PyGNSS-RT Technical Design Document

Python-Based Real-Time PPP-AR GNSS Positioning System

Comprehensive Technical Specification and Implementation Guide

GNSS Research Group
Institute for Earth Sciences and Space Geodesy

`pygnss_rt v1.4.0`

January 15, 2026

Abstract

This document provides a comprehensive technical specification for the `pygnss_rt` Python package, a production-grade framework for Real-Time Precise Point Positioning with Ambiguity Resolution (PPP-AR). The system integrates with the Bernese GNSS Software v5.4 for core geodetic computations while providing a sophisticated Python orchestration layer for data management, product acquisition, and operational automation. This document covers the complete system architecture, mathematical foundations of PPP-AR, advanced error modeling including VMF3 tropospheric mapping functions and Observation Specific Biases (OSB), the LAMBDA-based ambiguity resolution strategy, and detailed Python implementation patterns. The system supports multi-GNSS processing (GPS, GLONASS, Galileo, BeiDou) with sub-centimeter positioning accuracy under favorable conditions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Document Purpose and Scope

This technical design document serves as the authoritative reference for the `pygnss_rt` real-time GNSS processing system. It provides:

- Complete mathematical formulations for PPP-AR processing
- Detailed system architecture and data flow specifications
- Implementation details for atmospheric and bias modeling
- Ambiguity resolution algorithms and validation strategies
- Python code structure and optimization techniques

1.2 System Overview

The `pygnss_rt` package implements a sophisticated orchestration framework for GNSS data processing, designed for:

1. **Near Real-Time (NRT) Operations**: Hourly and sub-hourly processing with configurable latencies
2. **Multi-GNSS Support**: GPS (G), GLONASS (R), Galileo (E), and BeiDou (C) constellations
3. **PPP-AR Processing**: Integer ambiguity resolution using CODE products
4. **Atmospheric Monitoring**: ZTD estimation and IWV derivation for meteorology
5. **Network Processing**: Five international networks (IGS, EUREF, GB, RGP, Supersites)

1.3 Key Capabilities

Table 1.1: PyGNSS-RT System Capabilities

Feature	**Implementation**	**Status**
PPP-AR Processing	CODE products + BSW	Operational
Multi-GNSS (GRE)	GPS+GLONASS+Galileo	Operational
VMF3 Troposphere	TU Wien grids	Operational
OSB/BIA Handling	CODE signal biases	Operational
Hourly Processing	3-hour latency	Operational
Sub-hourly (15-min)	Climate research	Operational
IWV Generation	Bevis formulation	Operational

Chapter 2

System Architecture and Data Flow

2.1 Architectural Overview

The `pygnss_rt` system follows a modular pipeline architecture with clear separation of concerns:

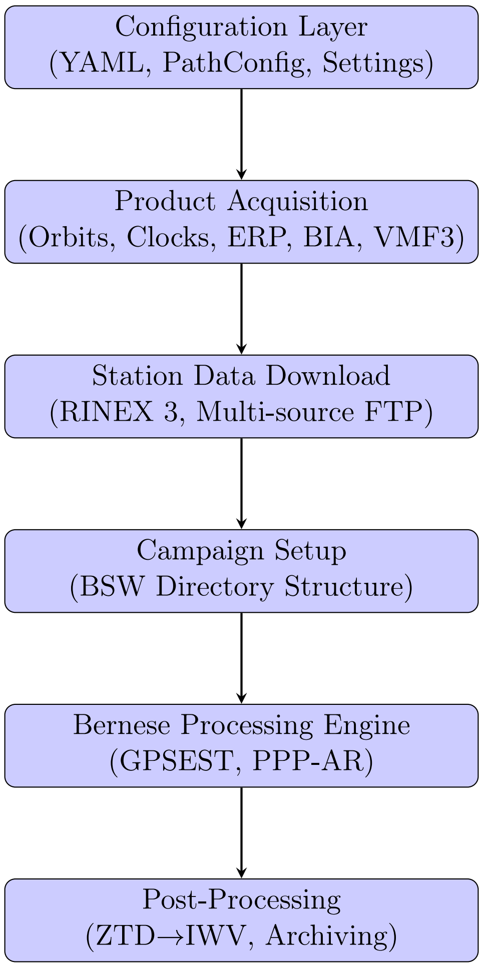
Configuration Layer
(YAML, PathConfig, Settings)
Product Acquisition
(Orbits, Clocks, ERP, BIA, VMF3)
Station Data Download
(RINEX 3, Multi-source FTP)
Campaign Setup
(BSW Directory Structure)
Bernese Processing Engine
(GPSEST, PPP-AR)
Post-Processing
(ZTD→IWV, Archiving)

Figure 2.1: PyGNSS-RT Processing Pipeline Architecture

2.2 Module Organization

The codebase comprises approximately 43,000 lines of Python organized into specialized modules:

Table 2.1: Module Structure and Responsibilities

Module	Purpose	LOC
`core/`	Configuration, paths, orchestration	1,500
`processing/`	PPP pipeline, networks, coordinates	4,000+
`bsw/`	Bernese GNSS Software interface	1,200+
`data_access/`	FTP/HTTP downloads	2,500+
`stations/`	Station management, metadata	5,100+
`atmosphere/`	ZTD→IWV conversion	500+
`utils/`	Dates, RINEX, compression	6,000+

2.3 Real-Time Data Ingestion

2.3.1 Product Selection Logic

The system implements a tiered product selection strategy based on latency requirements:

$$\text{Product Tier} = \begin{cases} \text{Final} & \text{if } \Delta t > 14 \text{ days} \\ \text{Rapid} & \text{if } 2 < \Delta t \leq 14 \text{ days} \\ \text{Ultra-rapid} & \text{if } \Delta t \leq 2 \text{ days} \end{cases} \tag{2.1}$$

where Δt is the time elapsed since the observation epoch.

```python
class ProductTier(Enum):
    FINAL = "final"      # Highest accuracy, ~14 day latency
    RAPID = "rapid"      # Good accuracy, ~17 hour latency
    ULTRA = "ultra"      # Near real-time, ~3 hour latency
    PREDICTED = "predicted"  # Forecast products

def select_product_tier(obs_date: date, current_date: date) -> ProductTier:
    """Select appropriate product tier based on latency."""
    delta_days = (current_date - obs_date).days

    if delta_days > 14:
        return ProductTier.FINAL
    elif delta_days > 2:
        return ProductTier.RAPID
    else:
        return ProductTier.ULTRA
```

Listing 2.1: Product Tier Selection Logic

2.3.2 Data Source Configuration

Products are acquired from multiple redundant sources:

```yaml
# From config/ftp_servers.yaml
servers:
  CDDIS:
    host: "gdc.cddis.eosdis.nasa.gov"
    protocol: "https"
    auth: "earthdata"  # NASA Earthdata Login
    products: ["orbit", "clock", "erp", "bia"]

  CODE:
    host: "ftp.aiub.unibe.ch"
    protocol: "ftp"
    products: ["orbit", "clock", "bia", "ion"]
```

```

VMF3:
  host: "vmf.geo.tuwien.ac.at"
  protocol: "https"
  products: ["vmf3"]
```
Listing 2.2: FTP Server Configuration

