *tlim* - time range for calculating satellite coordinates

```
import gps_lib as gnx
from datetime import datetime
import numpy as np
import matplotlib.pyplot as plt
from plotting_tools import df_head
np.set_printoptions(threshold=np.inf, linewidth=200, suppress=True,
    precision=3)
```

Let us examine the errors of the IGS broadcast orbit against the final CODE products. Let us assume that we are interested in the orbit performance for a single-frequency user. We will select a 5-minute interval for calculating coordinates, correct the clock for the average bias per epoch, and limit our analysis to 12 hours to save time.

```
ORBIT_0 = '../data/BRDC00IGS_R_20240350000_01D_MN.rnx'
ORBIT_1 = '../data/COD0OPSFIN_20240350000_01D_05M_ORB.SP3'

ATX = '../data/ngs20.atx'
DCB_1 = '../data/COD0OPSFIN_20240350000_01D_01D_OSB.BIA'
TLIM = [datetime(2024,2,4,0,0,0), datetime(2024,2,5,0,0,0)]
```

We fill in the configuration class: $SISConfig$, according to previous assumptions.

```
config = gnx.SISConfig(orb_path_0=ORBIT_0,
                    dcb_path_0=ORBIT_0,
                    orb_path_1=ORBIT_1,
                    dcb_path_1=DCB_1,
                    atx_path=ATX,
                    interval=30,
                    system='G',
                    gal_mode='L1L2',
                    clock_bias=True,
                    clock_bias_function='mean',
                    apply_eclipse=True,extend_eclipse=True,extension_time
    =30,
                    compare_dcb=False,
                    tlim=TLIM)
```

We start processing with the *run* method. The result is a dataframe containing errors/differences between orbit 0 and orbit 1. I'm using the term 'differences' here because the word 'errors' implies that one orbit is worse than the other. This does not have to be the case, unless we deliberately treat one orbit as a

reference. In our case, however, this is exactly what we do – we treat the CODE final orbit as a reference relative to IGS broadcast orbit.

```
df = gnx.SISController(config=config).run()
```

```
[diag] G01: epochs=2881, NAV=14, in_window=2881
```

The columns dR, dA, dC, dt in our dataframe are errors in the Radial, Cross-Track and Long-track planes, and the satellite clock. $dTGD$ are errors in the DCB value – for the broadcast orbit, this will be the transmitted TGD, for DCB – the value from the .BSX file for a given satellite. The columns $sisre_{orb}, sisre, sisre_{notgd}$ are signal-in-space error values calculated according to the formula presented in the introduction. The suffixes *_orb and _notgd* mean that the given value was calculated without taking into account the clock error and without taking into account the TGD/DCB error.

```
print(df.columns.tolist())
```

```
['dR', 'dA', 'dC', 'dt', 'dt_mean', 'dTGD', 'dTGD_mean', 'sisre', '
    sisre_orb', 'sisre_notgd', 'dx', 'dy', 'dz', 'eclipse_raw', 'eclipse
    ']
```

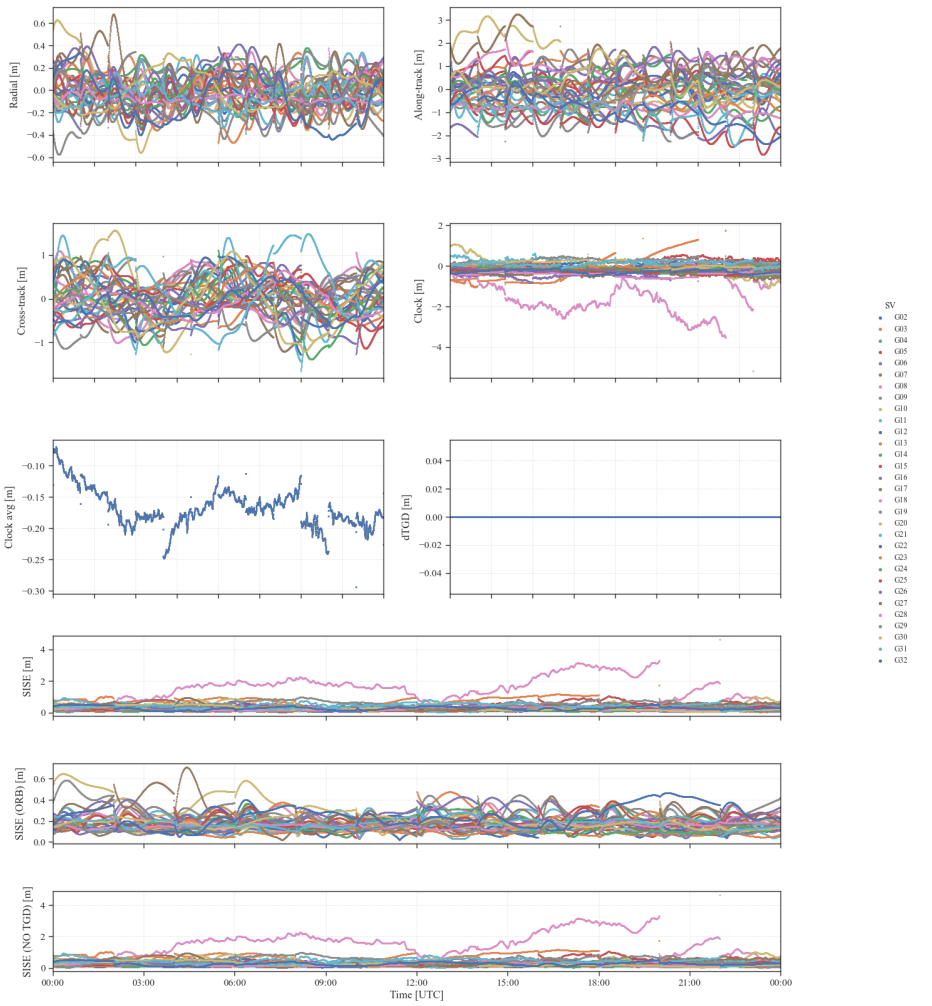
```
df_head(df,nrows=3,ncols=6)
```

sv	time	dR	dA	dC	dt
G02	2024-02-04 00:00:00	0.026	0.694	0.084	-0.030
G02	2024-02-04 00:00:30	0.192	1.777	-0.475	0.026
G02	2024-02-04 00:01:00	0.173	1.711	-0.449	0.033

The *plotting_tools.py* file contains auxiliary functions for visualising our results. The graph shows all $SISE$ components and the three $SISE$ values discussed above. You can assess the impact and behaviour of each component.

```
from plotting_tools import plot_sisre
```

```
plot_sisre(df=df,out_path='./figures',name='myorbits')
```



In the top three rows, you can see all the components that make up $SISE$. The *Clock avg* graph is the correction applied to the clock error - the average clock error across the entire constellation per epoch.

Additionally, in the *stat_tools.py* file, you will find the *orbit_stats* function, which allows you to calculate basic statistics for a selected column or set of columns. Using the *plot_stats* function, you can visualise the obtained statistics on a bar chart.

```
from stat_tools import orbit_stats
```

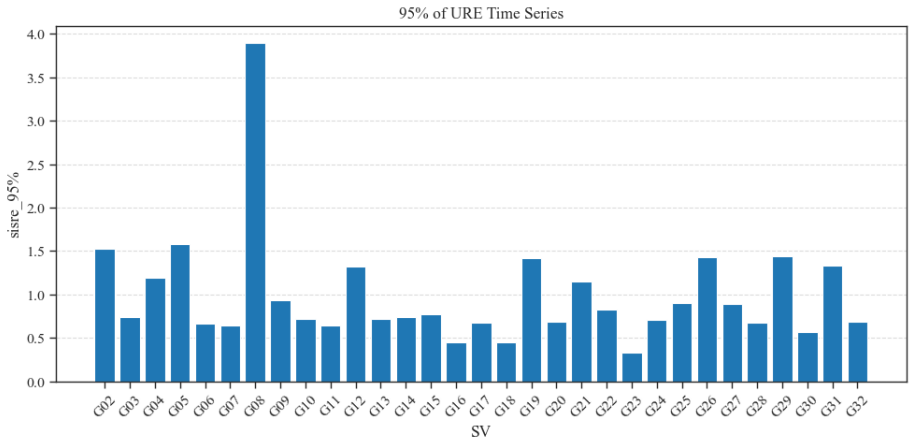
```
stats = orbit_stats(df=df,columns='sisre')
```

```
df_head(stats,nrows=3,ncols=6)
```

sv	sisre_min	sisre_max	sisre_mean	sisre_median	sisre_95%
G02	1.015	1.594	1.319	1.320	1.525
G03	0.113	1.232	0.421	0.419	0.741
G04	0.553	1.266	0.820	0.799	1.191

```
from plotting_tools import plot_sv_stats
```

```
plot_sv_stats(stats_df=stats,column_stat='sisre_95%',kind='bar',title='
    95% of URE Time Series')
```




```
(<Figure size 1000x500 with 1 Axes>,
 <Axes: title={'center': '95% of URE Time Series'}, xlabel='SV', ylabel='
    sisre_95%'>)
```

You have probably already noticed that clock error and TGD have a key impact on the size of SISE. In fact, TGD error sometimes exceeds the scale of the clock error itself. To learn more about this phenomenon, you can use the GNXpy library to analyse DCB/TGD itself. As reported by X, among others, we are dealing with either OSB or DCB biases (we mentioned them briefly in the notebook dedicated to PPP). X discusses the relationship between TGD transmitted in broadcast messages and DSB, and between OSB and DSB. We can easily compare DSB with TGD, and convert OSB to DSB, thus obtaining three values that we can treat as relating to the same phenomenon - the hardware delay of the satellite transmitter.

Let us open and compare the TGD and DCB files that we processed in our *SISController*. Additionally, let us open the .BIA file with the final OSB biases. We will load the files in *GNSSDataProcessor* and perform selections and visualisations in pandas and matplotlib.

```
dcb_1, dcb_type = gnx.GNSSDataProcessor2(sys='G').read_bia(path=DCB_1)
DCB_2 = '../data/COD0OPSFIN_20240350000_01D_01D_OSB.BIA'
dcb_2, dcb2_type = gnx.GNSSDataProcessor2(sys='G',galileo_modes='L1').
    read_bia(path=DCB_2)
print('DCB TYPE: ', dcb_type, '\n')
print('DCB 2 TYPE: ', dcb2_type, '\n')
```

```
DCB TYPE:  DSB

DCB 2 TYPE:  OSB
```

From the opened DCB files, we only extract those for GPS.

```
gps_dcb = dcb_1.loc[(dcb_1.index.get_level_values('sv').str.startswith('G
    '))]
gps_dcb2 = dcb_2.loc[(dcb_2.index.get_level_values('sv').str.startswith('
    G'))]
df_head(gps_dcb,nrows=3,ncols=5)
```

sv	time	BIAS_C1C_C2C	BIAS_C1C_C2L	BIAS_C1C_C2P
G	2024-02-04 00:00:00	-22.406	0.000	0.000
G01	2024-02-04 00:00:00	0.000	0.000	0.000
G02	2024-02-04 00:00:00	0.000	0.000	0.000

Now we need to load the broadcast orbit for GPS.

```
nav = gnx.GNSSDataProcessor2(sys='G',nav_path=ORBIT_0,galileo_modes='L1L2
    ').load_broadcast_orbit()
brdc = nav.gps_orb
```

```
brdc=brdc.reset_index()
brdc['sv'] = brdc['sv'].str[:3]
brdc = brdc.set_index(['sv','time'])
sats=brdc.index.get_level_values('sv').unique().tolist()
```

As described in X, $DSB(obs1, obs2) = OSB(obs2) - OSB(obs1)$, while for TGD transmitted in a broadcast message $TGD = k2 \times DSB(C1W, C2W)$, where $k2 = \frac{-f2^2}{f1^2 - f2^2}$, $f1, f2 = 1575.42, 1227.60 MHz$ In our case, therefore, $obs1, obs2$ are C1W and C2W.

```
C=299792458
f1 = 1575.42e06
f2 = 1227.60e6
f2 = 1176.45e6
k2  = -f2**2/(f1**2-f2**2)
```

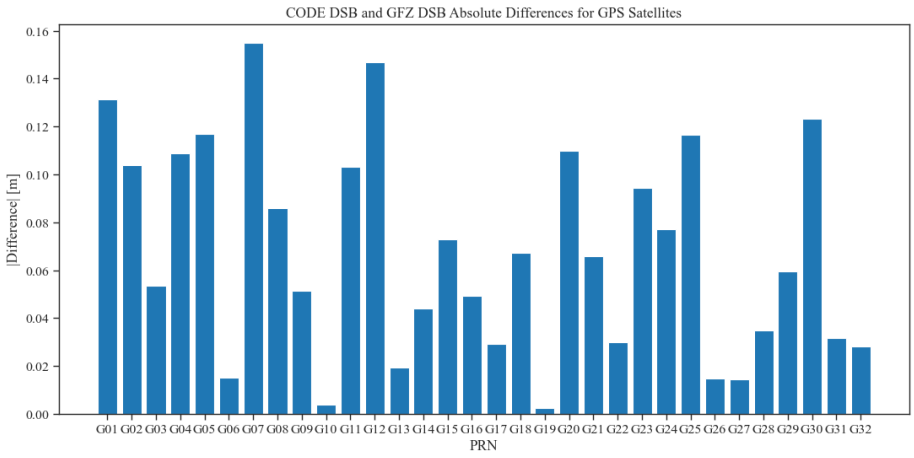
Let's write a simple code that will calculate and show us the differences in TGD. As you can see, the differences between TGD and DCB from the .BSX file and those converted from the .BIA file amount to several metres, while the differences between the values from the .BSX/.BIA files are less than 20 cm, bearing in mind that they originate from different centres and are different types of products: rapid and final.

```
d1, d2, d3 = [], [], []
ns2m = 1e-09*C
for sat in sorted(sats):
    tgd = (brdc.loc[sat,'TGD'].unique()/1e-09)[0].round(4)
    dcb = k2*gps_dcb.loc[sat,'BIAS_C1W_C2W'].unique()[0].round(4)
    dcb2 = k2*(gps_dcb2.loc[sat,'OSB_C1W'] - gps_dcb2.loc[sat,'OSB_C2W']
    ).unique()[0].round(4)
    d1.append(np.round(tgd-dcb,3)*ns2m)
    d2.append(np.round(tgd-dcb2,3)*ns2m)
    d3.append(np.round(dcb-dcb2,3)*ns2m)
```

Using matplotlib, let us visualise our differences on a bar chart.

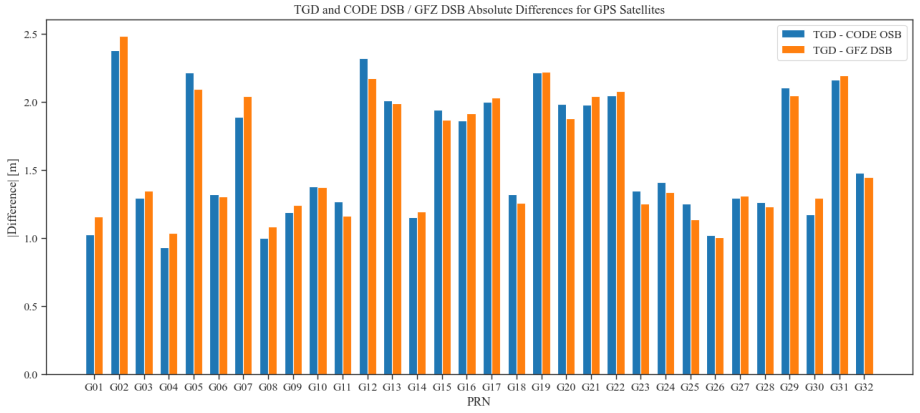
```
plt.figure(figsize=(13,6))
plt.title('CODE DSB and GFZ DSB Absolute Differences for GPS Satellites')
plt.xlabel('PRN')
plt.ylabel('|Difference| [m]')
plt.bar(sorted(sats), abs(np.array(d3)))
```

```
<BarContainer object of 32 artists>
```



We calculated the differences between TGD and the two types of DSB. We will show them on a common graph using matplotlib. As you can see, the absolute differences are drastically greater than in the case of rapid and final product differentiation.

```python
import numpy as np
import matplotlib.pyplot as plt

svs = sorted(sats)
x = np.arange(len(svs))
width = 0.35

plt.figure(figsize=(13,6))
plt.title('TGD and CODE DSB / GFZ DSB Absolute Differences for GPS
    Satellites')
plt.xlabel('PRN')
plt.ylabel('|Difference| [m]')
plt.bar(x - width/2, np.abs(d1), width, label='TGD - CODE OSB')
plt.bar(x + width/2, np.abs(d2), width, label='TGD - GFZ DSB')

plt.xticks(x, svs)
plt.legend()
plt.tight_layout()
plt.show()
```



In this example, we used GNX-py functionality to calculate orbit errors and performed some additional DCB component analysis by combining GNX-py functions and popular Python tools. We provided numerical and visual results comparing two types of orbits and TGD/DCB values. You can try replacing the orbits and DCB sources in the configuration class and compare other products. In the dataframe resulting from the processing, you have an "eclipse" column, which is a flag indicating whether the satellite is in eclipse or not. For practice, you can investigate whether there is a correlation between any type of error and the eclipse period. I also encourage you to perform the same calculations for Galileo. Just keep in mind that the .BSX file has DCB values only for the signal pair: C1C-C5Q. In the next example, we will compare two SP3 orbits – rapid and final – along with their corresponding DCB products.

