wavpy v2.0: User manual

APPROVAL LIST		
NAME	SIGNATURE	DATE
Prepared by: Fran Fabra		30/10/2019
Revised by: –		–
Approved by: –		–
Authorised by: –		–

Table of Contents

Document Approval List 1
Table of Contents 7

1 Introduction **8**

2 GNSS_composite **9**
2.1 Public variables 9
2.2 Relevant private variables 10
2.3 Functions 10
2.3.1 dump_parameters 10
2.3.2 set_instrumental_params 10
2.3.3 compute_lambda_func 11
2.3.4 get_lambda_func 11
2.3.5 set_lambda_func 12

3 RF_FrontEnd **13**
3.1 Additional information 13
3.2 Relevant private variables 14
3.3 Functions 15
3.3.1 dump_parameters 15
3.3.2 set_antenna_orientation_BF_EH 16
3.3.3 set_antenna_orientation_BF_k 16
3.3.4 get_antenna_orientation_BF 16
3.3.5 set_antenna_whole_pattern 17
3.3.6 set_val_antenna_pattern 17
3.3.7 set_antenna_pattern_FF 18
3.3.8 set_antenna_pattern_FH 18
3.3.9 set_antenna_pattern_interp 18
3.3.10 set_antenna_patterns_FF 19
3.3.11 set_antenna_patterns_FH 19
3.3.12 set_antenna_patterns_interp 20
3.3.13 get_antenna_whole_pattern 20
3.3.14 get_antenna_patterns 21

3.3.15 set_receiver_params 21
3.3.16 set_antenna_eff_area 22
3.3.17 set_noise_T 22
3.3.18 set_noise_pow_dBW 22
3.3.19 set_frequency 22
3.3.20 get_PhiTheta_gain_dB 23
3.3.21 get_incvector_gain_dB 23
3.3.22 get_frequency 23
3.3.23 get_antenna_Gain_dB 24
3.3.24 get_antenna_Aeff 24
3.3.25 get_antenna_T 24
3.3.26 get_noise_T 24
3.3.27 get_noise_pow_dBW 25
3.3.28 get_noise_F_dB 25
3.3.29 get_filter_BB_BW 25
3.3.30 set_antenna_elements_pos_AF 26
3.3.31 set_phase_delays 26
3.3.32 get_phase_delays 26
3.3.33 compute_array_factor 27
3.3.34 get_array_factor 27
3.3.35 compute_phase_delays_UPA 27
3.3.36 compute_phase_delays_pos_ECEF_RT 28

4 Specular_geometry **29**
4.1 Additional information 29
4.2 Public variables 30
4.3 Relevant private variables 30
4.4 Functions 31
4.4.1 dump_parameters 31
4.4.2 set_ECEFpos_Receiver 32
4.4.3 get_ECEFpos_Receiver 32
4.4.4 set_ECEFvel_Receiver 32
4.4.5 get_ECEFvel_Receiver 33
4.4.6 set_ECEFpos_Transmitter 33
4.4.7 get_ECEFpos_Transmitter 33
4.4.8 set_ECEFvel_Transmitter 33
4.4.9 get_ECEFvel_Transmitter 34
4.4.10 get_ECEFpos_Specular 34
4.4.11 set_LongLatHeight_Receiver 34
4.4.12 get_LongLatHeight_Receiver 35

4.4.13 set_LongLatHeight_Transmitter . . . 35
4.4.14 get_LongLatHeight_Transmitter . . . 36
4.4.15 set_geometry_from_ElevHeightsSpec . . . 36
4.4.16 set_tangEarthVel_Receiver . . . 37
4.4.17 set_tangEarthVel_Transmitter . . . 37
4.4.18 set_Undulation . . . 38
4.4.19 get_Undulation . . . 38
4.4.20 read_ECEFpos_Receiver . . . 38
4.4.21 read_ECEFpos_Transmitter . . . 39
4.4.22 read_ECEFpos_GNSS_Transmitter . . . 39
4.4.23 load_sp3File . . . 40
4.4.24 free_sp3File . . . 40
4.4.25 read_ECEFpos_GNSS_Transmitter_sp3Loaded . . . 40
4.4.26 compute_specular_point . . . 41
4.4.27 compute_specular_point_Undu_Spherical_Earth . . . 41
4.4.28 compute_ElevAzimT_from_receiver . . . 41
4.4.29 set_inertials . . . 42
4.4.30 get_inertials . . . 42
4.4.31 rotate_vector_BF_to_local . . . 43
4.4.32 rotate_vector_BF_to_ECEF . . . 43
4.4.33 compute_inertial_delay . . . 44
4.4.34 read_Inertials_Receiver . . . 44
4.4.35 compute_Beyerle_windup_direct . . . 44
4.4.36 compute_Beyerle_windup_reflected . . . 45