2.3.3 Latency Management

The system supports configurable latency windows for different processing modes:

Table 2.2: Processing Modes and Latency Configuration

Mode	Session Length	Default Latency	Use Case
Daily	24 hours	21 days	Reference coordinates
Hourly	1 hour	3 hours	NRT troposphere
Sub-hourly	15 minutes	1 hour	Severe weather

2.4 Buffering Strategies

2.4.1 Product Caching

Downloaded products are cached to minimize redundant downloads:

```python
class ProductDownloader:
    def __init__(self, cache_dir: Path):
        self.cache_dir = cache_dir
        self.cache_index: Dict[str, CacheEntry] = {}

    def get_product(self, product_type: str, date: GNSSDate) -> Path:
        """Get product with caching."""
        cache_key = f"{product_type}_{date.year}_{date.doy}"

        if cache_key in self.cache_index:
            entry = self.cache_index[cache_key]
            if entry.is_valid():
                return entry.local_path

        # Download and cache
        local_path = self._download_product(product_type, date)
        self.cache_index[cache_key] = CacheEntry(
            local_path=local_path,
            timestamp=datetime.now(),
            ttl_hours=168  # 1 week cache
        )
        return local_path
```
Listing 2.3: Product Caching Strategy

2.4.2 Parallel Download Management

Station data downloads are parallelized for efficiency:

```python
from concurrent.futures import ThreadPoolExecutor, as_completed

class StationDownloader:
    MAX_WORKERS = 12  # Concurrent download threads

    def download_stations(self, stations: List[Station],
                          date: GNSSDate) -> Dict[str, DownloadResult]:
        """Download RINEX files for multiple stations in parallel."""
        results = {}

        with ThreadPoolExecutor(max_workers=self.MAX_WORKERS) as executor:
            futures = {
                executor.submit(self._download_single, sta, date): sta
                for sta in stations
            }

            for future in as_completed(futures):
                station = futures[future]
                try:
                    results[station.id] = future.result()
                except Exception as e:
                    results[station.id] = DownloadResult(
                        success=False, error=str(e)
                    )

        return results
```

Listing 2.4: Parallel Download Implementation

Chapter 3

Kalman Filter Design: The Estimation Engine

3.1 Overview

The `pygnss_rt` system delegates core geodetic estimation to the Bernese GNSS Software (BSW) v5.4, which implements a sophisticated Kalman filter within the `GPSEST` program. This chapter documents the mathematical foundations underlying the estimation process.

3.2 State Vector Definition

The complete PPP state vector $\boldsymbol{x}$ comprises:

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{r} \\ \delta t_r \\ \text{ZTD} \\ G_N \\ G_E \\ \boldsymbol{N} \\ \text{ISB} \end{bmatrix} \in \mathbb{R}^n \tag{3.1}$$

where the components are:

Table 3.1: State Vector Components

Symbol	Description	Dimension	Units
$\boldsymbol{r}$	Receiver position (X, Y, Z)	3	meters
δt_r	Receiver clock offset	1	meters
ZTD	Zenith Tropospheric Delay	1	meters
G_N, G_E	Tropospheric gradients (N, E)	2	millimeters
$\boldsymbol{N}$	Float ambiguities	$n_{\text{sat}} \times n_{\text{freq}}$	cycles
ISB	Inter-System Biases	$n_{\text{sys}} - 1$	meters

3.2.1 Position State

Receiver coordinates are expressed in the Earth-Centered Earth-Fixed (ECEF) frame:

$$\boldsymbol{r} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_{\text{ITRF}} \tag{3.2}$$

For static positioning (default mode), these are estimated as constant parameters. For kinematic applications, a random walk process model is applied.

3.2.2 Clock State

The receiver clock offset δt_r absorbs timing errors:

$$\delta t_r = c \cdot (t_{\text{receiver}} - t_{\text{GPS}}) \tag{3.3}$$

where c is the speed of light. For multi-GNSS, system-specific clock offsets are required.

3.2.3 Tropospheric State

The tropospheric state includes the zenith delay and horizontal gradients:

$$\tau_{\text{trop}}(\epsilon, \alpha) = \text{ZTD} \cdot m_w(\epsilon) + G_N \cdot m_g(\epsilon) \cos(\alpha) + G_E \cdot m_g(\epsilon) \sin(\alpha) \tag{3.4}$$

where:

- ϵ = satellite elevation angle
- α = satellite azimuth angle
- $m_w(\epsilon)$ = wet mapping function (VMF3)
- $m_g(\epsilon)$ = gradient mapping function

3.2.4 Ambiguity State

For each satellite-frequency combination, the carrier phase ambiguity N_i^j is estimated:

$$N_i^j \in \mathbb{R} \quad \text{(float)} \rightarrow N_i^j \in \mathbb{Z} \quad \text{(fixed)} \tag{3.5}$$

The ambiguity resolution process converts float estimates to integer values.

3.3 State Transition Model

The discrete-time state transition follows:

$$\boldsymbol{x}_{k+1} = \boldsymbol{\Phi}_k \boldsymbol{x}_k + \boldsymbol{w}_k \tag{3.6}$$

where $\boldsymbol{\Phi}_k$ is the state transition matrix and $\boldsymbol{w}_k \sim \mathcal{N}(0, \mathbf{Q}_k)$ is process noise.

3.3.1 Transition Matrix

For static PPP with stochastic troposphere:

$$\boldsymbol{\Phi}_k = \begin{bmatrix} \mathbf{I}_3 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & 1 & 0 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & 0 & 1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I}_2 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I}_{n_a} \end{bmatrix} \tag{3.7}$$

where n_a is the number of ambiguity parameters.