Example 2 - Rapid & Final products comparison

To compare two SP3 orbits, we use exactly the same configuration and management class as before, replacing only the paths to the files.

```
ORBIT_0 = '../data/GFZ0MGXRAP_20240350000_01D_05M_ORB.SP3'
DCB_0 = '../data/GFZ0MGXRAP_20240350000_01D_01D_DCB.BSX'
ORBIT_1 = '../data/COD0OPSFIN_20240350000_01D_05M_ORB.SP3'
DCB_1 = '../data/COD0OPSFIN_20240350000_01D_01D_OSB.BIA'
ATX = '../data/ngs20.atx'
TLIM = [datetime(2024,2,4,10,0,0), datetime(2024,2,4,16,0,0)]
```

```
config = gnx.SISConfig(orb_path_0=ORBIT_0,
                    dcb_path_0=DCB_0,
                    orb_path_1=ORBIT_1,
                    dcb_path_1=DCB_1,
                    atx_path=ATX,
                    interval=120,
                    system='G',
                    gal_mode='L1',
                    clock_bias=True,
                    clock_bias_function='mean',
                    apply_eclipse=True,extend_eclipse=True,extension_time
    =30,
                    compare_dcb=True,
                    tlim=TLIM)
```

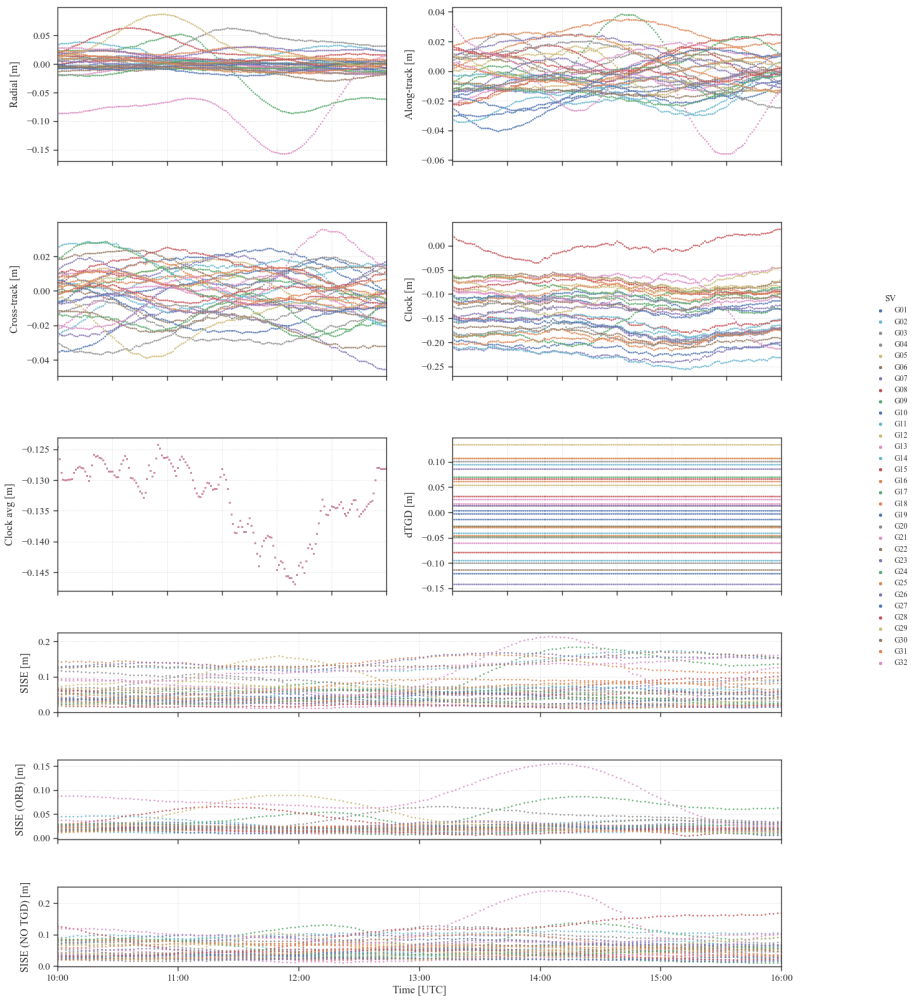
```
df = gnx.SISController(config=config).run()
```

```
df_head(df,nrows=3,ncols=6)
```

sv	time	dR	dA	dC	dt
G01	2024-02-04 10:00:00	-0.003	-0.025	-0.035	-0.191
G25	2024-02-04 10:00:00	-0.003	0.005	0.008	-0.072
G24	2024-02-04 10:00:00	-0.018	-0.014	0.018	-0.062

After obtaining the results, we will use the same functions as before for visualisation, but we will skip the additional steps we performed earlier – the analysis of DCB files.

```
plot_sisre(df=df,out_path='./figures',name='my_precise_orbits')
```

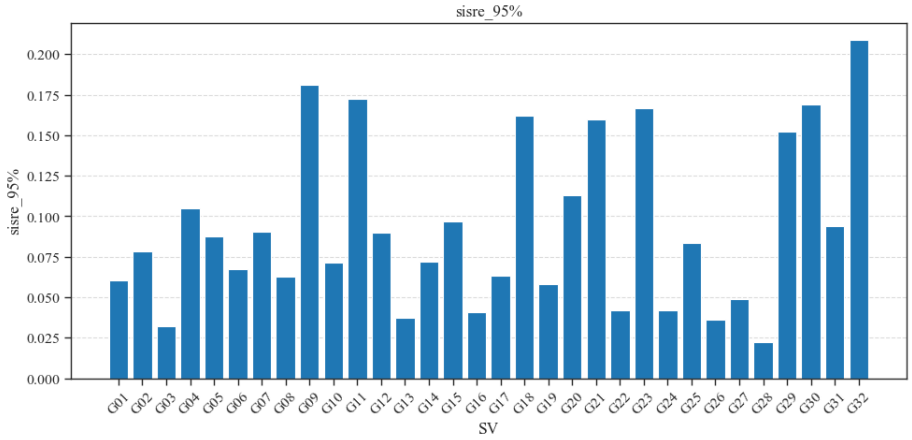



```
stats = orbit_stats(df=df,columns='sisre')
```

```
df_head(stats,nrows=3,ncols=6)
```

sv	sisre__min	sisre__max	sisre__mean	sisre__median	sisre__95%
G01	0.035	0.064	0.047	0.044	0.060
G02	0.024	0.090	0.051	0.056	0.078
G03	0.016	0.033	0.024	0.024	0.032

```
plot_sv_stats(stats_df=stats,column_stat='sisre_95%')
```




```
(<Figure size 1000x500 with 1 Axes>,
 <Axes: title={'center': 'sisre_95%'}, xlabel='SV', ylabel='sisre_95%'>)
```

The result, which indicates that the quality of rapid orbits is significantly better than that of final orbits, is in line with expectations. We observe 95% SISE at a level of 0.2 - 0.3 metres, while in the case of broadcast orbits it is definitely above 0.5 metres. Analysing the graph, we can see that the clock error and DCB have a significant impact on the error value. This may be useful information when planning PPP measurements. For practice, you can compare selected orbit and DCB products and also compare the results of PPP positioning using these products.

We have discussed the orbits module and its accompanying tools. You have learned about the possibilities offered by GNX-py in this area, how to configure processing, and how to use auxiliary functions for data visualisation. In the next chapter of this tutorial, we will move on to discussing the *ionosphere* module, which is used to analyse and model the ionosphere.

Ionospheric delay in GNSS observations

Ionospheric delay is one of the key factors degrading GNSS positioning. Long-established methods of dealing with it include modeling ionospheric delay or using a combination of observations at two frequencies – iono-free, which eliminates first-order ionospheric delay at the cost of increased observation noise (Teunissen and Montenbruck 2017; HofmannWellenhof et al. 2008). Ionospheric modeling and monitoring is a broad field of GNSS-related research (e.g (Mannucci et al. 1998; Schaer 1999)). GNX-py offers a number of functionalities for measuring, modeling, and monitoring the spatial and temporal scale of changes in the ionosphere. Below, we will present the theoretical background of the methods used and then move on to discuss the implemented algorithms.

Notation

- f_i: carrier frequency i [Hz],
- $\lambda_i = c/f_i$ - wavelength,
- c: speed of light
- $N_e(s)$: electron density [el/m^3]
- STEC $= \int_{\text{LOS}} N_e(s)\, ds$ [el/m^2],
- 1 TECU $= 10^{16}$ el/m^2
- ρ: geometric range,
- T: tropospheric delay
- $\delta t_r, \delta t_s$: receiver/satellite clock offsets
- P_i: code measurement on f_i [m]
- $\Phi_i = \lambda_i \varphi_i$: carrier phase observation [m]
- $d^P_{(\cdot),i}, d^L_{(\cdot),i}$: hardware delays,
- N_i: carrier ambiguity

Definition of STEC

Ionospheric delay on the satellite-receiver path is defined as:

$$\text{STEC} = \int_{\text{LOS}} N_e(s)\, ds \qquad [\text{el/m}^2]$$

The commonly accepted unit for describing STEC is TECU, Total Electron Content Units. 1 TECU is $10^{16}[el/m^2]$.

Ionospheric delay

In the case of GNSS observations, we are most often interested in the delay in meters for code and phase observation data.

$$I_i = \frac{40.3 \cdot 10^{16}}{f_i^2}\,\text{STEC}$$

Here I_i is an ionospheric delay in the unit of meters for signal at frequency i. Ionospheric delay causes a "delay" in code observations and an "advance" in phase observations, so the correction is entered into the equation with the opposite sign.

$$P_i : +I_i \quad \text{(group delay)}, \qquad \Phi_i : -I_i \quad \text{(phase advance)}$$

Observation equations

Recording our code and phase observations as P_i and Φ_i, respectively:

$$P_i = \rho + c(\delta t_r - \delta t_s) + T + I_i + d^P_{r,i} - d^P_{s,i} + \varepsilon_{P_i},$$

$$\Phi_i = \rho + c(\delta t_r - \delta t_s) + T - I_i + \lambda_i N_i + d^L_{r,i} - d^L_{s,i} + \varepsilon_{\Phi_i}.$$

STEC measurement using GNSS is possible thanks to a combination of observations at two frequencies. The so-called geometry-free combination eliminates all non-dispersive factors affecting the observation, such as tropospheric delay.

$$P_{\mathrm{GF}} = P_1 - P_2 \;=\; (I_1 - I_2) + \Big[(d^P_{r,1} - d^P_{r,2}) - (d^P_{s,1} - d^P_{s,2})\Big] + \eta_P,$$
$$\Phi_{\mathrm{GF}} = \Phi_1 - \Phi_2 \;=\; -(I_1 - I_2) + (\lambda_1 N_1 - \lambda_2 N_2) + \Big[(d^L_{r,1} - d^L_{r,2}) - (d^L_{s,1} - d^L_{s,2})\Big] + \eta_\Phi.$$

The remaining biases are differences in hardware delays between two frequencies for satellites and receivers, and in the case of phase observations, differences in phase ambiguities. Satellite DCB can be introduced as an external correction from daily or monthly products of various IAACs, e.g., CODE. Receivers DCB can also be obtained from external sources, but they are obviously not available for every receiver, so they must be estimated for absolute STEC measurement. In GNX-py, this is done either through a calibration procedure based on an external model, e.g., GIM, from the day of observation or the day preceding it, or through PPP in UDUC mode and determining biases from OSB delays obtained in the estimation process. Obtaining STEC observations is performed by applying following formulas:

$$\mathrm{STEC}^{(P)} = \frac{P_{\mathrm{GF}} - \mathrm{DCB}^P_{12}}{40.3\left(\frac{1}{f_1^2} - \frac{1}{f_2^2}\right)}$$

and from phase observations (with constant B_Φ):

$$\mathrm{STEC}^{(\Phi)} = -\frac{\Phi_{\mathrm{GF}} - B_\Phi}{40.3\left(\frac{1}{f_1^2} - \frac{1}{f_2^2}\right)}$$

B_Φ is a linear combination of phase ambiguities of signals at both frequencies resulting from the creation of a geometry-free combination. Due to this bias, the GF observation derived from the phase, although not affected by noise to the same extent as the code observation, is scaled by a bias unique to a given satellite. To remove it, a phase-to-code leveling technique is used, i.e., fitting the phase observation into the code observation (Mannucci et al. 1998). This allows for obtaining a precise, noise-free STEC on a real scale. It is also possible to use denoising filters on code observations and not to rely on phase observations at all.

$$\tilde{L}_4^{(n)} = L_4^n + \frac{1}{K}\sum_{k=1}^{K}\Big[(L_4^{(n)} - P_4^{(k)})\Big]$$

where: - $\tilde{L}_4^{(n)}$ – smoothed carrier phase geometry-free observation at epoch n, - $P_4^{(k)}$ – code geometry-free observation at epoch k, - $L_4^{(k)}$ – phase geometry-free observation at epoch k, - K – number of samples used for smoothing

For smoothing process window size parameter - K is defined,thus, the smoothing procedure for observations in epoch n is performed by adding to observation L_4^n the average difference between this observation and K code observations in epochs adjacent to n, i.e., $k_1 = n-K/2, K = n+K/2 $. You can find an example below. Alternatively, instead of phase leveling, we can use various techniques for noise reduction in code observations, such as Chebyshev and Savitzky-Golay filters (Savitzky and Golay 1964; Smith 2025; Arikan et al. 2008).

In ionosphere modeling using GNSS, it is usually assumed that electrons are concentrated in an infinitely thin layer of the atmosphere at a certain altitude, e.g., 350, 400, or 450 km. The conversion of measured STEC to VTEC and vice versa using a mapping function is based on this assumption. Assume shell height h_{ion} and Earth radius R_E:

$$M(E) = \frac{1}{\sqrt{1 - \left(\frac{R_E}{R_E + h_{\text{ion}}} \cos E\right)^2}}$$

$$\text{STEC} \approx M(E) \cdot \text{VTEC}, \qquad \text{VTEC} \approx \frac{\text{STEC}}{M(E)}$$

This is a simple mapping function, but you can find several others, sometimes more advanced ones, in the literature on the subject. In the next section, we will move on to an example of using GNX-py to measure STEC for a single station and compare ionosphere models using built-in tools.

Example of STEC processing in GNX-py

Let's now move on to an example of STEC measurement in GNX-py. The procedure is as follows: - we obtain the coordinates of the satellites at the time of signal transmission (see the Introduction notebook) - we introduce corrections to the observations (we only need to introduce what is not common to both frequencies - PCO of satellites and receiver, and - for already differentiated observations - DCB of satellites and receiver. - detection of cycle slips and loss of tracking - we must detect them, otherwise they will cause problems during leveling. In GNX-py, we divide observations to the satellite into arcs between cycle slips and tracking gaps, which is why you see indexes such as G01_0_1 in our dataframes. They should be interpreted as: observation to satellite G01, 0 - index of the arc between tracking losses, 1 - index of the arc between cycle slips - phase to code leveling - we smooth observations according to the method described above - conversion of meters of delay to TEC

For simplicity, we have enclosed the entire pipeline in a single class—$TECSession$—which is managed by the configuration class—$TECConfig$. In

the configuration class, we place the processing parameters, which are then sent to *TECSession*. This gives you a ready-made pipeline that you can modify according to your needs. You can view all the parameters of the configuration class in the orbits/config.py module. Here, we will only fill in those necessary for our measurements.

```
import gps_lib as gnx
import numpy as np
import warnings
from datetime import datetime
from tools import df_head
warnings.simplefilter(action='ignore', category=FutureWarning)
np.set_printoptions(threshold=np.inf, linewidth=200, suppress=True,
    precision=3)
```

```
OBS_PATH='../data/BOR100POL_R_20240350000_01D_30S_MO.rnx'
NAME = OBS_PATH.split('/')[-1][:4]


NAV = '../data/BRDC00IGS_R_20240350000_01D_MN.rnx'
SP3_LIST=[
'../data/COD0OPSFIN_20240340000_01D_05M_ORB.SP3',
'../data/COD0OPSFIN_20240350000_01D_05M_ORB.SP3',
'../data/COD0OPSFIN_20240360000_01D_05M_ORB.SP3']
ATX_PATH='../data/ngs20.atx'
DCB_PATH= '../data/COD0OPSFIN_20240350000_01D_01D_OSB.BIA'
GIM='../data/COD0OPSFIN_20240350000_01D_01H_GIM.INX'

TLIM = [datetime(2024,2,4,0,0,0),
        datetime(2024,2,4,23,59,30)]

SYS= 'G'
DOY=35
```

We will use the observation file from the BOR1 station for processing. Since the data from the station was used to generate the GIM, its DCB for GPS and Galileo is present in the IONEX file. If we did not have the receiver's DCB, we could either perform "on the fly" calibration using our GIM, or use our own DCB obtained, for example, from PPP measurements in undifferenced uncombined mode (see PPP notebook). Experiments with GNX-py prove that the latter method gives quite good agreement with the DCB from the IONEX file, typically at the level of 0.5-1.5 ns. For safety, you can always average the DCB from both sources. An example of calibration can be found in a separate notebook. In this example, we will perform an STEC measurement and calculate it from three additional models: GIM, Klobuchar, and NTCM. Next, we will analyze the obtained values.