5 Reflecting_surface 47
5.1 Public variables . . . 47
5.2 Relevant private variables . . . 48
5.3 Functions . . . 49
5.3.1 dump_parameters . . . 49
5.3.2 set_frequency . . . 49
5.3.3 set_k_threshold . . . 50
5.3.4 set_k_threshold_Brown . . . 50
5.3.5 get_k_threshold . . . 50
5.3.6 epsilon_sea_water . . . 50
5.3.7 epsilon_sea_ice . . . 51
5.3.8 epsilon_dry_snow . . . 51
5.3.9 epsilon_wet_snow . . . 51
5.3.10 compute_Rfresnel_linear . . . 52
5.3.11 compute_Rfresnel_circular . . . 52

5.3.12 compute_Tfresnel_linear 53
5.3.13 compute_Tfresnel_circular 53
5.3.14 compute_sea_spectrum 54
5.3.15 set_surf_spectrum 54
5.3.16 set_surf_spectrum_omnidir 55
5.3.17 get_surf_spectrum 55
5.3.18 get_surf_spectrum_omnidir 56
5.3.19 compute_mss_from_spectrum 56
5.3.20 compute_mss_from_wind 56
5.3.21 set_wind_grid 57
5.3.22 interp_wind_grid 57
5.3.23 disable_wind_grid 57
5.3.24 get_wind_grid_status 58

6 ZaVoModel_GNSSR **59**
6.1 Additional information 59
6.2 Public variables 60
6.3 Relevant private variables 61
6.4 Functions 62
6.4.1 enable_isolines_data_dump 62
6.4.2 disable_isolines_data_dump 64
6.4.3 set_stare_processing_mode 64
6.4.4 unset_stare_processing_mode 64
6.4.5 set_transmitter_EIRP 65
6.4.6 unset_transmitter_EIRP 65
6.4.7 compute_waveform 65
6.4.8 get_DDM_doppler_slice 65
6.4.9 get_cov_slice 66
6.4.10 get_noisy_waveform 66
6.4.11 get_noisy_DDM 67

7 MRSR_Model **68**
7.1 Additional information 68
7.2 Public variables 69
7.3 Relevant private variables 69
7.4 Functions 70
7.4.1 set_general_scenario 70
7.4.2 set_planar_layers_scenario 70
7.4.3 set_dry_snow_planar_layers_scenario 71
7.4.4 mod_height_depths 71

7.4.5 mod_alphas . . . 72
7.4.6 mod_epsilon . . . 72
7.4.7 compute_GNSS_wavcluster . . . 73
7.4.8 compute_LH_freqs_and_depths . . . 73
7.4.9 compute_pow_linearPol . . . 74

8 Waveform_power 76
8.1 Public variables . . . 76
8.2 Relevant private variables . . . 78
8.3 Functions . . . 78
8.3.1 set_waveform . . . 78
8.3.2 set_float_waveform . . . 79
8.3.3 set_norm_waveform . . . 79
8.3.4 set_amp_waveform . . . 79
8.3.5 get_waveform . . . 80
8.3.6 add_waveform_retracking . . . 80
8.3.7 set_sampling_rate . . . 81
8.3.8 set_rel_factor . . . 81
8.3.9 get_rel_factor . . . 81
8.3.10 set_min_resolution_fft_interp . . . 82
8.3.11 set_fit_length . . . 82
8.3.12 set_normtail_length . . . 83
8.3.13 set_tail_factor . . . 83
8.3.14 set_noise_lags . . . 83
8.3.15 set_tail_lags . . . 84
8.3.16 compute_delays . . . 84
8.3.17 compute_delays_wspeckle . . . 84
8.3.18 compute_delays_wlimits . . . 84
8.3.19 compute_delays_wlimits_LPF . . . 85
8.3.20 set_init_range . . . 86
8.3.21 get_range_waveform . . . 86
8.3.22 get_size_deriv_waveform_tracks . . . 87
8.3.23 get_deriv_waveform_tracks . . . 87
8.3.24 get_deriv_waveform_tracks_wlimits . . . 88
8.3.25 dump_norm_waveform . . . 89
8.3.26 dump_delays . . . 89
8.3.27 get_wav_length . . . 90