3.3.2 Process Noise Covariance

The process noise matrix $\mathbf{Q}_k$ models temporal variations:

$$\mathbf{Q}_k = \begin{bmatrix} \sigma_r^2\mathbf{I}_3 & & & & \\ & \sigma_{\delta t}^2 & & & \\ & & q_{\text{ZTD}}\Delta t & & \\ & & & q_g \Delta t \mathbf{I}_2 & \\ & & & & \mathbf{0}_{n_a} \end{bmatrix} \tag{3.8}$$

Typical values:

- Position (static): $\sigma_r = 0$ (constant)
- Clock: $\sigma_{\delta t} = 10^6$ m (white noise, re-estimated each epoch)
- ZTD: $q_{\text{ZTD}} = (5 \text{ mm})^2/\text{hour}$ (random walk)
- Gradients: $q_g = (0.3 \text{ mm})^2/\text{hour}$
- Ambiguities: $\sigma_N = 0$ (constant between cycle slips)

3.4 Measurement Model

3.4.1 Observation Equations

The fundamental GNSS observables are pseudorange (P) and carrier phase (L):

$$P_i^s = \rho_i^s + c(\delta t_r - \delta t^s) + T_i^s + I_i^s + b_{P,r} - b_P^s + \epsilon_P \tag{3.9}$$
$$L_i^s = \rho_i^s + c(\delta t_r - \delta t^s) + T_i^s - I_i^s + \lambda N_i^s + b_{L,r} - b_L^s + \epsilon_L \tag{3.10}$$

where:

Term	**Description**
ρ_i^s	Geometric range from receiver i to satellite s
$\delta t_r, \delta t^s$	Receiver and satellite clock offsets
T_i^s	Tropospheric delay
I_i^s	Ionospheric delay (frequency-dependent)
$b_{P,r}, b_P^s$	Code biases (receiver and satellite)
$b_{L,r}, b_L^s$	Phase biases (receiver and satellite)
λ	Carrier wavelength
N_i^s	Integer ambiguity
ϵ_P, ϵ_L	Measurement noise

3.4.2 Ionosphere-Free Combination

To eliminate first-order ionospheric effects, the ionosphere-free (IF) combination is formed:

$$L_{\text{IF}} = \frac{f_1^2 L_1 - f_2^2 L_2}{f_1^2 - f_2^2} \tag{3.11}$$

For GPS L1/L2:

$$L_{\text{IF}} = 2.5457 \cdot L_1 - 1.5457 \cdot L_2 \tag{3.12}$$

3.4.3 Linearized Measurement Model

The measurement model is linearized around the current state estimate:

$$\boldsymbol{y}_k = \mathbf{H}_k \boldsymbol{x}_k + \boldsymbol{v}_k \tag{3.13}$$

where $\boldsymbol{v}_k \sim \mathcal{N}(0, \mathbf{R}_k)$ is measurement noise.
The design matrix $\mathbf{H}_k$ contains partial derivatives:

$$\mathbf{H}_k = \begin{bmatrix} \frac{\partial \rho}{\partial X} & \frac{\partial \rho}{\partial Y} & \frac{\partial \rho}{\partial Z} & 1 & m_w & m_g \cos\alpha & m_g \sin\alpha & \lambda \mathbf{I} & \cdots \end{bmatrix} \tag{3.14}$$

The geometric partial derivatives are:

$$\frac{\partial \rho}{\partial X} = -\frac{X^s - X_r}{\rho}, \quad \frac{\partial \rho}{\partial Y} = -\frac{Y^s - Y_r}{\rho}, \quad \frac{\partial \rho}{\partial Z} = -\frac{Z^s - Z_r}{\rho} \tag{3.15}$$

3.4.4 Measurement Noise Covariance

The measurement covariance $\mathbf{R}_k$ is typically elevation-dependent:

$$\sigma^2(\epsilon) = \sigma_0^2 \cdot \left(1 + \frac{1}{\sin^2(\epsilon)}\right) \tag{3.16}$$

Alternative weighting schemes:

- COSZ: $\sigma^2 = \sigma_0^2 / \cos^2(z)$ where $z = 90 - \epsilon$
- Exponential: $\sigma^2 = \sigma_0^2 \cdot e^{-\epsilon/\epsilon_0}$

3.5 Filter Implementation

3.5.1 Prediction Step

$$\hat{\boldsymbol{x}}_{k|k-1} = \boldsymbol{\Phi}_k \hat{\boldsymbol{x}}_{k-1|k-1} \tag{3.17}$$
$$\mathbf{P}_{k|k-1} = \boldsymbol{\Phi}_k \mathbf{P}_{k-1|k-1} \boldsymbol{\Phi}_k^\mathsf{T} + \mathbf{Q}_k \tag{3.18}$$

3.5.2 Update Step

$$\mathbf{K}_k = \mathbf{P}_{k|k-1} \mathbf{H}_k^\mathsf{T} (\mathbf{H}_k \mathbf{P}_{k|k-1} \mathbf{H}_k^\mathsf{T} + \mathbf{R}_k)^{-1} \tag{3.19}$$
$$\hat{\boldsymbol{x}}_{k|k} = \hat{\boldsymbol{x}}_{k|k-1} + \mathbf{K}_k (\boldsymbol{y}_k - \mathbf{H}_k \hat{\boldsymbol{x}}_{k|k-1}) \tag{3.20}$$
$$\mathbf{P}_{k|k} = (\mathbf{I} - \mathbf{K}_k \mathbf{H}_k) \mathbf{P}_{k|k-1} \tag{3.21}$$

3.5.3 BSW Integration

The Bernese GPSEST program implements a batch least-squares with sequential processing capability:

```yaml
# From bsw_configs/iGNSS_D_PPP_AR_IG_IGS54_direct.yaml
GPSEST:
  # Coordinate estimation
  COORDEST: STATIC      # Static positioning

  # Troposphere estimation
  TROPOS: VMF3          # VMF3 mapping functions
  TRPEST: ZPD           # Estimate ZPD
```

```
TRPGRAD: 2            # N/S and E/W gradients

# Elevation weighting
ELVWGT: COSZ          # Cosine of zenith angle
MINEL: 7              # 7 degree cutoff

# Ambiguity handling
AMBRES: SIGMA         # Sigma-dependent resolution
AMBWGT: 0.001         # Ambiguity weight
```

Listing 3.1: BSW Options Configuration

Chapter 4

Advanced Error Modeling

4.1 Tropospheric Delay Modeling

4.1.1 Overview

The tropospheric delay is the dominant error source after ionospheric correction, contributing 2–25 meters of zenith delay depending on atmospheric conditions.

4.1.2 Delay Decomposition

Total tropospheric delay is decomposed into hydrostatic and wet components:

$$\mathrm{ZTD} = \mathrm{ZHD} + \mathrm{ZWD} \tag{4.1}$$

The slant delay at elevation ϵ is:

$$T(\epsilon) = \mathrm{ZHD} \cdot m_h(\epsilon) + \mathrm{ZWD} \cdot m_w(\epsilon) \tag{4.2}$$

4.1.3 Vienna Mapping Functions 3 (VMF3)

The VMF3 provides site-specific mapping functions derived from numerical weather models:

$$m(\epsilon) = \frac{1}{\sin(\epsilon) + \frac{a}{\tan(\epsilon) + \frac{b}{\sin(\epsilon)+c}}} \tag{4.3}$$

where a, b, c are the continued fraction coefficients varying with location and time.