We will obtain the satellite coordinates from SP3 files, while the PCO of the

satellites and receiver from the ANTEX file. Below you will find our configuration class. I have added comments to the selected elements so that you know what we are doing and why.

```
config = gnx.TECConfig(
   obs_path=OBS_PATH,# Observation file
   nav_path=NAV, # navigation file - we need it for screening and for Klobuchar/NTCM model computation (GPSA, GPSB and GAl coeffs)
   sp3_path=SP3_LIST, # satellite coordinates
   gim_path=GIM, # IONEX for GIM STEC computation and source of satellite DCB

   sys=SYS, # SYSTEM/S used - STEC is measured for GPS and GAL separately, so here we should use only {G} or {E}
   gps_freq='L1L2', # Signals used
   windup=True, # we  need windup for STEC measurement
   rel_path=False,#we dont need relativistic path range correction for STEC measurement
   sat_pco='los',
   rec_pco=True,# ... and reciever pco
   atx_path=ATX_PATH, # ... so here we have ATX file
   add_dcb=True, # we want to use DCB


   add_sta_dcb=True, # we want to use station DCB
   rcv_dcb_source='gim', # possible: gim | calibrate | defined, here we use station DCB from GIM
   screen=True, # we want to screen for unhealthly satellites
    ev_mask = 20,


   station_name=NAME, # station name
   day_of_year=DOY, # Day of year - must have for Klobuchar/NTCM models


   ionosphere_model='gim', # ...using GIM
   use_iono_rms=True, # ... with STEC RMS
   compare_models=['ntcm','klobuchar'],# we'd like to compare NTCM and Klobuchar models


   troposphere_model=False, # we dont need troposphere correction


   use_gfz=True,# GFZRNX for fast obs file reading


   leveling_ws=50,# window sie for leveling (K parameter) 50 is safe value

```

```
40
41    median_leveling_ws=50 # the same parameter but for median moving
       window leveling - alternative to phase leveling
42    )
```

Okay, after creating the configuration, we introduce it as an attribute to the TECSession class and run the calculations using the run() method.

```
1 controller = gnx.TECSession(config=config)
2 obs_tec = controller.run()
```

The resulting obs_tec dataframe contains all observational data and preprocessing results. The calculated STEC and modelled corrections can be found in the columns below.

```
1 df_head(obs_tec[['ion','klobuchar','ntcm']],nrows=5,ncols=5) # ION = GIM
```

sv	time	ion	klobuchar	ntcm
G02_0_1	2024-02-04 00:00:00	1.754	1.939	1.504
G02_0_1	2024-02-04 00:00:30	1.765	1.945	1.507
G02_0_1	2024-02-04 00:01:00	1.776	1.952	1.509
G02_0_1	2024-02-04 00:01:30	1.786	1.959	1.512
G02_0_1	2024-02-04 00:02:00	1.797	1.966	1.514

Please note that all correction values are in metres of delay on L1, so we will convert them to TECU for the convention in the STEC analysis. Output STEC is in electrons, so we convert it to TECU by dividing it by 1^{16}.

```
1 obs_tec[[c for c in obs_tec.columns if '_tec' in c]] /=1e16
```

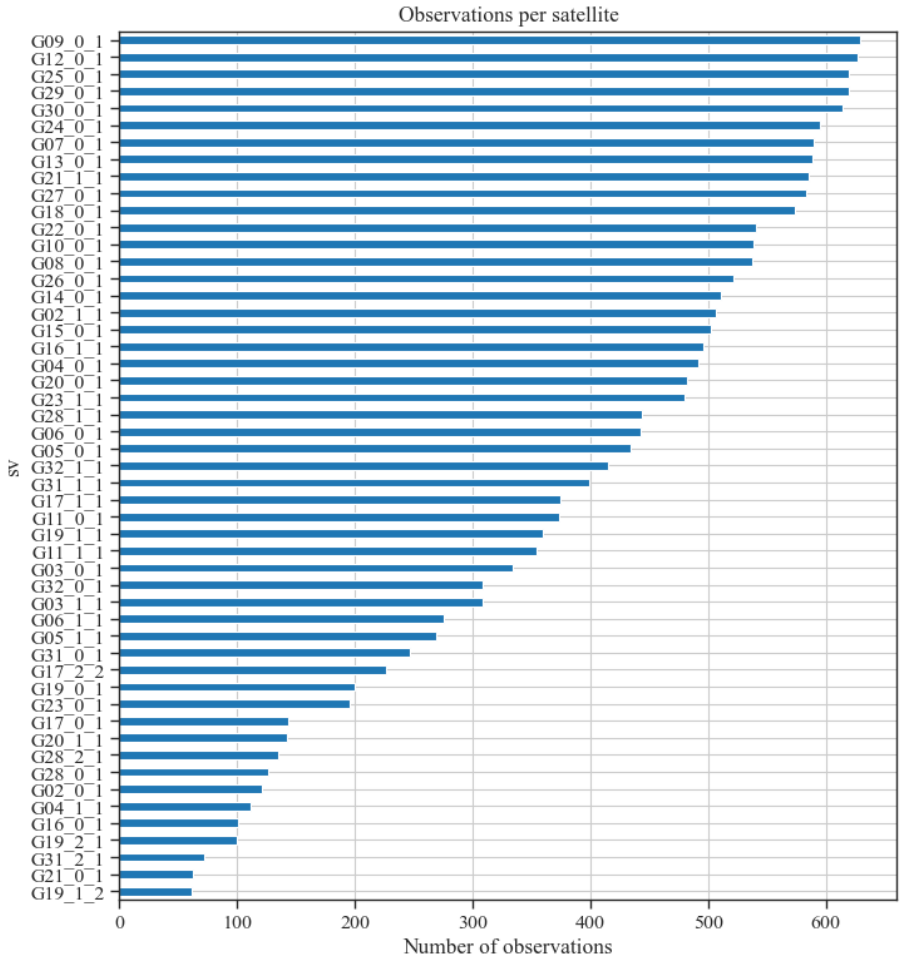
```
1 df_head(obs_tec[[c for c in obs_tec.columns if '_tec' in c]],nrows=5,
      ncols=7,floatfmt = ".2f",truncate_str=10)
```

sv	time	leveled_tec	code_tec	leveled_tec_sg	leveled_tec_cheby	phase_tec
G02_0_1	2024-02-04 00:00:00	8.24	7.48	7.54	6.38	-180.61
G02_0_1	2024-02-04 00:00:30	8.26	10.46	7.58	6.42	-180.58
G02_0_1	2024-02-04 00:01:00	8.18	7.89	7.63	6.47	-180.54
G02_0_1	2024-02-04 00:01:30	8.19	8.12	7.68	6.51	-180.52
G02_0_1	2024-02-04 00:02:00	8.12	9.00	7.72	6.56	-180.47

Let's take a look at the collection of continuous observations from satellites that we have gathered. We will select one of them to show the differences between code and phase STEC mentioned in the previous section.

```
obs_tec.groupby('sv').size().sort_values().plot(kind='barh', xlabel='
    Number of observations', title='Observations per satellite',figsize
    =(8,9),grid=True)
```

```
<Axes: title={'center': 'Observations per satellite'}, xlabel='Number of
    observations', ylabel='sv'>
```

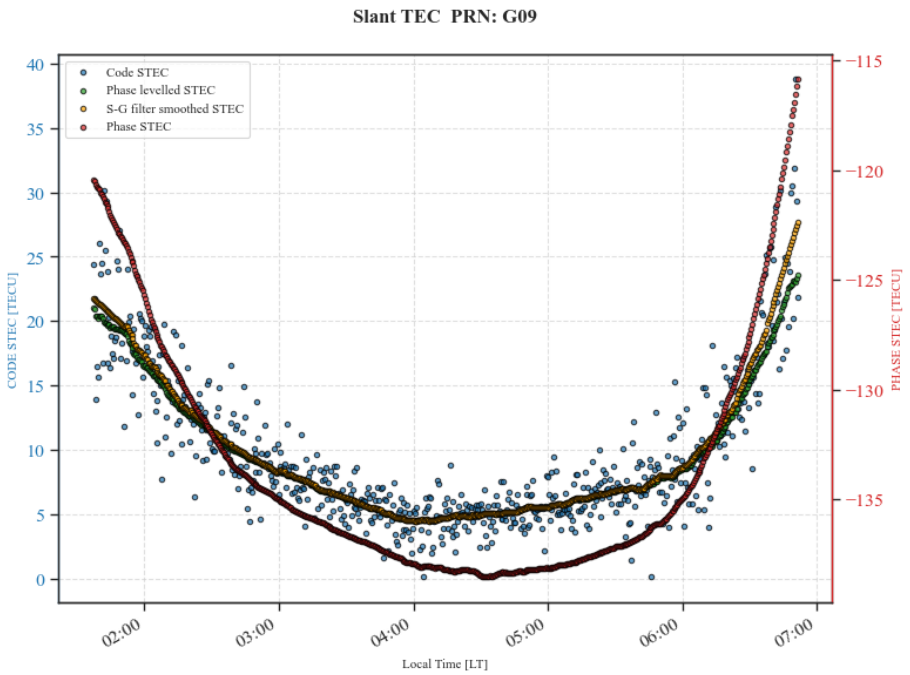


In the case of our station, I recommend taking a look at G09_0_1, because it has a lot of observations and you can also clearly see the effect of phase ambiguity.

To plot STEC for this satellite, you can use the auxiliary function $plot_stec$. Using the function you will see:

- STEC from code observations, significantly noisy,
- STEC smoothed with the Savitsky-Golay filter
- STEC from phase observations, levelled to STEC from code observations

```
from tools import plot_stec
plot_stec(
        obs_tec,
        "G09_0_1"
    )
```



With the STEC measured, corrected for the DCB of the satellites and receiver, and smoothed using phase observations, we can perform a series of activities. One of them may be the estimation of VTEC above the station during the observation period. To do this, let's use a tool from the *tools.py* module - VTECZenithEstimator. It uses the Whittaker–Eilers smoother to estimate the VTEC at the zenith from observations in a given epoch. Due to the specific nature of GNSS measurements, we rarely have the opportunity to observe a satellite directly at the zenith continuously, so we use interpolation on the observations closest to zenith. Whittaker–Eilers smoothing allows for a compromise between data fit and signal smoothness. VTECZenithEstimator smoothing parameters are *lam* and *order*. Below you'll find an brief explanation of their purpose. *lam* parameter controls the trade-off between fidelity to the input

data and the smoothness of the output. Small values of *lam* keep the smoothed series very close to the raw observations, with little suppression of noise. Large values of lam enforce stronger smoothness, producing a cleaner long-term trend but removing short-term variations. In practice, *lam* acts like a slider between "follow the data closely" and "ignore details and keep only the trend." *order* defines what kind of smoothness is enforced:

- $order = 1$ penalizes changes in slope, encouraging the output to be nearly constant within the chosen scale.
- $order = 2$ penalizes changes in curvature, encouraging the output to follow a nearly linear trend.

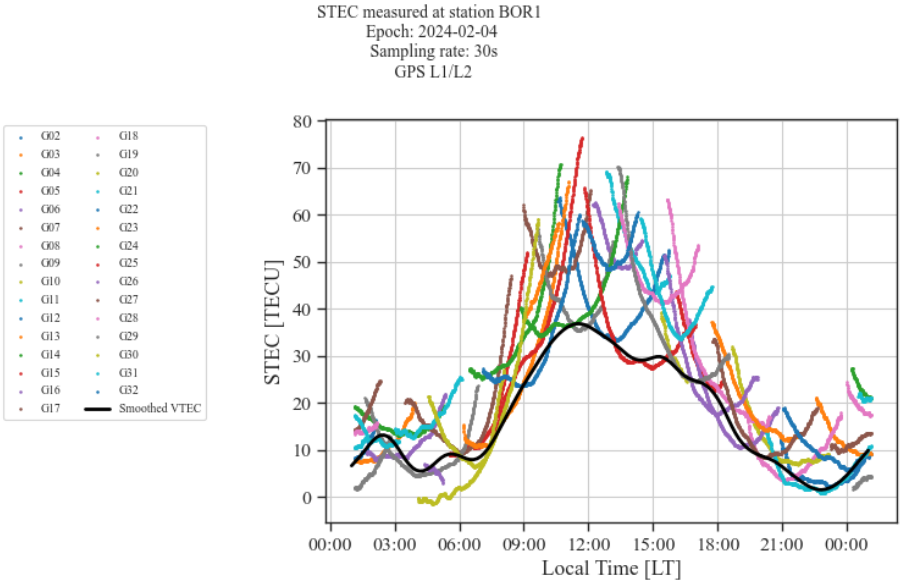
Higher orders continue this pattern, pushing the output toward low-degree polynomial shapes. In our case, $order = 2$ works good because it preserves linear trends while removing small-scale noise, and *lam* is tuned depending on how aggressive the smoothing should be.

```
from tools import VTECZenithEstimator
```

```
est = VTECZenithEstimator(
    df=obs_tec,
    time_col="LT", # LT or time
    elev_col="ev",# elevation column
    stec_col="leveled_tec", # STEC source column, try other sources
    sv_col="sv", # PRN identifier column/index
    min_elev_deg=60, # minimum elevation of satellites used for zenith
    VTEC interpolation
    resample_freq="600S", # resampling
    smooth_method="whittaker", # method
    smooth_params={"lam": 50, "order": 2}  # method parameters
)


df_zen, zen =est.run()
```

```
from matplotlib.dates import DateFormatter
import matplotlib.pyplot as plt
obs_tec['sat']=obs_tec.index.get_level_values('sv').str[:3] # removing
    arc signatures
fig, ax = plt.subplots(figsize=(8,5))
fig.suptitle(f'STEC measured at station {NAME} \nEpoch: 2024-02-04\n
    Sampling rate: 30s\n GPS L1/L2',fontsize=10)
for sv, gr in obs_tec.groupby('sat'):
    gr=gr.reset_index()
    ax.scatter(gr['LT'], gr['leveled_tec'], label=sv[:3],s=1)

    ax.set_ylabel('STEC [TECU]')
    ax.set_xlabel('Local Time [LT]')
    ax.grid()
```

```
    ax.xaxis.set_major_formatter(DateFormatter('%H:%M'))
ax.plot(zen['LT'],zen['vtec_smooth'],c='black',linewidth=2,label='
    Smoothed VTEC')
ax.legend(loc='best',ncol=2,bbox_to_anchor=(-0.2,1),fontsize=7)
plt.tight_layout()
```




```
df_head(zen,nrows=5,ncols=7,floatfmt = ".2f",truncate_str=10)
```

index	LT	vtec	ev	M	nsat	vtec_smooth
0.00	2024-02-04 01:00:00	7.52	84.00	1.00	1.00	6.60
1.00	2024-02-04 01:10:00	7.62	84.51	1.00	1.00	7.44
2.00	2024-02-04 01:20:00	7.39	82.91	1.01	1.00	8.30
3.00	2024-02-04 01:30:00	7.58	79.40	1.02	1.00	9.20
4.00	2024-02-04 01:40:00	7.50	75.30	1.03	1.95	10.14

In the *zen* dataframe, in the *vtec_smooth* column, you will find the estimated VTEC above the station. The technique presented here is not the most precise, but it allows us to obtain some very specific, even approximate, information from the measurements.

Remember that at the beginning we selected the *compare_models* parameter and chose the Klobuchar and NTCM models. We also calculated the GIM model, as we are taking the DCB from it anyway. Using the STECModelComparator tool, we can compare these three models in relation to the measured STEC. An example is provided below.

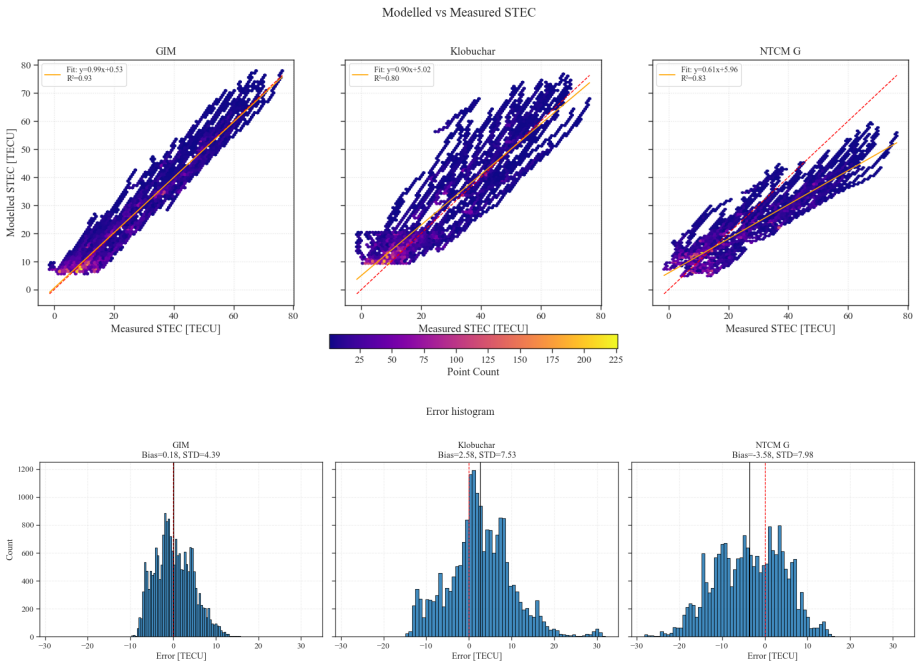
```
from tools import STECModelComparator
K = 1/0.16
cmp = STECModelComparator(
    df=obs_tec,
    x_col='leveled_tec',
    model_cols=['ion', 'klobuchar', 'ntcm'],     # add models you'd like to compare
    model_names=['GIM', 'Klobuchar', 'NTCM G'],  # Model labels for plotting
    scale=K,                                     # model scale, K = 1/0.16 - meters to TECU
    gridsize=80,
    mincnt=1
)
```

report() method returns an array with statistics of the differences between the modelled measured STEC, as well as graphs visualizing these statistics. A hexbin graph (2D histogram) was used to assess model-measurement agreement, where x-axis represents the measured STEC and y-axis represents the modelled STEC, and the color represents the number of observations in the bin. A reference line y=x and a linear regression $y = a \cdot x + b$ (reporting a, b, and R^2) were plotted on each panel. Additionally, the following metrics were calculated: Bias, MAE, STD, RMS, Corr, R^2 for each model. The colored bar describes the number of points in the bin.

The second graph is an error histogram. The graph shows the bias value as a solid line and zero as a dotted line. On the X-axis, we have the error value—the difference between the measurement and the model—and on the Y-axis, the number of measurements.

The statistics table can be displayed using the print() function.

```
stats_table = cmp.report(
    title="Modelled vs Measured STEC",
    show_hex=True,
    show_hist=True,
    return_stats=True
)
```



A cursory analysis of the figures leads to three conclusions:

1) The GIM model is the best fit for the measurements out of all three models. It cannot be overlooked that in the measured STEC we use the DCB of satellites and receivers estimated in the GIM generation process

2) The Klobuchar model twice the standard deviation and a tendency to slightly overestimate STEC values.