9 Waveform_complex_cluster 91
9.1 Public variables . . . 91

9.2 Relevant private variables 91
9.3 Functions 92
9.3.1 initialize 92
9.3.2 add_waveform 92
9.3.3 add_waveform_scale 93
9.3.4 add_waveform_GOLD 93
9.3.5 add_waveform_PIR 94
9.3.6 load_ITF_waveforms_SPIR 95
9.3.7 load_ITF_waveforms_SPIR_selected_signals 95
9.3.8 load_CR_waveforms_SPIR 96
9.3.9 load_CR_waveforms_SPIR_selected_signals 97
9.3.10 searching_CR_waveforms_SPIR 98
9.3.11 get_waveform 99
9.3.12 integrate_waveforms 99
9.3.13 integrate_waveforms_remdir 100
9.3.14 integrate_waveforms_retracking 101
9.3.15 dump_phase 101
9.3.16 dump_phase_peak 102
9.3.17 store_phasor_wavs 102
9.3.18 get_phasor 102
9.3.19 get_sigma_phase_phasor 103
9.3.20 get_sigma_phase_phasor_interv 103
9.3.21 counterrot_phasor 104
9.3.22 counterrot_waveforms 104
9.3.23 correct_navigation_bit 105
9.3.24 compute_coherence_time 105
9.3.25 compute_singlefreq_DDM 106
9.3.26 compute_singlefreq_DDM_remdir 107
9.3.27 compute_singlelag_DDM 107
9.3.28 compute_singlelag_DDM_remdir 108
9.3.29 compute_DopplerMap_BW 109
9.3.30 compute_DopplerMap_BW_remdir 110
9.3.31 compute_LagHologram 110
9.3.32 get_wav_length 111
9.3.33 get_cluster_length 111

References **112**

1 Introduction

Rather than just a simple GNSS+R simulator, **wavpy** is a set of *C++* classes that characterize a GNSS+R scenario. Working from a high level programming language like *python*, which is a user-friendly environment, a user in the GNSS+R field has a valid tool for working at a different levels in the processing chain, ranging from the characterization of certain parameters, such as the evolution of the reflectivity for a given surface as a function of the elevation angle, to a whole waveform/DDM simulation case or even analysis of real GNSS+R data.

During the next chapters, this manual provides the definition of each variable and function from each of the different classes available in wavpy v1.0, as well as examples of how to work with them for a *python* user. In particular:

- **GNSS_composite** class: characterization of a GNSS signal. Chapter 2.
- **RF_FrontEnd** class: description of the different parameters in radio-frequency front-end. Chapter 3.
- **Specular_geometry** class: characterization of position and velocities in a GNSS+R scenario. Chapter 4.
- **Reflecting_surface** class: characterization of a reflecting surface. Chapter 5.
- **ZaVoModel_GNSSR** class: dedicated to waveform and DDM modelling. Chapter 6.
- **MRSR_Model** class: dedicated to waveform modelling under a multiple layer scenario. Chapter 7.
- **Waveform_power** class: dedicated to analysis of GNSS+R power waveforms. Chapter 8.
- **Waveform_complex_cluster** class: dedicated to analysis of time series of GNSS+R complex waveforms. Chapter 9.

2 GNSS_composite

This class generates the autocorrelation function of a set of GNSS signals.

Example of object construction with arbitrary name *my_GNSS_func*:

```
my_GNSS_func = wavpy.GNSS_composite()
```

In this case, public variables and functions from object *my_GNSS_func* of class *GNSS_composite* can be respectively checked/modified or called with:

```
my_GNSS_func.variable
```

```
my_GNSS_func.function()
```

2.1 Public variables

- **lambda_size** (integer): Number of samples of the autocorrelation function. Default value: 157
- **weight_CA** (float): Weight for the C/A code (GPS L1). Default value: 1.0
- **weight_PY** (float): Weight for the PY code (GPS L1). Default value: 1.0
- **weight_M** (float): Weight for the M code (GPS L1). Default value: 1.0
- **weight_IM** (float): Weight for the Inter-Modulation component (GPS L1). Default value: 1.0
- **weight_E1A** (float): Weight for the A code (Galileo E1). Default value: 0.0
- **weight_E1B** (float): Weight for the B code (Galileo E1). Default value: 0.0

- **weight_E1C** (float): Weight for the C code (Galileo E1). Default value: 0.0
- **weight_B1I** (float): Weight for the C code (BeiDou B1). Default value: 0.0
- **weight_L1C** (float): Weight for the L1C code (QZSS). Default value: 0.0
- **frequency** (double): Frequency of the GNSS signal. Default value: 1575420000.0 (GPS L1)