VMF3 Grid Download Implementation

```python
class ProductDownloader:
    VMF3_BASE_URL = "https://vmf.geo.tuwien.ac.at/trop_products/GRID/"
    VMF3_HOURS = ["00", "06", "12", "18"]  # 6-hourly grids

    def download_vmf3(self, date: GNSSDate, output_dir: Path) -> DownloadResult:
        """Download and combine VMF3 grid files."""
        grid_files = []
        date_str = date.to_datetime().strftime("%Y%m%d")

        for hour in self.VMF3_HOURS:
            url = f"{self.VMF3_BASE_URL}/{date.year}/VMF3_{date_str}.H{hour}"
            local_file = output_dir / f"VMF3_{date_str}.{hour}"

            if self._download_file(url, local_file):
                grid_files.append(local_file)

        # Combine into Bernese format
        output_file = output_dir / f"VMF3_{date.yy}{date.doy:03d}0.GRD"
        self._combine_vmf3_grids(grid_files, output_file)

        return DownloadResult(success=True, local_path=output_file)
```
Listing 4.1: VMF3 Download and Processing

VMF3 Physical Constants

```python
# From atmosphere/ztd2iwv.py
class AtmosphericConstants:
    """Physical constants for troposphere modeling."""

    # Gas constants (J/kg/K)
    R_DRY = 287.0586       # Dry air
    R_VAPOR = 461.525      # Water vapor

    # Refractivity constants (K/mbar)
    K1 = 77.6890           # Dry term
    K2 = 71.2952           # Wet term 1
    K3 = 375463.0          # Wet term 2

    # Bevis et al. (1994) coefficients
    BEVIS_K2P = 22.1       # K2' constant
    BEVIS_K3 = 373900.0    # K3 constant

    # Standard atmosphere
    G = 9.80665            # Gravity (m/s^2)
    T0 = 288.15            # Standard temperature (K)
    P0 = 1013.25           # Standard pressure (hPa)
```
Listing 4.2: Atmospheric Constants for IWV Conversion

4.1.4 Horizontal Gradient Estimation

Tropospheric gradients capture azimuthal asymmetry:

$$T(\epsilon,\alpha) = \mathrm{ZHD}\cdot m_h(\epsilon) + \mathrm{ZWD}\cdot m_w(\epsilon) + m_g(\epsilon)[G_N\cos\alpha + G_E\sin\alpha] \tag{4.4}$$

The gradient mapping function:

$$m_g(\epsilon) = \frac{1}{\sin(\epsilon)\tan(\epsilon) + 0.0032} \tag{4.5}$$

4.1.5 ZTD to IWV Conversion

Integrated Water Vapor (IWV) is derived from ZTD using the Bevis formulation:

$$\mathrm{IWV} = \frac{\mathrm{ZWD}}{\Pi(T_m)} \tag{4.6}$$

where the dimensionless quantity Π depends on mean atmospheric temperature T_m:

$$\Pi(T_m) = 10^{-6}\cdot\left(k_2' + \frac{k_3}{T_m}\right)\cdot\frac{R_v}{R_d} \tag{4.7}$$

```python
def ztd_to_iwv(ztd: float, zhd: float, T_m: float) -> float:
    """
    Convert ZTD to IWV using Bevis et al. (1994) formulation.

    Args:
        ztd: Zenith Total Delay (meters)
        zhd: Zenith Hydrostatic Delay (meters)
        T_m: Mean atmospheric temperature (K)

    Returns:
        IWV in kg/m^2 (equivalent to mm of precipitable water)
    """
    # Derive ZWD
    zwd = ztd - zhd

    # Bevis conversion factor
    k2_prime = 22.1  # K/hPa
    k3 = 373900.0    # K^2/hPa
    Rv_Rd = 0.622    # R_dry / R_vapor

    # Pi factor (dimensionless)
    Pi = 1e-6 * (k2_prime + k3 / T_m) * Rv_Rd

    # IWV (kg/m^2)
    iwv = zwd / Pi

    return iwv
```

Listing 4.3: IWV Derivation Implementation

4.2 Bias Modeling

4.2.1 Observation Specific Biases (OSB)

CODE provides OSB products containing signal-specific biases for PPP-AR:

$$\phi_{\text{corrected}} = \phi_{\text{observed}} - \text{OSB}^s_\phi + \text{OSB}^\phi_r \tag{4.8}$$

OSB File Structure

```python
def download_bia(self, date: GNSSDate, provider: str = "CODE") -> DownloadResult
    :
    """
    Download CODE OSB/BIA file for PPP-AR.

    The BIA file contains:
    - Satellite phase biases (for integer ambiguity property)
    - Wide-lane biases
    - Narrow-lane biases
    """
    year = date.year
    doy = date.doy

    # CODE naming convention
    if provider == "CODE":
        filename = f"COD0OPSFIN_{year}{doy:03d}0000_01D_01D_OSB.BIA.gz"
        remote_path = f"/pub/products/mgex/{self._gps_week(date)}"

    # Download
    result = self._download_from_server("CODE", remote_path, filename)

    if not result.success:
        logger.warning("PPP-AR may not work without OSB/BIA file")

    return result
```
Listing 4.4: OSB/BIA Download Implementation

4.2.2 Code Biases

Differential Code Biases (DCB) relate different code observables:

$$\mathrm{DCB}_{P1-P2} = b_{P1} - b_{P2} \tag{4.9}$$

$$\mathrm{DCB}_{P1-C1} = b_{P1} - b_{C1} \tag{4.10}$$

These are critical for consistent ionosphere-free combinations.

4.2.3 Phase Biases for AR

For integer ambiguity resolution, phase biases must have the integer-cycle property:

$$b_\phi^s = \lambda \cdot \tilde{b}^s + \epsilon \tag{4.11}$$

where $\tilde{b}^s$ is the fractional cycle bias (FCB) and ϵ is a small residual. CODE products satisfy this property.

4.3 Antenna Calibration

4.3.1 Phase Center Corrections

Antenna Phase Center Offset (PCO) and Variations (PCV) are applied from ANTEX files:

$$\Delta\phi_{\text{ant}} = \mathrm{PCO}(\alpha, \epsilon) + \mathrm{PCV}(\alpha, \epsilon) \tag{4.12}$$

ANTEX Integration

```
# BSW configuration for antenna calibration
PHASECC: opt_PHASECC      # Path to phase center file
USE_ANTAZI: 0             # Azimuth-dependent corrections (0=off, 1=on)

# opt_PHASECC typically points to:
# - IGS20.ATX for current IGS products
# - IGS14.ATX for legacy compatibility
```
Listing 4.5: Antenna Calibration Application

4.3.2 Multi-GNSS Antenna Handling

Different GNSS systems require constellation-specific calibrations:

Table 4.1: Antenna Calibration by System

System	Frequencies	Calibration Status
GPS	L1, L2, L5	Full (IGS type-mean)
GLONASS	G1, G2, G3	Partial (offset from GPS)
Galileo	E1, E5a, E5b, E6	Growing database
BeiDou	B1I, B2I, B3I	Limited

4.4 Geophysical Corrections

4.4.1 Solid Earth Tides

Station coordinates vary due to tidal deformation:

$$\Delta \boldsymbol{r}_{\text{SET}} = \sum_{j=\text{Moon,Sun}} \frac{GM_j}{GM_\oplus} \frac{R_\oplus^4}{|\boldsymbol{r}_j|^3} \left[h_2 \hat{\boldsymbol{r}}_j (\hat{\boldsymbol{r}}_j \cdot \hat{\boldsymbol{R}}) - l_2 \hat{\boldsymbol{R}} |\hat{\boldsymbol{r}}_j \cdot \hat{\boldsymbol{R}}|^2 \right] \tag{4.13}$$

where $h_2 \approx 0.609$ and $l_2 \approx 0.085$ are Love/Shida numbers.