3) The NTCM model has a slightly higher STD than Klobuchar, and is clearly biased towards underestimating STEC values.

Keep in mind that we are still talking about a single day and a single station, so we cannot draw any far-reaching conclusions about the quality of the models based on this analysis. This example is intended to demonstrate the capabilities of the Ionosphere module in terms of STEC analysis for a single station.

```
stats_table.set_index('Model',inplace=True)
df_head(stats_table,ncols=10,floatfmt = ".2f")
```

Model	N	Bias	STD	RMS	MAE	Corr	R2	Slope	Intercept
GIM	19367.00	0.18	4.39	4.39	3.60	0.97	0.93	0.98	0.53
Klobuchar	19367.00	2.58	7.53	7.96	6.13	0.90	0.80	0.90	5.02
NTCM G	19367.00	-3.58	7.98	8.75	7.14	0.91	0.83	0.61	5.96

This concludes our introductory tutorial. You have learned about the TECConfig and TECSession classes and run calculations. You have visualised the results using the default built-in tools. I encourage you to use this knowledge to carry out your own small project. You can select several stations with known DCBs from the IGS website (select their names from the IONEX file header). You can try to select those at different latitudes. Download the observation data, SP3 and broadcast files, and perform the same analysis in a loop for several stations. Compare the results using the tools listed here. Add a few of your own ideas and you have a presentation ready for the seminar :)

In the next notebook, I will discuss the topic of GNSS receiver calibration and the capabilities of GNX-py in this area.

Reciever calibration

In this tutorial, I will briefly discuss the issue of calibrating GNSS receivers for TEC measurement. Several interesting scientific articles have been devoted to this topic, which I recommend reading for a full understanding of the subject (Sardón et al. 1994; Arikan et al. 2008). Let's start by writing down the observation equations for code observations at two frequencies.

$$P_1 = \rho + c(\delta t_r - \delta t_s) + T + I_1 + b^P_{r,1} + b^P_{s,1} + \varepsilon_{P_1}[m]$$

$$P_2 = \rho + c(\delta t_r - \delta t_s) + T + I_2 + b^P_{r,2} + b^P_{s,2} + \varepsilon_{P_2}[m]$$

$$P_{\mathrm{GF}} = P_1 - P_2 = (I_1 - I_2) + (b^P_{r,1} - b^P_{r,2}) + (b^P_{s,1} - b^P_{s,2}) + (\varepsilon_{P_1} - \varepsilon_{P_2})[m]$$

where:

- P_1 - code observation on frequency 1
- P_2 - code observation on frequency 2
- P_{GF} - geometry-free combination of P_1 and P_2
- I_n - ionospheric delay in meters of n frequency
- \$b^{P}*{r,n}*, b^{P}{s,n} \$ - reciever and satellite hardware delay for signals at frequency n

During global ionosphere maps generation process, receiver and satellite biases are elements of the state vector. Therefore, as I mentioned in the previous chapter, we have access to satellite and receiver biases in the IONEX file header. These are the biases of the satellites and receivers that were obtained during map generation process. On various data servers, you will also find SINEX files with the .BSX or .BIA extension, which contain DSB or OSB bias values for satellites and some recievers (Schaer 2018). What interests us is the bias for geometry-free observations. It is necessary to have it, because without it, we obtain a STEC value that is subject to systematic error. The process of determining the receiver bias is called receiver calibration (not to be confused with PCO/PCV determination, which is called the same). In GNX-py, the

receiver bias is determined in two ways, both of which depend directly on the ionosphere model used:

1) Calibration using the weighted average method

2) Calibration using PPP-UDUC

Both methods can be summarised as follows: it is the levelling of the measured STEC to the modelled STEC. In this sense, the process is somewhat similar to the phase to code levelling mentioned in the previous chapter. In the weighted average method, after correcting the levelled GF observations (in metres) for satellite bias converted from nanoseconds to metres, we differentiate between the modelled and calculated delay for satellites observed above a certain elevation angle (60 degrees by default). These differences are then averaged over time and weighted by elevation angle, thus obtaining a single estimate for a given timestep. Next, we apply the median function to the set of these values to obtain the final DCB estimate for the receiver. In the PPP-UDUC method, hardware delays of undifferentiated observations are part of the state vector. In fact, in this method, we directly estimate DCB, because GNX-py uses the method of 'freezing' the OSB bias of observations at one of the frequencies (L1 and E1 for GPS and Galileo) by default. In this way, we obtain DCB estimates for each of the epochs processed in the filter. In the software folder, you will find the *ppp2dsb.py* script, which processes the PPP processing output files and generates a CSV file with DCB values for stations for which results are located in the specified directory. You can pass DCBs contained in it to the configuration class, and then it will be used in the STEC measurement. I mentioned earlier that both methods depend on the ionosphere model used, and I would like to return to this point here. You have probably already noticed that the first method is actually a brute-force fit of the measurements to the modelled value. Thus, the quality of the background model value directly affects the DCB value. I recommend using GIM models rather than NTCM or Klobuchar for calibration, as this allows us to fit to the best quality model. This, of course, makes our measurements extremely dependent on the availability of GIM, but at the current stage of software development, it is the best intermediate step we can take. At first glance, it may seem that PPP-UDUC is less dependent on the model used as ionospheric constraints. After all, the influence of a priori ionospheric observations is gradually reduced in the filter. However, as we will show below, the differences in the determined DCBs are quite significant.

Let us now move on to specific examples. We will build an instance of the configuration class that is almost identical to the one in the Introduction, setting only the *rcv_dcb_source* parameter to *calibrate*. This way, the TECSession class will run the calibration procedure. We choose GIM as the background model, as we want to compare the calibrated DCB with the one obtained in GIM. The BOR1 station was chosen because it belongs to the stations used to generate GIM.

```
1 import gps_lib as gnx
```

```
import pandas as pd
import numpy as np
import warnings
from datetime import datetime
warnings.simplefilter(action='ignore', category=FutureWarning)
np.set_printoptions(threshold=np.inf, linewidth=200, suppress=True,
    precision=3)
```

```
OBS_PATH='../data/BOR100POL_R_20240350000_01D_30S_MO.rnx'
NAME = OBS_PATH.split('/')[-1][:4]


NAV = '../data/BRDC00IGS_R_20240350000_01D_MN.rnx'
SP3_LIST=[
'../data/COD0OPSFIN_20240340000_01D_05M_ORB.SP3',
'../data/COD0OPSFIN_20240350000_01D_05M_ORB.SP3',
'../data/COD0OPSFIN_20240360000_01D_05M_ORB.SP3']
ATX_PATH='../data/ngs20.atx'
DCB_PATH= '../data/COD0OPSFIN_20240350000_01D_01D_OSB.BIA'
GIM='../data/COD0OPSFIN_20240350000_01D_01H_GIM.INX'
TLIM = [datetime(2024,2,4,0,0,0),
        datetime(2024,2,4,23,59,30)]

SYS= 'G'

DOY=35
C=299792458
```

```
config = gnx.TECConfig(obs_path=OBS_PATH,
                                        nav_path=NAV,
                                        sp3_path=SP3_LIST,
                                        gim_path=GIM,
                                        sys="G",
                                        gps_freq='L1L2',
                                        gal_freq='E1E5a',
                                        windup=False,
                                        rel_path=False,
                                        sat_pco=True,
                                        rec_pco=True,
                                        atx_path=ATX_PATH,
                                        interpolation_method='lagrange',
                                        ev_mask=30,

                                        add_dcb=True,
                                        add_sta_dcb=True,

                                        rcv_dcb_source='calibrate',

                                        screen=True,
```

```
                                        # skip_sat=['G20','G02'],

                                        station_name=NAME,
                                        day_of_year=DOY,

                                        ionosphere_model='gim',
                                        compare_models=False,
                                        troposphere_model=False,
                                        use_gfz=True,
                                        leveling_ws=50,
                                        median_leveling_ws=50,
                                        min_arc_len=2
                                        )
```

```
controller = gnx.TECSession(config=config)
obs_tec = controller.run()
```

When reading the IONEX file, software also loads the DCB by default. If the station has a DCB defined in the IONEX file header, it will be placed in the dataframe in the *sta_bias* column. This gives us access to the calibrated and reference DCB. The calibrated DCB was, of course, used in the STEC calculations.

```
ref_bias = obs_tec['sta_bias'].iloc[0]
calibrated = obs_tec['sta_dcb'].iloc[0]
```

```
bias_error = (ref_bias-calibrated)
print(f"Calibration error for station {NAME}: {bias_error:.3f} [ns]  (
    Weighted average method)")
```

```
Calibration error for station BOR1: 0.281 [ns]  (Weighted average method)
```

We obtained an error of 0.2 nanoseconds. We can safely assume that an error below 1 nanosecond is expected and acceptable for such a simple calibration method (X,Y,Z). Now let's compare the DCB obtained using the PPP-UDUC method. To do this, let's load the data we processed while working with the PPP guide.

```
sol = pd.read_csv('../output/unc_sol.csv')
```

As I mentioned in PPP-UDUC, we freeze the OSB bias for the reference frequency, thereby forcing the OSB estimation for the second frequency in a differentiated form - DSB, which is exactly what we are interested in. As a result of the equations in the PPP method, we obtain this estimate with the opposite sign. Let us now compare the DCB value from the last filtration epoch and the average from all epochs after the 120th. The value 120, corresponding to 2 hours assuming a 30-second sampling rate, is chosen because even for a 'problematic'

station, after 2 hours we are almost certain to achieve filter stabilization. The reason why it shows these two values is that the ppp2dsb script returns precisely these two values.

```
ppp_dcb = -sol['dcb_gps_c2']/C/1e-09
```

```
last_ep = ppp_dcb.iloc[-1]
```

```
mn = ppp_dcb.iloc[120:].mean()
```

```
d1 = ref_bias-last_ep
d2 = ref_bias-mn
```

```
print(f"Calibration error for station {NAME}: {d1:.3f},{d2:.3f} [ns]  (
    PPP-UDUC)")
```

```
Calibration error for station BOR1: -0.436,-0.117 [ns]  (PPP-UDUC)
```

We see that last epoch DCB has a greater error than the one obtained in moving average approach, but mean DCB is around 10 ns closer to the reference. Let's now make a small comparison and perform STEC calculations for our DCBs with PPP and for those from the GIM file. The application of your own defined bias is done by configuring the appropriate fields in the control class:

```
config.rcv_dcb_source ='defined'
config.define_station_dcb = mn
controller = gnx.TECSession(config=config)
obs_tec_ppp = controller.run()
```

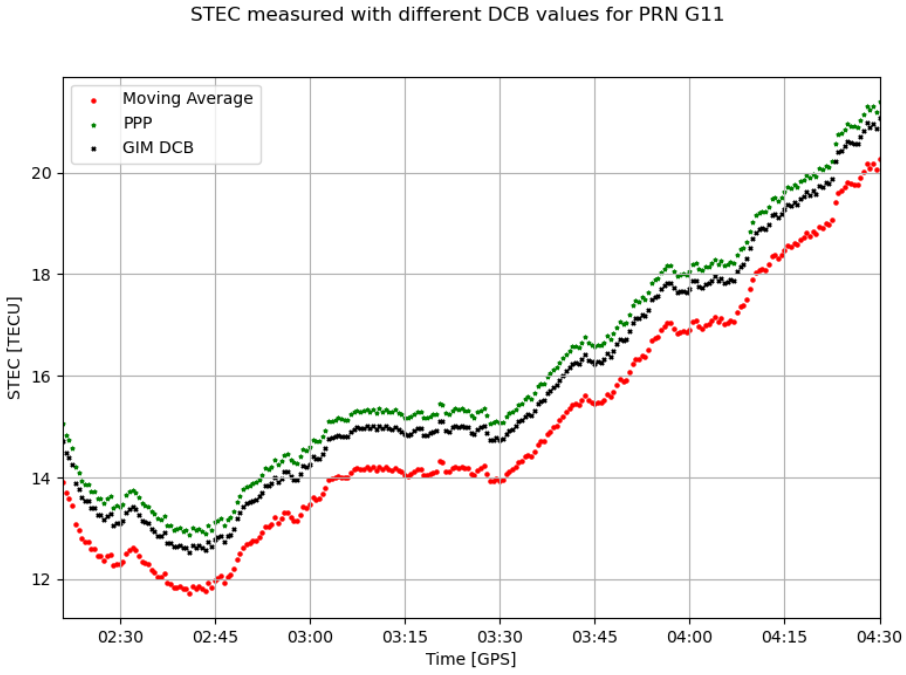
```
config.rcv_dcb_source ='gim'
controller = gnx.TECSession(config=config)
obs_tec_gim = controller.run()
```

We convert the obtained STEC into TECU units. Let us compare the observations obtained in the graph.

```
stec_calibrated = obs_tec[['leveled_tec']]/1e16
stec_ppp = obs_tec_ppp[['leveled_tec']]/1e16
stec_gim = obs_tec_gim[['leveled_tec']]/1e16
```

Let's write some simple code to visualise STEC on a shared graph. We will mark the STEC obtained from three sources on it so that we can see the DCB effect with our own eyes.

```
import matplotlib.pyplot as plt
from matplotlib.dates import DateFormatter

ind = 'G11_0_1'
fig, ax = plt.subplots(figsize=(9,6))
fig.suptitle(f"STEC measured with different DCB values for PRN {ind[:3]}")
gr = stec_calibrated.loc[ind]
ppp=stec_ppp.loc[ind]
gim = stec_gim.loc[ind]
gr=gr.reset_index()
gim=gim.reset_index()
ppp=ppp.reset_index()
ax.scatter(gr['time'],gr['leveled_tec'],label='Moving Average',s=5,c='r',marker='o')
ax.scatter(ppp['time'],ppp['leveled_tec'],label='PPP',s=5,c='g',marker='*')
ax.scatter(gim['time'],gim['leveled_tec'],label='GIM DCB',s=5,c='black',marker='x')
ax.set_xlim(gr['time'].min(),gr['time'].max())
ax.set_ylabel('STEC [TECU]')
ax.set_xlabel('Time [GPS]')
ax.grid()
ax.legend()
ax.xaxis.set_major_formatter(DateFormatter('%H:%M'))
```




```
print(f"Calibration error between GIM and PPP: {(stec_gim-stec_ppp)['
    leveled_tec'].round(3).unique()[0]} [TECU]")
print(f"Calibration error between GIM and Moving Average: {(stec_gim-
    stec_calibrated)['leveled_tec'].round(3).unique()[0]} [TECU]")
```

```
Calibration error between GIM and PPP: -0.334 [TECU]
Calibration error between GIM and Moving Average: 0.803 [TECU]
```

As you can see, the differences in DCB are quite significant. The receiver bias derived from the PPP solution turns out to be much closer to the reference than the one calibrated using the moving average method. In PPP-UDUC, we estimate it together with the ionospheric delay and the receiver clock, so it is a more advanced method than simple levelling of observations. For STEC measurements, which we discussed in the previous tutorial, the safest calibration method will be to first perform PPP with ionospheric constraints from the GIM model and then use the estimated DCB for STEC measurements.

Of course, both of these methods make us completely dependent on the availability of the model. In the future, the Ionosphere module will be developed with ionosphere modelling techniques that take into account the estimation of receiver DCBs. The PPP-UDUC technique presented here allows receivers to be calibrated with sufficient accuracy to perform STEC measurements and ensure that the DCB error does not exceed 1-2 TECU. In the next notebooks, we will present the programme's functionality that enables the calculation of ionosphere

acitvity indexes. For some of them, proper DCB estimation plays a crucial role.

Ionospheric activity indices

One of the ways in which we can use GNSS TEC measurements is to calculate ionospheric activity indices. GNX-py implements the calculation and visualisation of SIDX (Sudden Ionosphere Disturbance Index), GIX (Gradient Ionosphere Index), and ROT/ROTI (Rate of TEC, Rate of TEC Index) indices. These indices are described in detail in the following works: (Jakowski and Hoque 2019; Cherniak et al. 2018; Carmo et al. 2021). In this tutorial, we will discuss how to calculate them and show how to monitor ionospheric activity using GNX-py.

First, let us begin by discussing and deriving equations relating to SIDX, i.e. the index describing temporal changes in TEC in a given area.

Let us denote STEC, total electron content along the signal transmission path measured at time t, as:

$$\mathrm{STEC}(t) = \frac{\tilde{L}_4^{(n)}(t)}{40.3\left(\frac{1}{f_1^2} - \frac{1}{f_2^2}\right)} \text{ [TECU]}$$

where $\tilde{L}_4^{(n)}(t)$ is the code-leveled phase observation geometry free and f_1, f_2 are recieved signals frequencies in Hz. Conversion from STEC to VTEC is performed using the mapping function $M(e;H)$, where e and H are, respectively: the satellite elevation angle and the height of the ionospheric shell.

$$\mathrm{VTEC}(t) = \frac{\mathrm{STEC}(t)}{M(e;H)} \quad \text{[TECU]}$$

$$M(\varepsilon;H) = \frac{1}{\sqrt{1 - \left(\frac{R_E \cos e}{R_E + H}\right)^2}}$$

The SIDX index describes the rate of change of TEC in a given area. Imagine that we have a network of GNSS stations and we record observations at 30-second intervals. Let us assume that we are tracking a certain satellite and recording the measured STEC at epochs t_0 and t_1 with one of the receivers.

```
from helpers import plot_stec_change
from tools import df_head, df_tail
import matplotlib.dates as mdates
import warnings

warnings.filterwarnings(
    "ignore",
```

```
    message="invalid value encountered in create_collection",
    category=RuntimeWarning,
    module="shapely.creation"
)
warnings.filterwarnings(
    "ignore",
    message="The PostScript backend does not support transparency;
    partially transparent artists will be rendered opaque"
)
```

```
plot_stec_change()
```

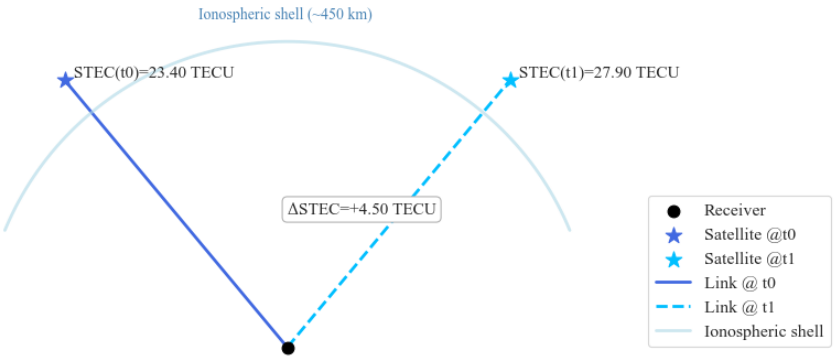


Figure 1. Visualization of STEC change between t0 and t1.