2.2 Relevant private variables

- **lambda_func** (double, array of **lambda_size** elements): Normalized autocorrelation function stored. Default value: GPS-L1 autocorrelation function with all code components.
- **sampling_rate** (double): Sampling rate of the receiver in samples/sec. Default value: 80000000.0
- **filter_BB_BW** (double): Base-band bandwidth of the filter applied in Hz. Default value: 12000000.0

2.3 Functions

2.3.1 dump_parameters

Print relevant information about the object's content (public and private variables).

Example:

```
my_GNSS_func.dump_parameters()
```

2.3.2 set_instrumental_params

Set the instrumental parameters that have to be taken into account for the computation of the autocorrelation function.

Example:

```
my_GNSS_func.set_instrumental_params(sampling_rate, filter_BW, computeLambda)
```

Input variables:

- **sampling_rate** (double): Input sampling rate in samples/sec.
- **filter_BW** (double): Input base-band bandwidth of the filter applied in Hz.
- **computeLambda** (char): "1" if you want to compute the autocorrelation function with this function call, or "0" if you just want to set the input parameters without computing the autocorrelation function.

2.3.3 compute_lambda_func

Compute and store the autocorrelation function according to current configuration.

Example:

```
my_GNSS_func.compute_lambda_func()
```

2.3.4 get_lambda_func

Provide the range and values of the autocorrelation function internally stored.

Example:

```
[range_lambda, lambda_func] = my_GNSS_func.get_lambda_func(range_len,
lambda_len)
```

Input variables:

- **range_len** (integer): Number of samples of the range of the autocorrelation function (it has to be equal to **lambda_size**).
- **lambda_len** (integer): Number of samples of the autocorrelation function (it has to be equal to **lambda_size**).

Output variables:

- **range_lambda** (double, array of size **range_len**): Range of the autocorrelation function in meters.
- **lambda_func** (double, array of size **lambda_len**): Normalized autocorrelation function.

2.3.5 set_lambda_func

Set a given autocorrelation function.

Example:

```
my_GNSS_func.set_lambda_func(lambda_func)
```

Input variables:

- **lambda_func** (double, array of N elements): Normalized autocorrelation function.

3 RF_FrontEnd

This class characterizes the main aspects of a receiver front end.

Example of object construction with arbitrary name *my_receiver*:

```
my_receiver = wavpy.RF_FrontEnd()
```

In this case, public variables and functions from object *my_receiver* of class *RF_FrontEnd* can be respectively checked/modified or called with:

```
my_receiver.variable
```

```
my_receiver.function()
```

3.1 Additional information

Receiver frame: The receiver body frame is a Cartesian coordinates system of the structure containing the receiver, typically a satellite or an aircraft, where the X-axis points towards the front, the Y-axis points towards the right-side (XY define the horizontal plane) and the Z-axis towards Nadir. In absence of inertial rotation of the body frame, we will assume that the X-axis points towards the Earth's North and the Z-axis points towards the Earth's center.

Antenna frame: the antenna frame is a Cartesian coordinates system with its origin at the antenna's physical center, X- and Y-axis defining the physical plane of the antenna and the Z-axis pointing perpendicularly towards the propagation direction by complying the right-hand rule. Typically, this system is also represented with spherical coordinates, using radial distance, polar angle (θ) and azimuth angle (ϕ) as commonly used in physics (ISO convention). The antenna gain pattern is given using this type of representation. We define the E-plane and the H-plane as XZ-plane and YZ-plane respectively.

Relationship between variables: Whenever a user changes the value of any variable by means of the available functions, the rest of them are updated to keep consistency with the following equations:

- **antenna_Aeff** = (C_LIGHT/**frequency**)^2 * 10.0**(**antenna_Gain_dB**/10.0)/(4.0 * π)
- **noise_T** = **antenna_T** + 290.0 * (10.0**(**noise_F_dB**/10.0) - 1.0)
- **noise_pow_dBW** = 10.0 * log10(K_BOLTZ * **noise_T** * **filter_BB_BW**)

where C_LIGHT is the speed of light in vacuum and K_BOLTZ is the Boltzmann's constant.