4.4.2 Ocean Tide Loading

Ocean tide loading (OTL) displacements are computed from harmonic coefficients:

$$\Delta \boldsymbol{r}_{\text{OTL}} = \sum_{k=1}^{11} A_k \cos(\chi_k - \phi_k) \tag{4.14}$$

OTL coefficients are station-specific, obtained from services like Onsala or Chalmers.

```
# BSW station file includes BLQ (ocean loading) reference
info_otl: ${P}/${CAMPAIGN}/STA/ocean_loading.BLQ

# BLQ format contains 11 tidal constituents:
# M2, S2, N2, K2, K1, O1, P1, Q1, Mf, Mm, Ssa
```

Listing 4.6: OTL File Reference

4.4.3 Pole Tide

Earth rotation irregularities cause pole tide displacements:

$$\Delta \boldsymbol{r}_{\text{pole}} = -\Omega^2 R_\oplus \frac{h_p}{g} (m_1 \sin 2\phi \cos \lambda + m_2 \sin 2\phi \sin \lambda) \tag{4.15}$$

where (m_1, m_2) are pole position offsets from ERP products.

Chapter 5

Ambiguity Resolution Module

5.1 Overview

Ambiguity Resolution (AR) is critical for achieving centimeter-level positioning accuracy. The system implements a complete AR pipeline:

1. Cycle slip detection and repair
2. Float ambiguity estimation
3. Integer ambiguity search (LAMBDA)
4. Validation and acceptance testing
5. Fixed solution computation

5.2 Cycle Slip Detection

5.2.1 Geometry-Free Combination

The geometry-free (GF) combination is sensitive to ionospheric variations and cycle slips:

$$L_{\mathrm{GF}} = L_1 - L_2 = \lambda_1 N_1 - \lambda_2 N_2 + I_1 \left(1 - \frac{f_1^2}{f_2^2}\right) + \text{biases} \tag{5.1}$$

A cycle slip on L_1 causes a jump of $\lambda_1 \approx 19$ cm in L_{GF}.

Algorithm 1 Geometry-Free Cycle Slip Detection

```
Require: Time series $L_{\mathrm{GF},k}$ for satellite $s$
Ensure: Detected slip epochs
1: Compute first difference: $\Delta L_{\mathrm{GF},k} = L_{\mathrm{GF},k} - L_{\mathrm{GF},k-1}$
2: Estimate ionospheric rate: $\dot{I}_k = \mathrm{median}(\Delta L_{\mathrm{GF}})$
3: Compute residual: $r_k = \Delta L_{\mathrm{GF},k} - \dot{I}_k$
4: Set threshold: $\tau = \max(4\sigma_r, 0.05\ \mathrm{m})$
5: if $|r_k| > \tau$ then
6:     Flag cycle slip at epoch $k$
7: end if
```

5.2.2 Melbourne-Wübbena Combination

The MW combination is geometry-free and ionosphere-free:

$$L_{\text{MW}} = L_{\text{WL}} - P_{\text{NL}} = \lambda_{\text{WL}} N_{\text{WL}} + \text{noise} \tag{5.2}$$

where:

$$L_{\text{WL}} = \frac{f_1 L_1 - f_2 L_2}{f_1 - f_2} \qquad \text{(Wide-lane phase)} \tag{5.3}$$

$$P_{\text{NL}} = \frac{f_1 P_1 + f_2 P_2}{f_1 + f_2} \qquad \text{(Narrow-lane code)} \tag{5.4}$$

The wide-lane wavelength $\lambda_{\text{WL}} \approx 86.2$ cm makes MW ideal for detecting cycle slips.

```
# BSW GPSEST options for MW processing
FREQUENCY: MELWUEBB      # Melbourne-Wubbena combination
USE_G: '1'               # GPS
USE_R: '1'               # GLONASS
USE_E: '1'               # Galileo
MINEL: '5'               # 5 degree elevation cutoff
SAMPLE: '300'            # 5-minute sampling
```

Listing 5.1: MW Cycle Slip Detection Configuration

5.3 LAMBDA Method

The Least-squares AMBiguity Decorrelation Adjustment (LAMBDA) is the standard integer least-squares approach.

5.3.1 Problem Formulation

Given float ambiguities $\hat{\boldsymbol{a}}$ with covariance $\mathbf{Q}_{\hat{a}}$, find:

$$\check{\boldsymbol{a}} = \arg\min_{\boldsymbol{a}\in\mathbb{Z}^n} (\hat{\boldsymbol{a}} - \boldsymbol{a})^{\mathsf{T}} \mathbf{Q}_{\hat{a}}^{-1} (\hat{\boldsymbol{a}} - \boldsymbol{a}) \tag{5.5}$$

5.3.2 Decorrelation

The key innovation of LAMBDA is the Z-transformation for decorrelation:

$$\boldsymbol{z} = \mathbf{Z}^{\mathsf{T}} \boldsymbol{a}, \quad \mathbf{Q}_{\hat{z}} = \mathbf{Z}^{\mathsf{T}} \mathbf{Q}_{\hat{a}} \mathbf{Z} \tag{5.6}$$

where $\mathbf{Z}$ is constructed via integer Gauss transformations to minimize off-diagonal correlations.