By converting to VTEC and differentiating the observation from epoch t_1 and t_0 and dividing by the time interval, we obtain an approximate value for the rate of change of VTEC. Of course, satellites in neighbouring epochs are located in different places in space, and thus the location of the IPP changes, but we assume that in short intervals (at short distances between IPPs) the change in VTEC is minimal. The graph above illustrates this type of situation at the level of a single link. With the aforementioned network of stations and observing many satellites in different locations in neighbouring epochs, we obtain a set of such VTEC differences. Let us write this down step by step:

SIDX (temporal VTEC change)

At the level of a single link:

$$\mathrm{SID}_{\mathrm{link}}(t_0, t_1) = \frac{\Delta\mathrm{STEC}_{t_0,t_1}}{M(e;H)\,\Delta t} = \frac{\Delta\mathrm{VTEC}_{t_0,t_1}}{\Delta t} \quad [\mathrm{TECU/s}]$$

With a set of links between two neighbouring epochs, we can assign a given difference to epoch t_1 and use an aggregation function to represent all recorded VTEC changes in a given area.

$$\mathrm{SIDX}(t_1) = \mathrm{Agg}\left\{\mathrm{SID}_{\mathrm{link},\ell}(t_0, t_1) \;:\; \ell \in \mathcal{L}(t_0, t_1)\right\} \quad [\mathrm{TECU/s}]$$

Where ℓ is a single link and $\mathcal{L}(t_0, t_1)$ is the set of all links between two neighbouring epochs and $\mathrm{Agg} \in \{\mathrm{median}, \mathrm{mean}, \mathrm{p95}, \max\}$ is the aggregation function.

In GNX-py, we use the 95% percentile by default for SIDX visualisation. We also scale the calculated changes to mTECU/s.

$$\mathrm{SIDX}_{\mathrm{min}}(t) = 1000 \cdot \mathrm{SIDX}(t) \quad [\mathrm{mTECU/s}]$$

Finally, all calculated and aggregated differences constitute our SIDX index for a given reference epoch (t_1). Note that I have omitted the receiver and satellite DCB in the equations – they cancel out during the differentiation operation. Since we can assume their constancy over a 24-hour period (X,Y,Z), we can calculate the SIDX index on uncalibrated observations. The same assumption will also apply below when discussing ROTI and GIX, which we will now move on to.

ROT (Rate Of TEC)

SIDX is a certain (mean, median, percentile) measure representing temporal changes in VTEC in a given area; it can be represented as an aggregation of ROT values:

$$\mathrm{ROT}(t) \approx \frac{\Delta\mathrm{STEC}}{\Delta t} = \frac{\mathrm{STEC}(t) - \mathrm{STEC}(t - \Delta t)}{\Delta t} \quad [\mathrm{TECU/s}]$$

$$\mathrm{ROT}_{\mathrm{min}}(t) = 60 \cdot \mathrm{ROT}(t) \quad [\mathrm{TECU/min}]$$

ROTI, on the other hand, is calculated as the standard deviation of ROT in a given time window, e.g. 5 minutes. In GNX-py, we work with VTEC for SIDX, and with STEC for ROT and ROTI, following the convention presented in X and Y. ## ROTI (Rate Of TEC Index)

$$\mathrm{ROTI}(t) = \sqrt{\frac{1}{N-1}\sum_{k=1}^{N}\left(\mathrm{ROT}_{\mathrm{min}}(t_k) - \overline{\mathrm{ROT}_{\mathrm{min}}}\right)^2} \quad [\mathrm{TECU/min}]$$

gdzie: - t_k — epochs in given time window, - $\overline{\mathrm{ROT}_{\mathrm{min}}}$ — mean ROT in given time window.

As with SIDX, in the case of ROT we also work with receiver-satellite pairs. In GNX-py, ROTI is presented in the form of a coordinate grid in a geographic or solar-geomagnetic system. The ROTI values calculated in the window can be assigned to grid points according to the following approaches: - the ROT value is assigned to the 'centre' position, i.e. halfway along the satellite tracking path on the ionospheric shell - at the initial position in the window (in the first IPP for a given satellite-receiver link)

GIX (VTEC spatial gradient)

The last of the indices available in GNX-py is the GIX family of indices: GIX, GIXS, and GIX95. These are indices describing the spatial variation of VTEC in a given area. Unlike ROT and SIDX, in case of GIX we process data from a single epoch. Again, let us assume that we have a network of stations recording GNSS satellites signals. In each epoch, we observe a set of satellites from each station and obtain the corresponding STEC measurements along with their IPP according to the assumed model (by default, thin-shell 450 km). The GIX is calculated as follows:

Having pair of IPPs (i, j) (see figure below):

$$\Delta\mathrm{VTEC} = \mathrm{VTEC}_i - \mathrm{VTEC}_j$$

We can calculate the VTEC gradient in the direction of $ j →i $:

$$g_{ij} = \frac{\Delta\mathrm{VTEC}}{\Delta s} \quad [\mathrm{TECU/km}] \qquad \Delta s \approx s_{ij}$$

where Δs is the distance between IPPs

By calculating the azimuth $\alpha_{j\to i}$ of the line connecting two points, we can decompose the gradient into a gradient in the direction WE/NS:

$$g_{WE} = g_{ij}\sin\alpha_{j\to i}, \qquad g_{NS} = g_{ij}\cos\alpha_{j\to i} \quad [\mathrm{TECU/km}]$$

The calculated gradient is located at the central point $CP_{i,j}$ (see figure) between the IPP points. Then all pairs (i, j) from the set $\mathcal{P}(t)$ from a given epoch are aggregated using the mean function:

$$\overline{g}_{WE}(t) = \mathrm{mean}\{g_{WE}\}_{(i,j)\in\mathcal{P}(t)}, \qquad \overline{g}_{NS}(t) = \mathrm{mean}\{g_{NS}\}_{(i,j)\in\mathcal{P}(t)}$$

Finally, the GIX index for a given epoch is calculated as:

$$\mathrm{GIX}(t) = \sqrt{\overline{g}_{WE}(t)^2 + \overline{g}_{NS}(t)^2} \quad [\mathrm{TECU/km}]$$

$$\mathrm{GIXS}(t) = \mathrm{std}\{|g_{ij}|\}_{(i,j)\in\mathcal{P}(t)} \quad [\mathrm{TECU/km}]$$

$$\mathrm{GIXP95}(t) = \mathrm{percentile}_{95}\{|g_{ij}|\}_{(i,j)\in\mathcal{P}(t)} \quad [\mathrm{TECU/km}]$$

The figure below illustrates an example on a micro scale – for a pair of points. In GNX-py, criterion for selecting pairs is based on the distance between points. The user can set this range themselves or select the default, e.g. a range from 30 to 250 km. The above calculations are therefore performed for all pairs created in a given epoch.

```
from helpers import plot_gix_two_points
plot_gix_two_points(
    draw_gc_arc=True,
    show_cp_arrow=False,
    title="GIX for two IPPs (with CP)"
)
```

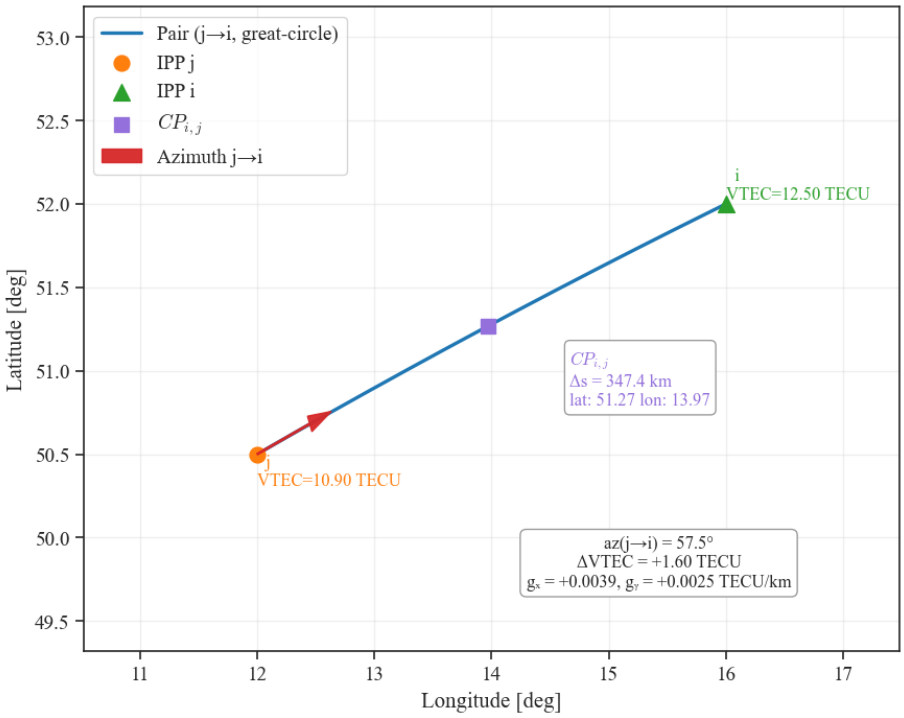


Figure 2. Visualization of TEC gradient between pair of IPPs.

To learn more about the issues related to the indices described above, please refer to works X, Y, and Z. Most of the methodology is based on these works.

Now we will show how the above indices can be obtained in GNX-py. To speed up the tutorial and save space, we will perform calculations on a very limited GNSS station network. Due to the fact that the calculations are based on built-in pandas tools, for large networks (e.g. 100 or more stations) calculations for the entire day at 30-second intervals can take up to several dozen minutes. Therefore, we will focus on a small network and a short period of time so that you can learn how to call functions, save and visualise data, which will help you implement algorithms for your own needs.

Processing example

```
import matplotlib.pyplot as plt
import gps_lib as gnx
import pandas as pd
import numpy as np
```

Let's start by loading the data. To calculate indices representing ionospheric activity over a given region, we must, of course, have access to the network of stations in that region. In this example, in the ./network folder, you'll find several files with processed measurements from EUREF network stations. These data was processed using the *STECSessionclass*, with reciever's DCB's obtained from PPP-UDUC measurements. Carefully studying the equations presented above, you'll notice that the receiver's and satellite's DCB's, as constant biases, cancel out during the time-differentiation operation performed for SIDX and ROTI calculation. We can therefore work with uncalibrated measurements, as we're interested in the change, not the absolute value, of TEC. However for GIX family calculation we must have calibrated STEC measurements. Data loading is performed using the *load_stec_folder* function, whose parameters are described below:

```

data_path = './network'
data, info  = gnx.load_stec_folder(folder=data_path,
                                   file_suffix="STEC.parquet.gzip",
                                   min_elev_deg=30,
                                   station_name_len=6,
                                   unique_id_len=4,
                                   quiet=True,
                                   skip_negative=True,
                                   required_columns = ("lat_ipp", "
    lon_ipp", "leveled_tec", "ev","sv","time"))
```

This function is designed to read sets of output files from individual station processing, saved by default in Parquet format. This format takes up less space than CSV, and it's useful as we usually want to access more than one station. The function arguments shown are default values, except, of course, for the folder path. The arguments define an elevation mask, name filtering

using *network_fiter* to load a subset of data from selected stations, and the *skip_negative* flag to skip files containing negative TEC, which indicates an incorrect DCB or other station/measurement malfunction. $file\_suffix$ parameter is intended for situations when you have various parquet files in a folder, e.g. positioning results. Let's take a look at the obtained data. The info dictionary contains information about the upload process—the number of stations accepted and rejected, etc. You can study the keys below:

```
df_head(data[['leveled_tec','lat_ipp','lon_ipp','name']],nrows=4)
```

sv	time	leveled_tec	lat_ipp	lon_ipp	name
E02_0_1	2024-02-04 08:35:30	48.201	46.225	9.380	BAUT_E
E02_0_1	2024-02-04 08:36:00	48.157	46.261	9.398	BAUT_E
E02_0_1	2024-02-04 08:36:30	47.955	46.297	9.415	BAUT_E
E02_0_1	2024-02-04 08:37:00	47.882	46.333	9.432	BAUT_E

```
info.keys()
```

```
dict_keys(['files_scanned', 'files_accepted', 'files_rejected', '
    stations_unique', 'stations_accepted', 'stations_rejected', '
    rejected_files', 'errors'])
```

As you can see, our data includes the satellite PRN, station ID, STEC measurements (code and phase), and the geocentric IPP coordinates. Let's perform some additional preparations for calculating the indices, specifically, calculate the mapping function for our model (450 km thin-shell) and VTEC.

```
data = data.reset_index()
data['time'] = pd.to_datetime(data['time'])

df= data.copy()
R=6371
ish = 450
m_f = 1 / np.sqrt(1 - (R / (R + ish)) ** 2 * np.cos(np.deg2rad(df['ev']))
     ** 2)
df['M'] = m_f
df['stec'] = df['leveled_tec']
df['vtec'] = df['stec']/df['M']
```

Using the Region class from the *monitoring.py* module, we define the area over which we want to determine our indexes. This area is already implicitly defined by the selected stations, but when processing larger networks, it's worth specifying the study area more precisely, as we may sometimes have access to measurements very distant from the locations of interest, which still require computational time.

```
reg = gnx.Region(lat_min=50,lat_max=57.5,lon_min=10,lon_max=30)
```

We're ready to calculate ROTI. We do this in the *compute_roti_links* function. We define:

- *df*-our dataframe with observations,
- *reg*-region class instance,
- *tec_source*-name of the source observation column (stec or vtec),
- *tec_scale*-scale in which we report stec or vtec (1 for TECU, 1e16 for electrons),
- *window_min*-the window for the standard deviation and the minimum number of observations needed for the deviation (note - once this window is reached, std will be calculated in a smaller window than defined),
- *min_elev_deg*_elevation mask,
- *coord_mode*_reference system (geographic or solar-geomagnetic),
- *detrend_5min*-flag indicating whether to use a 5-minute moving average for detrending stec.

```

roti = gnx.compute_roti_links(
    df=df,
    region=reg,
    tec_source='stec',
    tec_scale = 1,
    window_min=5,
    min_samples=10,
    max_gap_s=31,
    split_on_day_change = False,
    min_elev_deg=30,
    detrend_5min=True,
    coord_mode='GEO'

)
```

```
df_tail(roti[['time','lat_ipp','lon_ipp','ROT_tecu_per_min','
    ROTI_tecu_per_min']],nrows=3,ncols=10,reset_index=False,index=False,
    truncate_str=3)
```

time	lat_ipp	lon_ipp	ROT_tecu_per_min	ROTI_tecu_per_min
2024-02-04 23:59:30+00:00	51.828	14.679	0.058	0.084
2024-02-04 23:59:30+00:00	50.185	21.901	-0.028	0.050
2024-02-04 23:59:30+00:00	50.265	18.852	0.065	0.074

Let's look at the results. Each row represents a satellite-receiver pair at a given epoch. Let's choose a sample station and look at the first 10 minutes of observations. As you can see, ROTI is calculated within a 5-minute window, as we chose. The first value is obtained after 5 minutes and is assigned to the epoch 00:05:00, and comes from calculating the standard deviation over the last 5 minutes of calculated ROTI. Subsequent ROTI observations are calculated in the same way – each epoch contains $std(ROT)$ from the last $window\_min$ minutes.

```
roti_sample = roti[(roti['name'] == 'BOGE_E') & (roti['sv'].str.
    startswith('E10'))].sort_values(by='time')[['time','name','sv','
    ROTI_tecu_per_min']]
df_head(roti_sample,nrows=12,ncols=10)
```