3.2 Relevant private variables

- **antenna_pattern_dB** (double, 2D array of 181 x 360 elements): Antenna pattern (for a single element) as a function of θ and ϕ in the antenna frame in dB. Default value: 0.0 for all elements
- **antenna_vector_BF_E** (double, array of 3 elements): Antenna frame's X-axis in the receiver body frame (defines the orientation of the antenna in the receiver frame). Default value: [1.0, 0.0, 0.0]
- **antenna_vector_BF_H** (double, array of 3 elements): Antenna frame's Y-axis in the receiver body frame (defines the orientation of the antenna in the receiver frame). Default value: [0.0, 1.0, 0.0]
- **antenna_vector_BF_k** (double, array of 3 elements): Antenna frame's Z-axis in the receiver body frame (defines the orientation of the antenna in the receiver frame). Default value: [0.0, 0.0, 1.0]
- **isotropic** (boolean): True - An isotropic antenna is employed (antenna pattern neglected), False - A non-isotropic antenna is employed. Default value: True
- **frequency** (double): Frequency of the receiver in Hz. Default value: 1575420000.0 (GPS L1)
- **antenna_Gain_dB** (double): Antenna gain in dB (this value is added to the antenna pattern). Default value: 3.0
- **antenna_Aeff** (double): Effective area of the antenna in meters^2. Default value: 0.00575

- **antenna_T** (double): Antenna temperature in K. Default value: 200.0
- **noise_T** (double): Noise temperature in K. Default value: 488.626
- **noise_pow_dBW** (double): Noise power in dBW. Default value: -134.719
- **noise_F_dB** (double): Noise figure in dB. Default value: 3.0
- **filter_BB_BW** (double): Base-band bandwidth of the receiver in Hz. Default value: 5000000.0
- **array_num_elements** (integer): If > 1, the antenna is a 2D planar array of such number of elements. Default value: 1 (single antenna)
- **element_pos_AF** (double, 2D array of **array_num_elements** x 2 elements): Position of the array elements as 2D coordinates in plane XY of the antenna frame in meters. Default value: void
- **phase_delay** (double, array of **array_num_elements**): Phase applied to each element to obtain a desired array factor in radians. Default value: void
- **array_factor_dB** (double, 2D array of 90 x 360 elements): Array factor as a function of θ and ϕ in the antenna frame in dB. Default value: void
- **array_factor_ready** (boolean): True - Array factor is computed, False - Array factor is not computed. Default value: False

3.3 Functions

3.3.1 dump_parameters

Print relevant information about the object's content (public and private variables).

Example:

```
my_receiver.dump_parameters()
```

3.3.2 set_antenna_orientation_BF_EH

Set the orientation of the antenna in the receiver body frame by editing private variables **antenna_vector_BF_E** and **antenna_vector_BF_H** (**antenna_vector_BF_k** is constructed using the right hand rule).

Example:

```
my_receiver.set_antenna_orientation_BF_EH(vector_E_in, vector_H_in)
```

Input variables:

- **vector_E_in** (double, array of 3 elements): Antenna frame's X-axis in the receiver body frame.
- **vector_H_in** (double, array of 3 elements): Antenna frame's Y-axis in the receiver body frame.

3.3.3 set_antenna_orientation_BF_k

Set the pointing direction of the antenna in the receiver body frame by editing the private variable **antenna_vector_BF_k**. In this case, **antenna_vector_BF_E** and **antenna_vector_BF_H** are arbitrary constructed using the right hand rule (this method is valid when there is symmetry between both axis).

Example:

```
my_receiver.set_antenna_orientation_BF_k(vector_k_in)
```

Input variables:

- **vector_k_in** (double, array of 3 elements): Antenna frame's Z-axis in the receiver body frame.

3.3.4 get_antenna_orientation_BF

Get the antenna orientation.

Example:

```
[vector_E_out, vector_H_out, vector_k_out] = my_receiver.get_antenna_orientation_BF()
```

Output variables:

- **vector_E_out** (double, array of 3 elements): Antenna frame's X-axis in the receiver body frame.
- **vector_H_out** (double, array of 3 elements): Antenna frame's Y-axis in the receiver body frame.
- **vector_k_out** (double, array of 3 elements): Antenna frame's Z-axis in the receiver body frame.

3.3.5 set_antenna_whole_pattern

Set the antenna pattern.

Example:

```
my_receiver.set_antenna_whole_pattern(ant_pattern)
```

Input variables:

- **ant_pattern** (double, 2D array of 181 x 360 elements): Antenna pattern as a function of θ and ϕ in the antenna frame in dB.

3.3.6 set_val_antenna_pattern

Set an individual value of the antenna pattern.