5.3.3 Search Algorithm

After decorrelation, the search is performed in the transformed space:

5.4 Validation Tests

5.4.1 Ratio Test

The most common validation is the ratio test:

Algorithm 2 LAMBDA Search Algorithm

```
Require: Float solution $\hat{z}$, decorrelated covariance $\mathbf{Q}_{\hat{z}}$
Ensure: Best and second-best integer candidates $\check{z}_1, \check{z}_2$
 1: Compute $\mathbf{LDL}^\mathsf{T}$ decomposition of $\mathbf{Q}_{\hat{z}}$
 2: Initialize search ellipsoid with $\chi^2$ threshold
 3: for $z_n$ in valid range do
 4:     Update conditional bounds for $z_{n-1}, \ldots, z_1$
 5:     if candidate inside ellipsoid then
 6:         Evaluate cost function
 7:         Update best/second-best if improved
 8:     end if
 9: end for
10: Transform back: $\check{a} = \mathbf{Z}^{-\mathsf{T}}\check{z}$
```

$$R = \frac{\|\hat{\boldsymbol{a}} - \check{\boldsymbol{a}}_2\|^2_{\mathbf{Q}_{\hat{a}}^{-1}}}{\|\hat{\boldsymbol{a}} - \check{\boldsymbol{a}}_1\|^2_{\mathbf{Q}_{\hat{a}}^{-1}}} \tag{5.7}$$

where $\check{\boldsymbol{a}}_1$ and $\check{\boldsymbol{a}}_2$ are the best and second-best candidates.

$$\text{Accept if } R > R_{\text{threshold}} \tag{5.8}$$

Typical thresholds: $R_{\text{threshold}} \in [2.0, 3.0]$

5.4.2 Success Rate Estimation

The theoretical success rate can be bounded:

$$P_s \geq \prod_{i=1}^{n} \left(2\Phi\left(\frac{1}{2\sigma_i}\right) - 1 \right) \tag{5.9}$$

where σ_i are the decorrelated ambiguity standard deviations.

```
# BSW AR validation parameters
AMBRES: SIGMA           # Sigma-based resolution
AMBWGT: 0.001           # Ambiguity constraint weight (cycles)
RATIO_THRESHOLD: 2.5    # Minimum ratio for acceptance

# Sigma thresholds (cycles)
WL_SIGMA_MAX: 0.15      # Wide-lane threshold
NL_SIGMA_MAX: 0.10      # Narrow-lane threshold
```

Listing 5.2: AR Validation Configuration

5.5 Partial Ambiguity Resolution

When full AR fails, partial AR fixes a subset of ambiguities:

5.6 Multi-GNSS Ambiguity Resolution

5.6.1 Inter-System Bias Handling

For multi-GNSS AR, inter-system biases (ISB) must be considered:

$$\phi_E = \rho + c\delta t_r + \text{ISB}_{G-E} + \lambda_E N_E + \ldots \tag{5.10}$$

Algorithm 3 Partial Ambiguity Resolution

```
Require: Float ambiguities $\hat{\boldsymbol{a}}$, covariance $\mathbf{Q}_{\hat{a}}$
Ensure: Partially fixed solution
 1: Attempt full AR with LAMBDA
 2: if ratio test fails then
 3:     Sort ambiguities by $\sigma_i$ (ascending)
 4:     for $k = n - 1$ down to $n_{\min}$ do
 5:         Select $k$ best-determined ambiguities
 6:         Attempt AR on subset
 7:         if ratio test passes then
 8:             Fix subset, propagate to remaining floats
 9:             return partial fixed solution
10:         end if
11:     end for
12: else
13:     return fully fixed solution
14: end if
```

5.6.2 GRE Configuration

The system supports GPS+GLONASS+Galileo (GRE) processing:

```
# From bsw_configs/iGNSS_D_PPP_AR_IG_IGS54_direct.yaml
# Critical AR steps with GRE enabled:

# Receiver clock synchronization
TITLE: 'PPP_$YYYSS+0_$(FFFF): Receiver clock synchronization'
USE_G: '1'    # GPS
USE_R: '1'    # GLONASS
USE_E: '1'    # Galileo
USE_C: '0'    # BeiDou (disabled - complex biases)
USE_J: '0'    # QZSS (disabled)

# Melbourne-Wubbena WL estimation
TITLE: SCRIPT
FREQUENCY: MELWUEBB
USE_G: '1'
USE_R: '1'
USE_E: '1'

# Ambiguity-fixed solution
TITLE: 'PPP_$YYYSS+0_$(FFFF): COMPUTATION OF AMBIGUITY-FIXED SOLUTION'
USE_G: '1'
USE_R: '1'
USE_E: '1'
FREQUENCY: L3
```

Listing 5.3: Multi-GNSS AR Configuration

5.6.3 System-Specific Considerations

5.7 AR Performance Metrics

Key metrics for AR quality assessment:

- **WL Fix Rate**: Percentage of wide-lane ambiguities fixed
- **NL Fix Rate**: Percentage of narrow-lane ambiguities fixed

Table 5.1: Multi-GNSS AR Characteristics

System	**Modulation**	**AR Support**	**Notes**
GPS	CDMA	Full	Reference system
GLONASS	FDMA	Full	IFB estimation required
Galileo	CDMA	Full	Excellent signal quality
BeiDou	CDMA/mixed	Partial	GEO vs MEO complexity

- **Overall Fix Rate**: Combined WL and NL success
- **Ratio Value**: Discrimination between candidates

Target performance with CODE products and GRE:

- WL Fix Rate: $> 85\%$
- NL Fix Rate: $> 75\%$
- Position accuracy: < 2 cm (horizontal), < 5 cm (vertical)

Chapter 6

Python Implementation Details

6.1 Code Architecture

6.1.1 Class Hierarchy

The system follows object-oriented design principles:

```python
# Station management
@dataclass
class Station:
    """GNSS reference station."""
    id: str                         # 4-character ID
    name: str                       # Full name
    domes: str                      # DOMES number
    coordinates: Coordinates        # ITRF coordinates
    antenna: AntennaInfo            # Antenna type and radome
    receiver: ReceiverInfo          # Receiver information
    networks: List[str]             # Network memberships

    def is_multi_gnss(self) -> bool:
        """Check if station tracks multiple GNSS."""
        return self.receiver.supports_multi_gnss()

# Date handling
@dataclass
class GNSSDate:
    """GNSS date with DOY and GPS week support."""
    year: int
    month: int
    day: int
    hour: int = 0
    minute: int = 0
    second: float = 0.0

    @property
    def doy(self) -> int:
        """Day of year."""
        return self.to_datetime().timetuple().tm_yday

    @property
    def gps_week(self) -> int:
        """GPS week number."""
        return (self.mjd - 44244) // 7

    @property
    def mjd(self) -> float:
        """Modified Julian Date."""
        return self._compute_mjd()
```

```python

# Processing configuration
@dataclass
class ProcessingConfig:
    """Main processing configuration."""
    proc_type: ProcessingType
    gnss_date: GNSSDate
    campaign_name: str
    session_id: str

    # Products
    orbit: ProductConfig
    clock: ProductConfig
    erp: ProductConfig
    bia: ProductConfig
    vmf3: ProductConfig

    # Paths
    data_dir: Path
    bsw_campaign_dir: Path
    pcf_file: Path
```