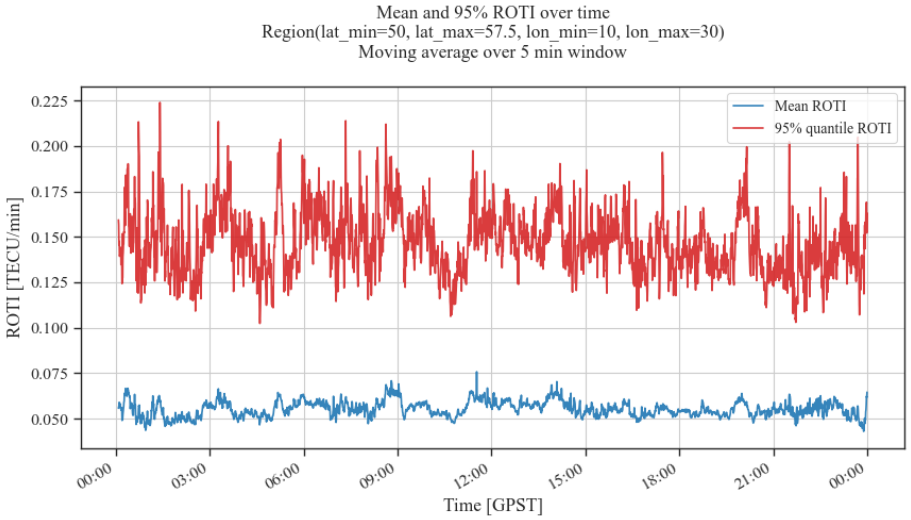
index	time	name	sv	ROTI_tecu_per_min
4.000	2024-02-04 00:00:00+00:00	BOGE_E	E10_0_1	nan
171.000	2024-02-04 00:00:30+00:00	BOGE_E	E10_0_1	nan
344.000	2024-02-04 00:01:00+00:00	BOGE_E	E10_0_1	nan
519.000	2024-02-04 00:01:30+00:00	BOGE_E	E10_0_1	nan
698.000	2024-02-04 00:02:00+00:00	BOGE_E	E10_0_1	nan
880.000	2024-02-04 00:02:30+00:00	BOGE_E	E10_0_1	nan
1064.000	2024-02-04 00:03:00+00:00	BOGE_E	E10_0_1	nan
1244.000	2024-02-04 00:03:30+00:00	BOGE_E	E10_0_1	nan
1425.000	2024-02-04 00:04:00+00:00	BOGE_E	E10_0_1	nan
1608.000	2024-02-04 00:04:30+00:00	BOGE_E	E10_0_1	nan
1787.000	2024-02-04 00:05:00+00:00	BOGE_E	E10_0_1	0.003
1968.000	2024-02-04 00:05:30+00:00	BOGE_E	E10_0_1	0.005

Using pandas tools, we can create a first simple visualization of our data - let's look at the mean and 95th percentile of ROTI in each epoch.

```

roti['mn']  = roti.groupby('time')['ROTI_tecu_per_min'].transform('mean')
roti['q95'] = roti.groupby('time')['ROTI_tecu_per_min'].transform(lambda
    s: s.quantile(0.95))
```

```
to_plot = (
    roti
    .set_index('time')[['mn', 'q95']]
    .sort_index()
    .rename(columns={
        'mn':  'Mean',
        'q95': 'ROTI_95'
    })
)
ax = to_plot.plot(
    figsize=(10, 5),
    ylabel='ROTI [TECU/min]',
    xlabel='Time [GPST]',
    title=(
        f'Mean and 95% ROTI over time\n{reg}\n'
        'Moving average over 5 min window\n'
    ),
    color=['tab:blue', 'tab:red'],
    lw=1.2,
    alpha=0.9,
    grid=True,
)
ax.xaxis.set_major_formatter(mdates.DateFormatter('%H:%M'))
ax.legend(
    [r'Mean ROTI', r'95% quantile ROTI'],
    loc='upper right',
    fontsize=10,
    frameon=True
)
```

```
<matplotlib.legend.Legend at 0x17a0a4890>
```



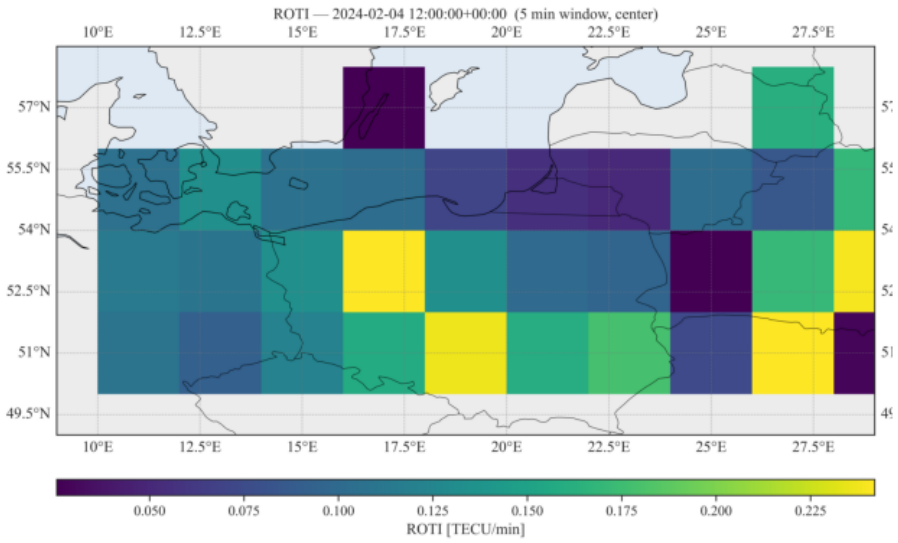
As you can see, the graph allows you to observe the evolution of the ROTI over time. Day 2024 035 is a quiet day for geomagnetic activity (Ap ~6), so it's not surprising that we observe a relatively stable graph without any anomalies. You can also plot the graph in mTECU/s units if you want to observe the index at a different time resolution. A key feature of the module in case of ROTI is the ability to visualize it in the form of snapshot images covering a given time window. This is achieved using the *make_roti_snapshots* function. We insert our resulting dataframe into the function and define several plotting parameters. Let's now discuss the most important ones: - *roti_df* is a dataframe with roti results - *every_n* - is the interval at which we save maps in epochs (e.g., with a 30-s sampling rate, 10 epochs = a 5-minute window) - *mode* ('center', 'trailing', or 'top') is the mechanism for assigning snapshots to epochs, at the middle, beginning, or end of the window, respectively. As mentioned, ROTI is assigned to the end of the window by default, so 'top' is most appropriate. - *global_scale* - whether to globally calculate the ROTI value range (used to build the legend) - *q_scale* - together with the global scale used to build the legend range, indicates the lower and upper percentiles of ROTI values, which constitute the boundaries of the values in the chart legend - *out_dir* - image output path - $vmin, vmax$ - ROTI min and max range, overrides the algorithm for automatically determining the scale on the chart - *agg* - aggregation function, defaulting to 'mean'. For each point in our ROTI grid, we aggregate the observed values using the aggregate function. When tracking a given satellite and having several ROTI observations, we average them and assign them to a given time (mode) and grid location. - *res_deg* - grid resolution - *min_points* - minimum number of points per square (res_deg x res_deg) needed to assign a value to the grid - *coord_mode* - coordinate system in which we plot the grid (GEO or SM)

```
roti_copy = roti[(roti['time'].dt.hour>=10) & (roti['time'].dt.hour<=13)
```

```
    ].copy()
extent = [9, 29, 49,58.5]
gnx.make_roti_snapshots(
    roti_df=roti_copy,
    every_n=30,                    # every 30 epochs  => 30*0.5min = 15
    min
    window_min=5,
    mode='center',
    res_deg=2,
    min_points=3,
    global_scale=False,
    q_scale=(5,99),
    out_dir="./figures/roti",
    # vmin=0,
    # vmax=0.3,
    agg='Q95',
    coord_mode = 'GEO',
    extent=extent,
)
```

You can view the created images in the .figures/roti directory, below you will see randomly selected figure. The images are generated as follows: each square with dimensions corresponding to the resolution *res_deg* is assigned a value aggregated using the *agg* function. If we have *min_points* points for which we calculated ROTI in a given area covering a tile, we aggregate the values and assign them to the tile. Together with the linear graph you saw above, this graph is useful in spatial visualisation.

```
import matplotlib.image as mpimg
path ='./figures/roti/roti_20240204_120000.png'
img = mpimg.imread(path)
plt.figure(figsize=(8, 8))
plt.imshow(img)
plt.axis('off')
plt.show()
```



As explained in the introduction, SIDX is also an indicator of TEC temporal variability. With data from multiple stations available, we calculate it in GNX-py in the same way as ROTI. The arguments of the function calculating SIDX are the same as in the function calculating ROTI links.

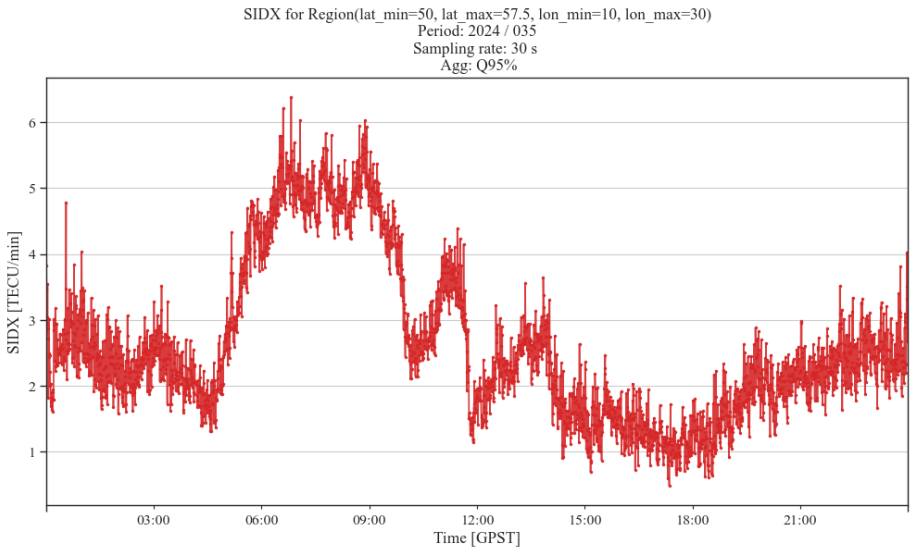
```

sidx = gnx.compute_sid(df=df,
    region=reg,
    min_elev_deg=30,
    iono_h_m=450,
    stec_scale=1,
    max_gap_s=30,
    split_on_day_change= True,
    agg= "p95")
```

```
df_head(sidx,nrows=3,ncols=2)
```

time	SIDX__tecu__per__min
2024-02-04 00:00:30+00:00	3.821
2024-02-04 00:01:00+00:00	3.474
2024-02-04 00:01:30+00:00	3.040

The result is a pd.Series with SIDX calculated for each epoch. Visualisation can be provided by built-in pandas tools.

```
ax=sidx.plot(
    x='time',
    y='SIDX_tecu_per_min',
    figsize=(12, 6),
    color='tab:red',
    alpha=0.9,
    lw=1.2,
    marker='.',
    markersize=3,
    ylabel='SIDX [TECU/min]',
    xlabel='Time [GPST]',
    title=(
        f'SIDX for {reg}\n'
        'Period: 2024 / 035\n'
        'Sampling rate: 30 s \n Agg: Q95%'
    ),
    grid=True
)
```



The GIX index, like the previous ones, is calculated using a dedicated func-

tion. The function arguments are almost identical to those for ROTI and SIDX, with a few minor exceptions. We will now discuss the arguments unique to GIX: - *dipole_preset* - dipole length range (distance between IPP pairs). The choice is between three defined ranges: 30-250, 50-500, 100-1000 km. - *dmin_km*, *max_km* - as an alternative to the dipole length, you can select your own lower and upper distance limits for which IPP pairs are formed. Overwrites dipole_preset - *max_delta_M* - maximum difference in the mapping function between two IPPs - the range 0.0-0.2 is safe in terms of minimising the impact of an incorrectly estimated receiver DCB. Unlike ROTI and SIDX, we have to operate on calibrated measurements here, because in the measurements we differentiate between observations from different satellite-receiver pairs - *stratified_sampling* - point thinning, together with *max_pairs_per_epoch*, is a condition for the maximum number of points per epoch, which can be useful when there are a large number of measurements.

```

df_copy = df[(df['time'].dt.hour>=12) & (df['time'].dt.hour<=13)].copy()
```

```
dmin, dmax = 150, 500

times, pairs = gnx.compute_gix(
    df=df_copy,
    region=reg,
    min_elev_deg=30,
    # dipole_preset = "30-250",
    dmin_km = dmin,
    dmax_km =dmax,
    iono_h_m=450,
    stec_scale = 1.0,
    max_delta_M= 0.1,
    max_pairs_per_epoch = 200e3,
    stratified_sampling =True
)
```

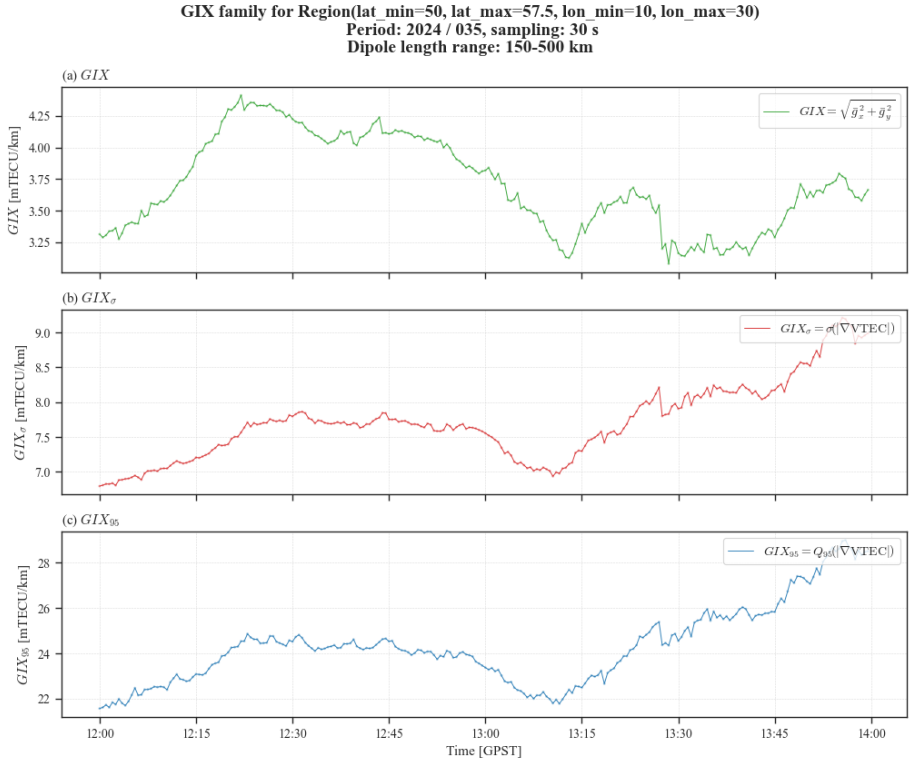
Our output consists of two dataframes. The *times* dataframe contains a family of GIX indices for each epoch. It includes: - GIX - average GIX in the E-W and N-S directions (subscripts x and y, respectively) - GIXS with directional components - GIXP95 with directional components, as well as negative and positive components, i.e. the 5th and 95th percentiles for positive and negative gradients, respectively, in the E-W (x) and N-E (y) - npairs - number of pairs in a given epoch that were used to calculate the indices

In addition to the built-in pandas tools, you can visualise the indices using a function from the GNX-py library.

```
df_head(times[['time','GIX_mtecu_km','GIXS_mtecu_km']],nrows=3,ncols=7)
```

index	time	GIX_mtecu_km	GIXS_mtecu_km
0.000	2024-02-04 12:00:00+00:00	3.314	6.794
1.000	2024-02-04 12:00:30+00:00	3.288	6.807
2.000	2024-02-04 12:01:00+00:00	3.306	6.825

```
fig, axes = gnx.plot_gix_family(times, reg=reg, dmin=dmin, dmax=dmax)
plt.show()
```



In the *pairs* dataframe, you will find data on each pair that was created in each epoch. Each row contains the dipole centre coordinates, CP_lat and CP_lon, the absolute VTEC gradient calculated in the dipole direction, as shown in the figure in the introductory section, directional gradients, dipole length, mapping function values, station and satellite names, and VTEC differences between IPPs.

This data can be used for spatial visualisation of GIX. The *plot_gix* function from the GNX-py library is used for this purpose. Keep in mind that we are operating on dipoles of different lengths (you can modify this range with *dmin_km* and *dmax_km*), so when interpreting the image, do not be confused by the colours of individual points. What you are looking at are GIX values calculated

for pairs of points separated by *dmin_km* and *dmax_km*. I encourage you to change the dipole length range and examine the plots and *pairs* dataframe after that. This will give you an intuitive insight into how GIX is calculated.

```
df_head(pairs[['time','CP_lat','CP_lon','grad_abs_mtecu_km']],nrows=3,
    ncols=7)
```

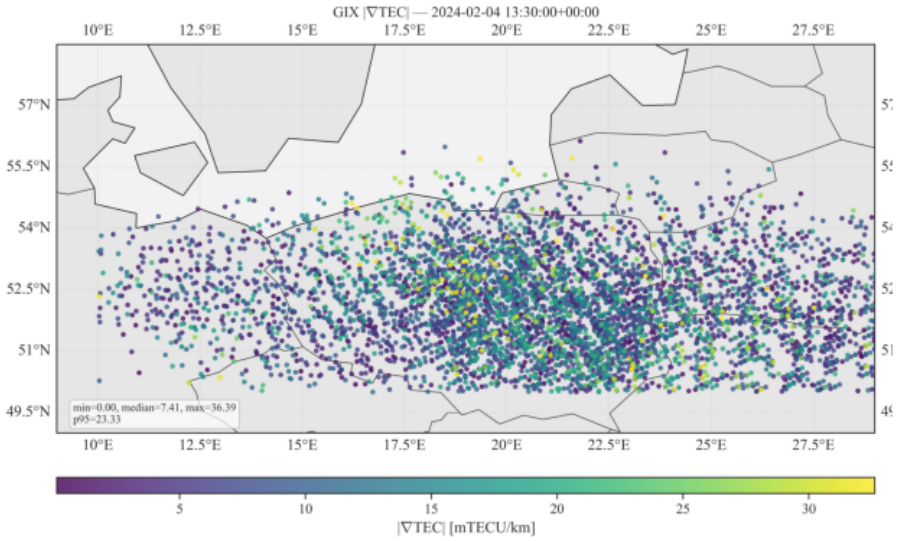
index	time	CP_lat	CP_lon	grad_abs_mtecu_km
0.000	2024-02-04 12:00:00+00:00	50.706	16.104	0.197
1.000	2024-02-04 12:00:00+00:00	52.207	18.852	6.390
2.000	2024-02-04 12:00:00+00:00	52.184	18.324	5.389

The displayed value is the column *grad_abs_mtecu_km*, i.e. the absolute value of the gradient per kilometre in mTECU units. The images below can be interpreted as follows. When we see a large accumulation of bright spots in a given area, we can assume that there is a high VTEC gradient in that region, within a radius of $dmin\_km, dmax\_km$ from the spot, and similarly, in the case of dark spots, a low TEC gradient.

```
epochs = pairs['time'].unique()
epoch = epochs[180]
fig, ax, sc = gnx.plot_gix(pairs, epoch,save_path='./figures/gix/
    gix_epochs.png',extent=extent)
```

As you can see, all our data is created in the pandas dataframe or series format. You can save data in any format supported by pandas. I recommend CSV or parquet.gzip, especially the latter in terms of space optimisation.

```
import matplotlib.image as mpimg
path ='./figures/gix/gix_epochs.png'
img = mpimg.imread(path)
plt.figure(figsize=(8, 8))
plt.imshow(img)
plt.axis('off')
plt.show()
```




```
out='./output'
sidx.to_frame().to_parquet(f'{out}/sidx.parquet.gzip')
times.to_parquet(f'{out}/times.parquet.gzip')
pairs.to_parquet(f'{out}/pairs.parquet.gzip')
roti.to_parquet(f'{out}/roti.parquet.gzip')
```

In the next notebook, we will discuss the use of kriging to create VTEC models based on measurements from a network of GNSS stations.