Listing 6.1: Core Class Structure

6.1.2 Processing Pipeline

```python
class DailyPPPProcessor:
    """Daily PPP-AR processing pipeline."""

    def __init__(self, config: ProcessingConfig):
        self.config = config
        self.downloader = ProductDownloader()
        self.station_downloader = StationDownloader()
        self.bpe_runner = BPERunner()

    def process(self, date: GNSSDate,
                stations: List[Station]) -> ProcessingResult:
        """Execute complete processing pipeline."""
        result = ProcessingResult(gnss_date=date)

        try:
            # 1. Download products
            products = self._download_products(date)
            result.products_downloaded = products

            # 2. Download station data
            station_data = self._download_stations(stations, date)
            result.stations_available = len(station_data)

            # 3. Setup campaign
            campaign_dir = self._setup_campaign(date, products, station_data)

            # 4. Run BSW/BPE
            bpe_result = self.bpe_runner.run(
                campaign_dir=campaign_dir,
                pcf_file=self.config.pcf_file,
                session=self._get_session_name(date)
            )

            # 5. Post-process results
            if bpe_result.success:
                self._extract_results(campaign_dir, result)
                result.success = True
            else:
                result.error_message = bpe_result.error

        except Exception as e:
            result.error_message = str(e)
            logger.exception("Processing failed")

        return result
```

Listing 6.2: Main Processing Pipeline

6.2 NumPy/SciPy Optimization

6.2.1 Matrix Operations

For coordinate transformations and covariance propagation:

```python
import numpy as np
from scipy.spatial.transform import Rotation

def ecef_to_enu(ecef_coords: np.ndarray,
                ref_lat: float,
                ref_lon: float) -> np.ndarray:
    """
    Transform ECEF coordinates to local ENU frame.

    Uses vectorized NumPy operations for efficiency.

    Args:
        ecef_coords: Nx3 array of ECEF coordinates
        ref_lat: Reference latitude (radians)
        ref_lon: Reference longitude (radians)

    Returns:
        Nx3 array of ENU coordinates
    """
    # Rotation matrix (ECEF -> ENU)
    sin_lat, cos_lat = np.sin(ref_lat), np.cos(ref_lat)
    sin_lon, cos_lon = np.sin(ref_lon), np.cos(ref_lon)

    R = np.array([
        [-sin_lon,               cos_lon,              0],
        [-sin_lat * cos_lon,   -sin_lat * sin_lon,   cos_lat],
        [cos_lat * cos_lon,     cos_lat * sin_lon,   sin_lat]
    ])

    # Vectorized transformation
    return ecef_coords @ R.T


def propagate_covariance(P: np.ndarray,
                         Phi: np.ndarray,
                         Q: np.ndarray) -> np.ndarray:
    """
    Propagate state covariance through transition.

    P_new = Phi @ P @ Phi.T + Q

    Uses optimized BLAS operations via NumPy.
    """
    return Phi @ P @ Phi.T + Q
```

Listing 6.3: Optimized Coordinate Transformations

6.2.2 Statistical Computations

```python
import numpy as np
from scipy import stats

def robust_mean(data: np.ndarray,
                sigma_threshold: float = 3.0,
                max_iterations: int = 10) -> Tuple[float, float, int]:
    """
    Compute robust mean with iterative outlier rejection.

    Used for coordinate averaging in NRT processing.

    Args:
        data: Input array
        sigma_threshold: Rejection threshold in sigma
        max_iterations: Maximum iterations

    Returns:
        (mean, std, n_valid) after outlier rejection
    """
    mask = np.ones(len(data), dtype=bool)

    for _ in range(max_iterations):
        valid_data = data[mask]
        if len(valid_data) < 3:
            break

        mean = np.median(valid_data)  # Robust initial estimate
        std = stats.median_abs_deviation(valid_data, scale='normal')

        # Update mask
        new_mask = np.abs(data - mean) < sigma_threshold * std

        if np.array_equal(mask, new_mask):
            break
        mask = new_mask

    final_data = data[mask]
    return np.mean(final_data), np.std(final_data), len(final_data)
```

Listing 6.4: Robust Statistics for Outlier Detection

6.3 Configuration Management

6.3.1 YAML Configuration

```python
from dataclasses import dataclass, field
from pathlib import Path
import yaml

@dataclass
class PathConfig:
    """Centralized path configuration (Singleton pattern)."""

    _instance: ClassVar[Optional['PathConfig']] = None

    # Base directories
    bern54_dir: Path = field(default_factory=lambda: Path("/opt/BERN54"))
    gpsuser_dir: Path = field(default_factory=lambda: Path.home() / "GPSUSER54")
    data_root: Path = field(default_factory=lambda: Path("/data/gnss"))

    # Derived paths
    @property
    def campaign_dir(self) -> Path:
        return self.data_root / "CAMPAIGN54"

    @property
    def datapool_dir(self) -> Path:
        return self.data_root / "DATAPOOL"

    @classmethod
    def get_instance(cls) -> 'PathConfig':
        """Get singleton instance."""
        if cls._instance is None:
            cls._instance = cls._load_from_environment()
        return cls._instance

    @classmethod
    def _load_from_environment(cls) -> 'PathConfig':
        """Load paths from environment variables."""
        return cls(
            bern54_dir=Path(os.environ.get('BERN54_DIR', '/opt/BERN54')),
            gpsuser_dir=Path(os.environ.get('GPSUSER_DIR',
                                            str(Path.home() / 'GPSUSER54'))),
            data_root=Path(os.environ.get('DATA_ROOT', '/data/gnss'))
        )


def load_network_config(network_id: str) -> NetworkProfile:
    """Load network-specific configuration."""
    config_path = Path(__file__).parent / "networks.yaml"

    with open(config_path) as f:
        configs = yaml.safe_load(f)

    if network_id not in configs:
        raise ValueError(f"Invalid network ID: {network_id}")

    return NetworkProfile(**configs[network_id])
```

Listing 6.5: Configuration Loading System

6.4 BSW Integration

6.4.1 BPE Runner

```python
import subprocess
from pathlib import Path
import tempfile

class BPERunner:
    """Execute Bernese Processing Engine."""

    def __init__(self, bern_dir: Path, timeout: int = 7200):
        self.bern_dir = bern_dir
        self.timeout = timeout  # 2 hours default
        self.loadgps = bern_dir / "GPS" / "EXE" / "LOADGPS.setvar"

    def run(self, campaign_dir: Path,
            pcf_file: str,
            session: str,
            cpu_file: str = "USER") -> BPEResult:
        """
        Execute BPE processing.

        Args:
            campaign_dir: Path to campaign directory
            pcf_file: PCF filename (e.g., PPP54IGS.PCF)
            session: Session identifier (e.g., 25358IG)
            cpu_file: CPU control file

        Returns:
            BPEResult with success status and outputs
        """
        # Create temporary user area
        with tempfile.TemporaryDirectory() as tmp_user:
            # Setup environment
            env = self._setup_environment(campaign_dir, tmp_user)

            # Build command
            cmd = [
                str(self.bern_dir / "GPS" / "EXE" / "menu.sh"),
                "-c", str(campaign_dir),
                "-s", session,
                "-y", session[:2],
                "-p", pcf_file,
                "-u", cpu_file
            ]

            try:
                result = subprocess.run(
                    cmd,
                    env=env,
                    cwd=campaign_dir,
                    capture_output=True,
                    text=True,
                    timeout=self.timeout
                )

                return BPEResult(
                    success=(result.returncode == 0),
                    stdout=result.stdout,
                    stderr=result.stderr
                )

            except subprocess.TimeoutExpired:
                return BPEResult(
                    success=False,
                    error="BPE timeout exceeded"
                )

    def _setup_environment(self, campaign_dir: Path,
                           user_dir: str) -> Dict[str, str]:
        """Setup BSW environment variables."""
        env = os.environ.copy()

        # Source LOADGPS.setvar equivalent
        env['C'] = str(campaign_dir)
        env['U'] = user_dir
        env['P'] = str(campaign_dir.parent)
        env['BERN54'] = str(self.bern_dir)

        return env
```