Ionosphere modelling with kriging

In this guide, we will show you how to use the Kriging modules built into GNX-py to model the ionosphere. Kriging is a geostatistical method of spatial interpolation, thanks to which we can estimate the value of a given quantity in places not covered by measurements, based on a set of measured values in the vicinity of the point we are interested in. Kriging owes its effectiveness

and popularity to the fact that it takes into account the spatial correlations of the measured samples, distance and direction between them. I will not derive equations for kriging here, as this goes beyond the scope of this tutorial to some extent. I encourage you to read publications: (Sparks et al. 2011; Wielgosz et al. 2003) and materials on the internet, e.g.: (Esri, n.d.; Maxwell 2018; Maurer 2022). I based my implementation mainly on these materials.

What do we actually gain from kriging in the context of ionosphere modelling? Let's start with the spatial resolution of the model. Empirical models such as NeQuick, Klobuchar or NTCM can, in theory, be densified at will, and there are no restrictions on generating a VTEC grid from such models. However, as you probably know, these models, although effective in navigation, are not very accurate. Assimilation models such as GIM are strongly based on measurements,however, the assumptions of the model, such as global and 24-hour coverage, mean that, however precise they may be, they have limitations when it comes to modelling local, short-term phenomena in the ionosphere.

Kriging offers two major advantages: we can obtain a modelled state of the ionosphere in a given area with a spatial resolution significantly higher than that of the GIM model and, at the same time, with a precision superior to empirical models, because we operate on real TEC measurements in a given epoch. The downside is that, unlike the techniques used in GIM, we do not have a tool for propagating our model over time, i.e. we use measurements from a given epoch to obtain a VTEC model/map for the same epoch. In addition, we have to operate with known DCBs of satellites and receivers, so we need external data sources or calibration. As I mentioned in the notebook dedicated to calibration, the optimal solution is to use the final GIM product for calibration, which is available with some delay. Spatial resolution is theoretically independent of the number of measurements, but in practice, the number of available measurements is important if we want to map conditions as close to reality as possible.

In previous guides, we discussed how to perform STEC measurements and calibration, and calculate ionospheric activity indices in GNX-py. In this notebook, we will use the data we used before to calculate the indices. The satellite DCB in this data comes from GIM, while the receiver DCB comes from the PPP-UDUC solution with ionospheric constraints from the GIM model. As we showed in the *calibration.py* notebook, the receiver DCB obtained in this way should not have an error greater than 1 TECU. However, we ignore data where a negative STEC occurred somewhere after calibration, because this tells us that the calibration error is too large. In GNX-py, we have access to Ordinary Kriging, Universal Kriging, and Kriging based on the methodology described in (Sparks et al. 2011) In this guide, we will perform kriging on a given area and then compare the VTEC value between our model and GIM and NTCM models.

```
1 import gps_lib as gnx
2 import numpy as np
3 import pandas as pd
4 import vtec_viz as vv
```

```
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings(
    "ignore",
    message="invalid value encountered in create_collection",
    category=RuntimeWarning,
    module="shapely.creation"
)
```

First, let us define a few parameters that control our code. $PLOT\_EVERY\_MIN$ is the time interval at which we want to generate VTEC snapshots. For simplicity and to save time, we will set this parameter to 60 minutes. R is the average radius of the Earth, and ISH is the height of the ionospheric shell. These values are used to calculate the mapping function from STEC to VTEC. *zenith_error* is the measurement error of a single VTEC as a function of the zenith angle.

```
PLOT_EVERY_MIN = 60   # Produce map every 4 hours
R=6371 #km
ish = 450 #km
zenith_error = 1.5
def is_tick(ts: pd.Timestamp) -> bool:
    if PLOT_EVERY_MIN > 60:
        plot_every_h = PLOT_EVERY_MIN//60
        return (ts.second == 0) and (ts.hour % plot_every_h == 0) and (ts
    .minute == 0)
    else:
        return (ts.second == 0) and (ts.minute % PLOT_EVERY_MIN == 0)
```

Next, we set the path where our network measurements are located – these are the same files as in the case of index calculation. We also define the output path for saving our maps/models.

```
# I/O paths
path = './network'
out = './output'
```

Data loading and preparation is performed in the same way as for calculating ionospheric activity indices, which we discussed in the previous notebook.

```
df_all, sum  = gnx.load_stec_folder(folder=path,min_elev_deg=15,quiet=
    True)
m_f = 1 / np.sqrt(1 - (R / (R + ish)) ** 2 * np.cos(np.deg2rad(df_all['ev
    ''])) ** 2)
df_all['mf'] = m_f
df_all['stec'] = df_all['leveled_tec']
df_all['vtec'] = df_all['stec'] / df_all['mf']
df_all = df_all.reset_index().copy()
df_all['sv'] = df_all['sv'].str[:3]
```

```
df_all = df_all[df_all['time'].dt.second % 30 == 0]
var = zenith_error / np.sin(np.deg2rad(df_all['ev'].values))
df_all['meas_var'] = var
```

Next, we will define the grid on which we will interpolate VTEC, as well as its spatial resolution. We select a grid over our area to enable comparison of the results at as many points as possible with values from the GIM model. We set the spatial resolution of the maps to 0.1 x 0.1 degrees.

```
# GRID definition
lat_min, lat_max = 47.5, 57.5 # 2.5 degrees resolution in latitude - 5
    GIM points
lon_min, lon_max = 10, 30 # 5 degree resolution in longitude - 5 GIM
    points
lat_grid = np.arange(lat_min, lat_max, .1) # 0.1 degree x 0.1 degree
    spatial resolution
lon_grid = np.arange(lon_min, lon_max, .1)
```

The heart of the kriging module in GNX-py is the $IonoKrigingMonitor$ class. We will discuss its arguments and functionalities. Generally, you can perform kriging in three ways: using Ordinary Kriging, Universal Kriging, or the methodology described in (Sparks et al. 2011). We will refer to these methodology as the Sparks methodology. All these methods are implemented in GNX-py. You can take snapshots in a geocentric or solar-geomagnetic coordinate system. In the latter system, you must substitute the epoch in UTC time into the class constructor, because the orientation of the system is practically constant during the day, whereas in contrast to ECEF, the positions of stations and IPPs in this system change from epoch to epoch. You can find more about the solar-geomagnetic system in (Knecht and Shuman 1985; Mannucci et al. 1998).

When using the UK and OK methods, you can substitute your own variogram type and parameters. The Sparks methodology has its own unique parameters and separate variogram implementations. Optionally, you can set the default measurement variation if you have not defined it earlier in the input dataframe.

So, in the class constructor, you add: - the lat/lon grid of your map - the coordinate system: geocentric (GEO) or solar-geomagnetic (SM) - SM system constructors: observation time (sm_obstime), which you must edit in each epoch, transformer from the GNX-py library (SolarGeomagneticTransformer) - kriging method: UK, OK, or WAAS

For OK and UK methods: - type of variogram, variogram parameters (more about them in X and Y), and the number of nearest points for interpolation

For the WAAS methodology:

- the minimum and maximum radius in which IPPs are searched for in relation to a given IGP in order to estimate VTEC for it

- the target and minimum number of IPP points that are needed to estimate TEC for a given IGP
- the type of variogram from the GNX-py library
- optionally, the variance of measurements

```
mon = gnx.IonoKrigingMonitor(
    grid_lon=lon_grid,
    grid_lat=lat_grid,

    # coordinate system of IGP and IPP
    coord_frame = "GEO"  ,          # "GEO" or "SM"

    # Parameters to use with Solar Geomagnetic frame
    sm_transformer = None ,      # SolarGeomagneticTransformer
    sm_obstime = None  ,    # astropy.Time (scalar)

    # Kriging method
    kriging_mode = "UK",

    # OK/UK method parameters
    variogram_model= "spherical",
    variogram_parameters={"sill": 7.0, "range": 1500.0, "nugget": 1},
    n_closest_points = 60,

    # WAAS/Spark method parameters
    Rmin_km= 800.0,
    Rmax_km= 2100.0,
    Ntarget = 30,
    Nmin = 10,

    # WAAS/Sparks: model variogramu
    variogram = gnx.SparksVariogram(),

    # WAAS/Sparks: wariancje szumów (opcjonalnie w df)
    default_meas_var= 2.0,
)
```

We prepare containers for storing epoch results and run kriging for our grid in a loop, providing measurements from a given epoch.

```
# df_all = df_all[(df_all['time'].dt.hour>=12) & (df_all['time'].dt.hour
    <=13)]  ## can be used to save time
# prepare input
df_all = df_all.sort_values("time")

times_all = []
V_all, SS_all, ROT_all, ROTI_all = [], [], [], []

# loop over epoch

```

```
for t, df_ep in df_all.groupby("time"):
    if is_tick(t):
        V, SS, info = mon.krige_epoch(df_ep, epoch_time=t)
        times_all.append(t)
        V_all.append(V.astype(np.float32))
        SS_all.append(SS.astype(np.float32))

ds = gnx.save_grids_to_netcdf(times=times_all,V_all=V_all,SS_All=SS_all,
    lat_grid=lat_grid,lon_grid=lon_grid,out_path=out,out_name='model')
```

The results are stored in xarray.Dataset format and saved in netCDF format. The variables stored in the dataset are V (VTEC) and SS (VTEC variance from kriging). The data is stored in (*time* x *lat* x *lon*) format. GIM models are loaded into GNX-py in the same way. To display your model, you can use the *plot_model* function from the *vtec_viz.py* module that I have included for help in our tutorial.

```
print(ds.data_vars)
```

```
Data variables:
    V        (time, lat, lon) float32 2MB 11.18 11.21 11.23 ... 4.659
    4.663
    SS       (time, lat, lon) float32 2MB 1.049 1.048 1.048 ... 1.05 1.05
     1.05
```

```
list(ds.coords)
```

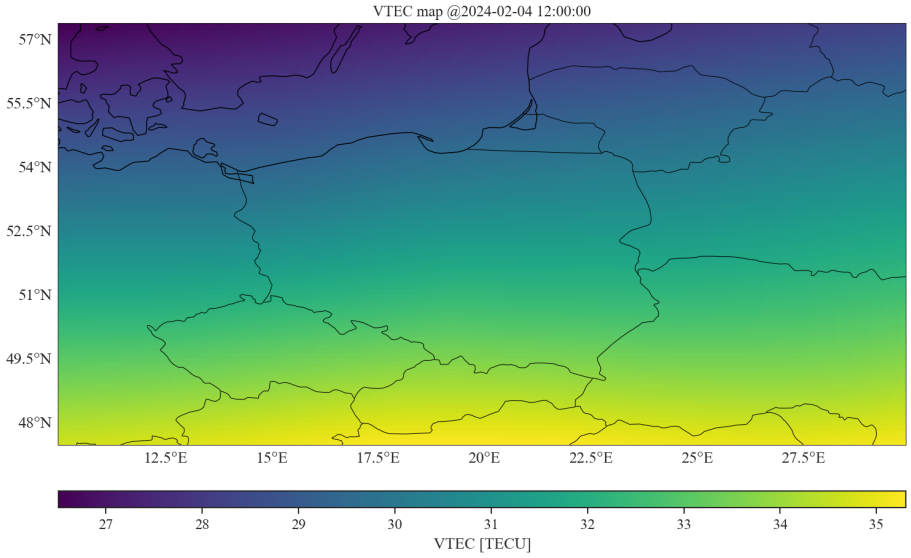
```
['time', 'lat', 'lon']
```

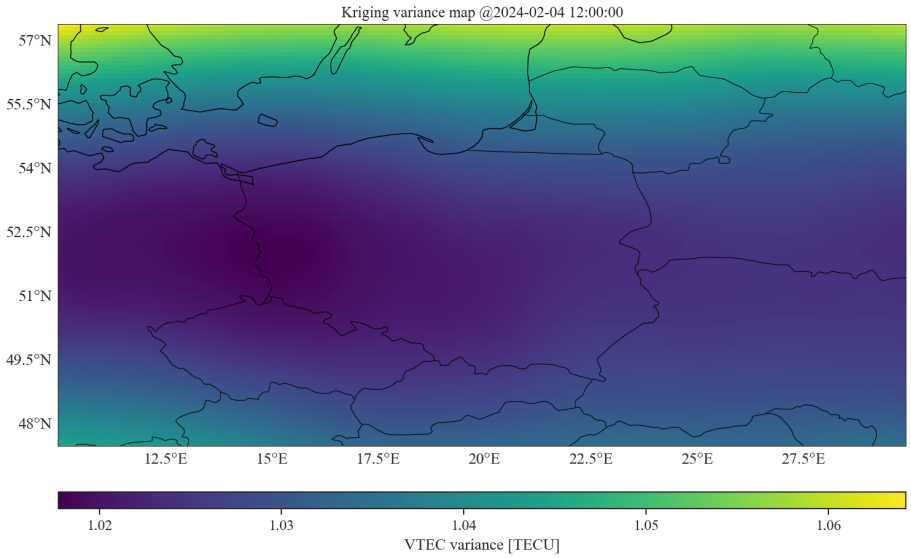
```
int(len(ds.time))
```

```
24
```

```
region = (10, 30, 50,57.5 )
model = "UK"
t_index =  int(len(ds.time)/2)
time = ds['time'][t_index].values
hh=pd.Timestamp(time).hour

vv.plot_model(
ds=ds,
var="V",
t_index=t_index,
cbar_orientation="horizontal",
cbar_location="bottom",
cbar_shrink=0.8,
cbar_aspect=40,
```

```
    cbar_label='VTEC [TECU]',
cbar_pad=0.0,
draw_coastlines=True,
figsize=(10,10),
cmap='viridis',
title=f'VTEC map @{pd.Timestamp(time)}',
savepath=f'{out}/{model}_MAP_{hh}.png')

vv.plot_model(
ds=ds,
var="SS",
t_index=t_index,
cbar_orientation="horizontal",
cbar_location="bottom",
cbar_shrink=0.8,
cbar_aspect=40,
    cbar_label='VTEC variance [TECU]',
cbar_pad=0.0,
draw_coastlines=True,
figsize=(10,10),
cmap='viridis',
title=f'Kriging variance map @{pd.Timestamp(time)}',
savepath=f'{out}/{model}_VAR_{hh}.png')
```






```
(<Figure size 1500x1500 with 2 Axes>,
 <GeoAxes: title={'center': 'Kriging variance map @2024-02-04
    12:00:00'}>)
```

Let us compare our model with NTCM and GIM. It is worth recalling what we said at the beginning—the GIM has a relatively low spatial resolution compared to ours (0.1 x 0.1 degrees), so when comparing, we compare the values at five common points. First, let's load our IONEX file using GNX-py.

```
reader = gnx.GIMReader(tec_path='../data/
    COD0OPSFIN_20240350000_01D_01H_GIM.INX')
ionex = reader.read()
inx_tec=ionex.tec
```

To calculate TEC from the NTCM model, we need Galileo ionospheric model coefficients from navigation message.

```
nav='../data/BRDC00IGS_R_20240350000_01D_MN.rnx'
processor = gnx.GNSSDataProcessor2(nav_path=nav)
hdr = processor.nav_header_reader()
gal_alpha = hdr[-1]
gal_alpha
```

```
[152.0, -0.22656, -0.0070496]
```

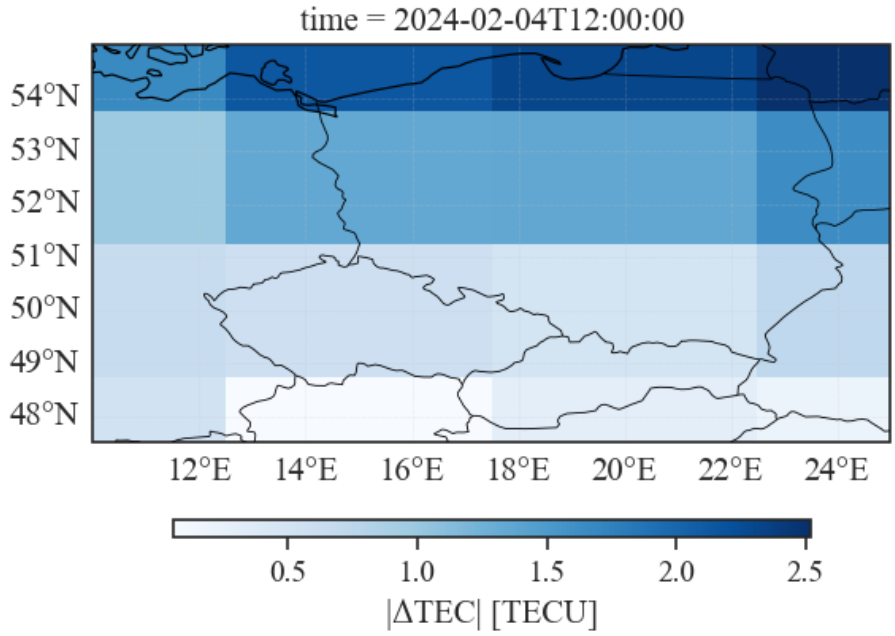
We will now generate an NTCM grid corresponding to our Kriging grid. We will do this using the *compute_ntcm_grid* function. The arguments are: the geocentric coordinate grid and the epoch list.