Listing 6.6: Bernese Processing Engine Integration

6.5 Logging and Monitoring

```python
import structlog
from typing import Any

def configure_logging(verbose: bool = False) -> None:
    """Configure structured logging."""

    processors = [
        structlog.stdlib.filter_by_level,
        structlog.stdlib.add_logger_name,
        structlog.stdlib.add_log_level,
        structlog.processors.TimeStamper(fmt="iso"),
        structlog.processors.StackInfoRenderer(),
        structlog.processors.format_exc_info,
    ]

    if verbose:
        processors.append(structlog.dev.ConsoleRenderer())
    else:
        processors.append(structlog.processors.JSONRenderer())

    structlog.configure(
        processors=processors,
        wrapper_class=structlog.stdlib.BoundLogger,
        context_class=dict,
        logger_factory=structlog.stdlib.LoggerFactory(),
        cache_logger_on_first_use=True,
    )


# Usage example
logger = structlog.get_logger()

def process_epoch(date: GNSSDate, network: str) -> None:
    logger.info(
        "processing_started",
        date=str(date),
        network=network,
        doy=date.doy
    )

    # ... processing ...

    logger.info(
        "processing_completed",
        date=str(date),
        network=network,
        stations_processed=42,
        ar_success_rate=0.85
    )
```

Listing 6.7: Structured Logging Configuration

Chapter 7

Operational Deployment

7.1 CLI Interface

```python
import click
from datetime import date

@click.group()
def cli():
    """PyGNSS-RT: Real-Time GNSS Processing System."""
    pass

@cli.command()
@click.argument('network', type=click.Choice(['IG', 'EU', 'GB', 'RG', 'SS']))
@click.option('-s', '--start-date', type=click.DateTime(), required=True)
@click.option('-e', '--end-date', type=click.DateTime())
@click.option('--cron', is_flag=True, help='Run in cron mode with latency')
@click.option('-v', '--verbose', is_flag=True)
def daily_ppp(network: str, start_date: date, end_date: date,
              cron: bool, verbose: bool):
    """Run daily PPP-AR processing."""

    config = load_config(network)
    processor = DailyPPPProcessor(config)

    for date in date_range(start_date, end_date):
        if cron and not is_ready(date, config.latency):
            continue

        result = processor.process(date)

        if verbose:
            print(f"Date: {date}")
            print(f"  Stations: {result.stations_processed}")
            print(f"  AR Rate: {result.ar_success_rate:.1%}")

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

Listing 7.1: Command Line Interface

7.2 Cron Scheduling

```
# /etc/cron.d/pygnss_rt

# Hourly processing (3-hour latency)
```

```
0 * * * * gnss python3 -m pygnss_rt.cli hourly-ppp IG --cron

# Daily processing (21-day latency)
30 6 * * * gnss python3 -m pygnss_rt.cli daily-ppp IG --cron

# NRT coordinate generation
0 8 * * * gnss python3 -m pygnss_rt.cli daily-crd IG
```
Listing 7.2: Cron Configuration Example

7.3 Performance Metrics

Table 7.1: Typical Processing Performance

Mode	**Stations**	**Runtime**	**AR Rate**
Daily (IG)	200	45 min	85%
Daily (EU)	150	35 min	82%
Hourly	50	8 min	78%

Appendix A

Configuration Reference

A.1 Network Profiles

```python
# From processing/networks.py
NETWORK_PROFILES = {
    "IG": NetworkProfile(
        network_id=NetworkID.IG,
        description="IGS core stations (global reference)",
        session_id="IG",
        requires_igs_alignment=False,
        orbit_source=ProductSource(provider="CODE", tier="final"),
        clock_source=ProductSource(provider="CODE", tier="final"),
        erp_source=ProductSource(provider="CODE", tier="final"),
        bia_source=ProductSource(provider="CODE", tier="final"),
    ),
    "EU": NetworkProfile(
        network_id=NetworkID.EU,
        description="EUREF European network",
        session_id="EU",
        requires_igs_alignment=True,
        # ... similar configuration
    ),
    # ... GB, RG, SS profiles
}
```

Listing A.1: Network Profile Definition

Appendix B

Mathematical Notation

Table B.1: Symbol Reference

Symbol	**Description**
$\boldsymbol{x}$	State vector
$\mathbf{P}$	State covariance matrix
$\mathbf{\Phi}$	State transition matrix
$\mathbf{Q}$	Process noise covariance
$\mathbf{H}$	Design (observation) matrix
$\mathbf{R}$	Measurement noise covariance
$\mathbf{K}$	Kalman gain
ρ	Geometric range
δt	Clock offset
T	Tropospheric delay
I	Ionospheric delay
N	Carrier phase ambiguity
λ	Carrier wavelength
ϵ	Elevation angle
α	Azimuth angle

Bibliography

[1] Teunissen, P.J.G. and Montenbruck, O. (2017). *Springer Handbook of Global Navigation Satellite Systems*. Springer International Publishing.

[2] Böhm, J., Niell, A., Tregoning, P., and Schuh, H. (2006). Global Mapping Function (GMF): A new empirical mapping function based on numerical weather model data. *Geophysical Research Letters*, 33(7).

[3] Landskron, D. and Böhm, J. (2018). VMF3/GPT3: Refined discrete and empirical troposphere mapping functions. *Journal of Geodesy*, 92(4), 349–360.

[4] Teunissen, P.J.G. (1995). The least-squares ambiguity decorrelation adjustment: a method for fast GPS integer ambiguity estimation. *Journal of Geodesy*, 70(1-2), 65–82.

[5] Bevis, M., Businger, S., Herring, T.A., et al. (1992). GPS meteorology: Remote sensing of atmospheric water vapor using the Global Positioning System. *Journal of Geophysical Research*, 97(D14), 15787–15801.

[6] Dach, R., Lutz, S., Walser, P., and Fridez, P. (2015). *Bernese GNSS Software Version 5.2*. Astronomical Institute, University of Bern.

[7] Kouba, J. (2009). A guide to using International GNSS Service (IGS) products. *IGS Publications*.