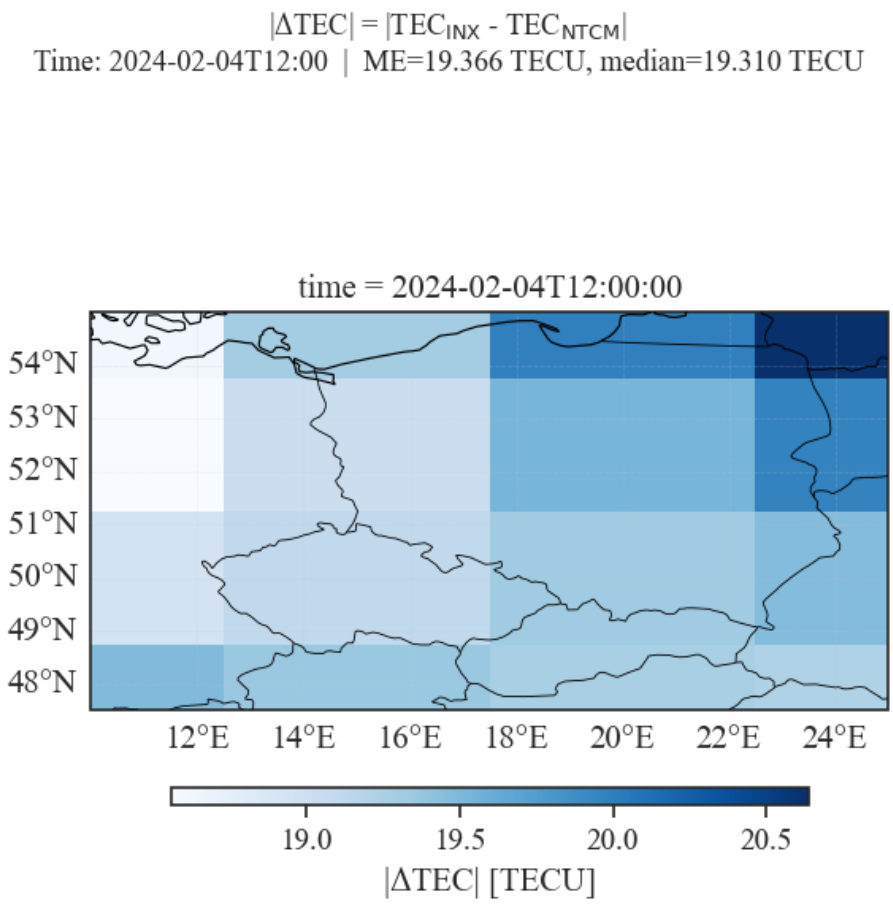
```
lat_grid = ds['V'].lat.values
lon_grid = ds['V'].lon.values
times    = ds.time.values

ds_ntcm = gnx.compute_ntcm_grid(lat_grid, lon_grid, times, gal_alpha=
    gal_alpha)
```

Datasets are convenient because the comparison will automatically take place at common points. I have prepared an auxiliary function *compare_plot* that allows us to compare TEC values between our models. Keep in mind that when comparing Kriging and GIM, my goal is not to show which model is "better" since they are produced using completely different methods. However, we will treat GIM as a reference and calculate the average TEC error in the area, assuming that the average TEC level should be similar.

```
mns = []
mns_ntcm =[]
epochs = ds.time.values
for num, time in enumerate(ds.time.values):
    if num == len(epochs)/2:
        show=True
    else:
        show=False
    fig, ax, mean_abs, median_abs = vv.compare_plot(
        ds_model=ds,
        ds_ref=inx_tec,
        time=time,
        var_model="V",
        var_ref="TEC",
        cmap="Blues",
        cbar_orientation="horizontal",
        cbar_pad=0.1,
        cbar_shrink=0.8,
        cbar_aspect=40,
        figsize=(5,5),
        title_prefix = r"Δ|TEC| = |TEC$_\mathrm{INX}$ - TEC$_\mathrm{
    KRIGING}$|",
        show=show
    )
    mns.append(mean_abs)
    fig, ax, mean_abs, median_abs = vv.compare_plot(
        ds_model=ds_ntcm,
        ds_ref=inx_tec,
        time=time,
        var_model="V",
        var_ref="TEC",
        cmap="Blues",
        cbar_orientation="horizontal",
        cbar_pad=0.1,
```

```
        cbar_shrink=0.8,
        cbar_aspect=40,
        figsize=(5,5),
        title_prefix = r"Δ|TEC| = |TEC$_\mathrm{INX}$ - TEC$_\mathrm{NTCM}$|",
        show=show
    )
    mns_ntcm.append(mean_abs)
```

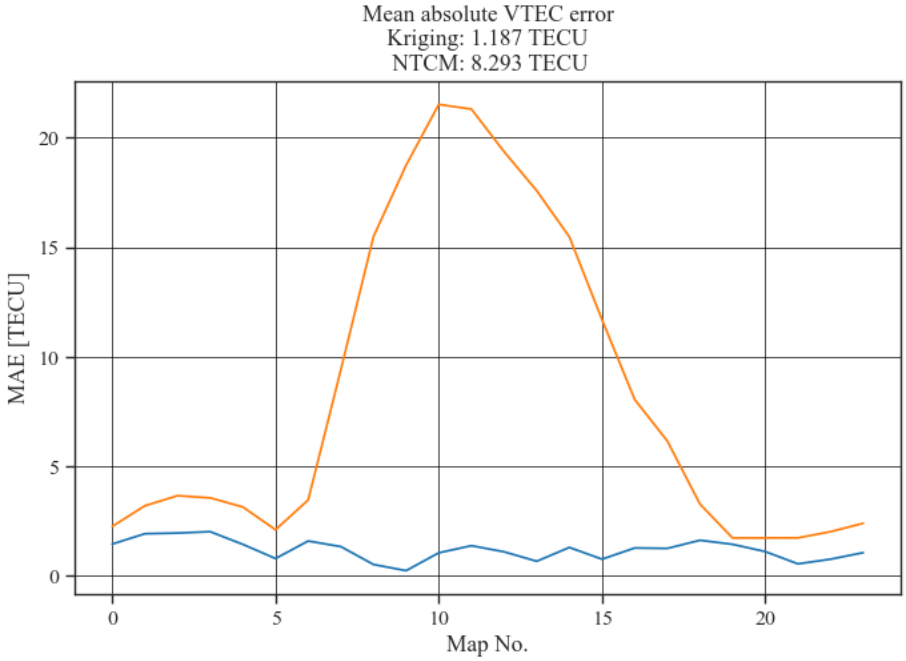




As you can see, the TEC differences between NTCM and GIM differ drastically from the differences between Kriging and GIM. I discussed some intuitions about the reasons for this at the beginning of this tutorial. Finally, let's plot the average error in each epoch on a common graph.

```
plt.figure(figsize=(8,5))
plt.title(f'Mean absolute VTEC error\n Kriging: {np.mean(np.asarray(mns))
    .round(3)} TECU \n NTCM: {np.mean(np.asarray(mns_ntcm)).round(3)}
    TECU')

plt.plot(range(len(mns)), mns,label='MAE for Kriging')
plt.plot(range(len(mns_ntcm)), mns_ntcm,label='MAE for NTCM')
plt.grid(color='k', linestyle='-', linewidth=0.5)
plt.xlabel('Map No.')
plt.ylabel('MAE [TECU]')
```

```
Text(0, 0.5, 'MAE [TECU]')
```



You already know how I use the Kriging module in GNX-py. Together with functions that calculate ionospheric activity indices, you can use kriging to monitor/visualize changes in electron concentration, e.g., during disturbances in the ionosphere. You already know the SPP and PPP modules – you can try to modify the code and connect the TEC model obtained by kriging to positioning and compare the results between the models. To do this, you need to have the modeled VTEC state for a given measurement epoch – you can obtain this by modifying the $PLOT\_EVERY\_MIN$ parameter.

The VTEC model generated using kriging is stored in xarray.Dataset format and saved in netcdf format. The file can be loaded using the *load_dataset*() function from xarray library. The xarray library offers a range of tools for data analysis, processing and visualisation and is quite convenient for grid data.

References

Arikan, F., H. Nayir, U. Sezen, and O. Arikan. 2008. "Estimation of Single-Station Inter-Frequency Receiver Bias Using GPS-TEC." *Radio Science* 43 (4): RS4004. https://doi.org/10.1029/2007RS003785.

Ashby, Neil. 2003. "Relativity in the Global Positioning System." *Living Reviews in Relativity* 6 (1): 1. https://doi.org/10.12942/lrr-2003-1.

Carmo, C. S. do, C. M. Denardini, Â. Figueiredo, et al. 2021. "Evaluation of

Different Methods for Calculating the ROTI Index over the Brazilian Sector." *Radio Science* 56 (4): e2020RS007140. https://doi.org/10.1029/2020RS007140.

Cherniak, Iurii, Andrzej Krankowski, and Irina Zakharenkova. 2018. "ROTI Maps: A New IGS Ionospheric Product Characterizing the Ionospheric Irregularities Occurrence." *GPS Solutions* 22: 69. https://doi.org/10.1007/s10291-018-0730-1.

Collins, John P. 1999. *Assessment and Development of a Tropospheric Delay Model for Aircraft Users of the Global Positioning System*. M.Sc.E. thesis / Technical Report No. 203. Department of Geodesy; Geomatics Engineering, University of New Brunswick.

Esri. n.d. *How Kriging Works*. ArcGIS Pro documentation (online). https://pro.arcgis.com/en/pro-app/3.4/tool-reference/3d-analyst/how-kriging-works.htm.

European Union Agency for the Space Programme. 2021. *European GNSS (Galileo) Open Service Signal-in-Space Interface Control Document*. Issue 2.0. European Union. https://www.gsc-europa.eu/sites/default/files/sites/all/files/Galileo-OS-SIS-ICD.pdf.

German Aerospace Center (DLR). 2023. *NTCM-g Ionospheric Model Description, Version 1.0*. European GNSS Service Centre (GSC) / DLR. https://doi.org/10.5281/zenodo.7326066.

Glaner, M. F. 2022. *Towards Instantaneous PPP Convergence Using Multiple GNSS Signals*. TU Wien Thesis / Online PDF. https://repositum.tuwien.at/bitstream/20.500.12708/95695/1/Glaner%20Marcus%20Franz%20-%202022%20-%20Towards%20instantaneous%20PPP%20convergence%20using...pdf.

Hatanaka, Y. 2008. "A Compression Format and Tools for GNSS Observation Data." *Bulletin of the Geospatial Information Authority of Japan* 55: 21–30.

Hirsch, Michael, Nikolay Mayorov, Joakim Strandberg, et al. 2023. *Georinex: RINEX 2/3/4 Reader and Converter to NetCDF4/HDF5. Computer software*, Zenodo. https://doi.org/10.5281/zenodo.10130117.

HofmannWellenhof, Bernhard, Herbert Lichtenegger, and Elmar Wasle. 2008. *GNSS – Global Navigation Satellite Systems: GPS, GLONASS, Galileo and More*. Springer Vienna. https://doi.org/10.1007/978-3-211-73017-1.

Jakowski, N., and M. M. Hoque. 2019. "Estimation of Spatial Gradients and Temporal Variations of the Total Electron Content Using Ground-Based

GNSS Measurements." *Space Weather* 17: 339–56. https://doi.org/10.1029/2018SW002119.

Klobuchar, John A. 1987. "Ionospheric Time-Delay Algorithms for Single-Frequency GPS Users." *IEEE Transactions on Aerospace and Electronic Systems* AES-23 (3): 325–31. https://doi.org/10.1109/TAES.1987.310829.

Knecht, D. J., and B. M. Shuman. 1985. "The Geomagnetic Field." Chap. 4 in *Handbook of Geophysics and the Space Environment*, edited by A. S. Jursa. Air Force Geophysics Laboratory, USAF. https://www.cnofs.org/Handbook_of_Geophysics_1985/Chptr04.pdf.

Kouba, Jan, and Pierre Héroux. 2001. "Precise Point Positioning Using IGS Orbit and Clock Products." *GPS Solutions* 5 (2): 12–28. https://doi.org/10.1007/PL00012883.

Labbe, Roger R. Jr. 2020. *Kalman and Bayesian Filters in Python*. GitHub repository. https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python.

Mannucci, A. J., B. D. Wilson, D. N. Yuan, C. M. Ho, U. J. Lindqwister, and B. A. Iijima. 1998. "A Global Mapping Technique for GPS-Derived Ionospheric Total Electron Content Measurements." *Radio Science* 33 (3): 565–82. https://doi.org/10.1029/97RS02707.

Maurer, Jeremy. 2022. *Kriging - a Mathematical Overview*. YouTube video. https://youtu.be/HDW-HLMxcnE?si=HigG7a3M9qVWHDS0.

Maxwell, Aaron. 2018. *Semivariogram Explained*. YouTube video. https://www.youtube.com/watch?v=L-hnxGq74q0.

Mervart, Leos. 1995. "Ambiguity Resolution Techniques in Geodetic and Geodynamic Applications of the Global Positioning System." PhD thesis, University of Bern, Astronomical Institute. https://zhw-b.s3.cloud.switch.ch/aiub/papers/lmdiss.ps.

Montenbruck, Oliver, Peter Steigenberger, and André Hauschild. 2015. "Broadcast Versus Precise Ephemerides: A Multi-GNSS Perspective." *GPS Solutions* 19 (2): 321–33. https://doi.org/10.1007/s10291-014-0390-8.

Montenbruck, Oliver, Peter Steigenberger, and André Hauschild. 2018. "Multi-GNSS Signal-in-Space Range Error Assessment – Methodology and Results." *Advances in Space Research* 61 (12): 3020–38. https://doi.org/10.1016/j.asr.2018.03.041.

National Executive Committee for Space-Based Positioning, Navigation, and

Timing. 2023. *Global Positioning Systems Directorate: Navstar GPS Space Segment/Navigation User Interfaces: IS-GPS-200N*. U.S. Government. https://www.gps.gov/technical/icwg/IS-GPS-200N.pdf.

Niell, A. E. 1996. "Global Mapping Functions for the Atmosphere Delay at Radio Wavelengths." *Journal of Geophysical Research: Solid Earth* 101 (B2): 3227–46. https://doi.org/10.1029/95JB03048.

Nischan, Thomas. 2016. *GFZRNX – RINEX GNSS Data Conversion and Manipulation Toolbox*. GFZ Data Services. https://doi.org/10.5880/GFZ.1.1.2016.002.

Saastamoinen, J. 1972. "Atmospheric Correction for the Troposphere and Stratosphere in Radio Ranging Satellites." In *The Use of Artificial Satellites for Geodesy*, edited by Soren W. Henriksen, Armando Mancini, and Bernard H. Chovitz, vol. 15. Geophysical Monograph Series. American Geophysical Union. https://doi.org/10.1029/GM015p0247.

Sardón, E., A. Rius, and N. Zarraoa. 1994. "Estimation of the Transmitter and Receiver Differential Biases and the Ionospheric Total Electron Content from Global Positioning System Observations." *Radio Science* 29 (3): 577–86. https://doi.org/10.1029/94RS00449.

Savitzky, Abraham, and Marcel J. E. Golay. 1964. "Smoothing and Differentiation of Data by Simplified Least Squares Procedures." *Analytical Chemistry* 36 (8): 1627–39. https://doi.org/10.1021/ac60214a047.

Schaer, S., W. Gurtner, and J. Feltens. 1998. *IONEX: The IONosphere Map EXchange format, Version 1.1*. IGS AC Workshop, Darmstadt, 9–11 February 1998; specification available at AIUB/IGS. https://ftp.aiub.unibe.ch/ionex/draft/ionex11.pdf.

Schaer, Stefan. 1999. "Mapping and Predicting the Earth's Ionosphere Using the Global Positioning System." PhD thesis, Astronomical Institute, University of Bern. https://mediatum.ub.tum.de/doc/1365835/647845.pdf.

Schaer, Stefan. 2018. *SINEX BIAS — Solution (Software/Technique) INdependent EXchange Format for GNSS Biases, Version 1.00*. Swisstopo / AIUB. https://files.igs.org/pub/data/format/sinex_bias_100.pdf.

Schönemann, Erik. 2014. "Analysis of GNSS Raw Observations in PPP Solutions." PhD thesis, Technische Universität Darmstadt. https://tuprints.ulb.tu-darmstadt.de/server/api/core/bitstreams/b434977f-79d8-410c-833a-00a5d316f723/content.

Smith, John O. 2025. "The Chebyshev Filter — Theory and Application." *DSP*

Guide – Online Resource. https://dspguide.com/ch20/1.htm.

Sparks, Lawrence, Juan Blanch, and Nitin Pandya. 2011. "Estimating Ionospheric Delay Using Kriging: 1. Methodology." *Radio Science* 46 (6): RS0D21. https://doi.org/10.1029/2011RS004667.

Teunissen, Peter J. G., and Oliver Montenbruck, eds. 2017. *Springer Handbook of Global Navigation Satellite Systems*. Springer International Publishing. https://link.springer.com/book/10.1007/978-3-319-42928-1.

Wielgosz, P., D. Grejner-Brzezińska, and I. Kashani. 2003. "Regional Ionosphere Mapping with Kriging and Multiquadric Methods." *Journal of Global Positioning Systems* 2 (1): 48–55. https://doi.org/10.5081/jgps.2.1.48.