Example:

```
my_receiver.set_val_antenna_pattern(phi_index, theta_index, pattern_dB_value)
```

Input variables:

- **phi_index** (integer): Sample at coordinate ϕ of the antenna pattern (coincides with the integer value of angle ϕ in the antenna frame).
- **theta_index** (integer): Sample at coordinate θ of the antenna pattern (coincides with the integer value of angle θ in the antenna frame).
- **pattern_dB_value** (double): Value of the antenna pattern in dB.

3.3.7 set_antenna_pattern_FF

Set antenna pattern from a single full (for all θ angles) full (for ϕ=0 and ϕ=180) cut at the E-plane. It is assumed that both E- and H-planes are equal and interpolation is applied for the remaining points.

Example:

```
my_receiver.set_antenna_pattern_FF(ant_pattern_E_cut)
```

Input variables:

- **ant_pattern_E_cut** (double, array of 360 elements): Antenna pattern in the E-plane as a function of θ angle in dB.

3.3.8 set_antenna_pattern_FH

Set antenna pattern from a single full (for all θ angles) half (only for ϕ=0) cut at the E-plane. It is assumed that both E- and H-planes are equal and interpolation is applied for the remaining points.

Example:

```
my_receiver.set_antenna_pattern_FH(ant_pattern_E_halfcut)
```

Input variables:

- **ant_pattern_E_halfcut** (double, array of 181 elements): Antenna pattern in the E-plane as a function of θ angle in dB.

3.3.9 set_antenna_pattern_interp

Set the antenna pattern by providing a set of points in the E-plane and a minimum level. The whole shape of the E-plane pattern is then built by means of a piece-wise spline interpolation strategy (spline interpolation for each segment between two minimum-level values). If θ values range from 0 to 180 degrees, symmetry is applied. Finally, it is assumed that both E- and H-planes are equal and interpolation is applied for the remaining points.

Example:

```
my_receiver.set_antenna_pattern_interp(theta_points, pattern_points,
min_level)
```

Input variables:

- **theta_points** (double, array of N elements): Theta angle values in the E-plane in degrees.
- **pattern_points** (double, array of N elements): Antenna pattern values at **theta_points** in dB.
- **min_level** (double): Minimum level to split the spline interpolation between contiguous segments.

3.3.10 set_antenna_patterns_FF

Set antenna pattern from single full (for all θ angles) full (for ϕ=0/90 and ϕ=180/270) cuts at both E- and H-plane. Interpolation is applied for the remaining points.

Example:

```
my_receiver.set_antenna_patterns_FF(ant_pattern_E_cut, ant_pattern_H_cut)
```

Input variables:

- **ant_pattern_E_cut** (double, array of 360 elements): Antenna pattern in the E-plane as a function of θ angle in dB.
- **ant_pattern_H_cut** (double, array of 360 elements): Antenna pattern in the H-plane as a function of θ angle in dB.

3.3.11 set_antenna_patterns_FH

Set antenna pattern from single full (for all θ angles) half (only for ϕ=0/90) cuts at both E- and H-plane. Interpolation is applied for the remaining points.

Example:

```
my_receiver.set_antenna_patterns_FH(ant_pattern_E_halfcut, ant_pattern_H_halfcut)
```

Input variables:

- **ant_pattern_E_halfcut** (double, array of 181 elements): Antenna pattern in the E-plane as a function of θ angle in dB.
- **ant_pattern_H_halfcut** (double, array of 181 elements): Antenna pattern in the H-plane as a function of θ angle in dB.

3.3.12 set_antenna_patterns_interp

Set the antenna pattern by providing a set of points in both E- and H-planes and a minimum level. The whole shape of both E- and H-plane patterns is then built by means of a piece-wise spline interpolation strategy (spline interpolation for each segment between two minimum-level values). If θ values range from 0 to 180 degrees, symmetry is applied. Finally, interpolation is applied for the remaining points.

Example:

```
my_receiver.set_antenna_patterns_interp(theta_E_points, pattern_E_points,
theta_H_points, pattern_H_points, min_level)
```

Input variables:

- **theta_E_points** (double, array of N elements): Theta angle values in the E-plane in degrees.
- **pattern_E_points** (double, array of N elements): Antenna pattern values at theta_E_points in dB.
- **theta_H_points** (double, array of M elements): Theta angle values in the H-plane in degrees.
- **pattern_H_points** (double, array of M elements): Antenna pattern values at theta_H_points in dB.
- **min_level** (double): Minimum level to split the spline interpolation between contiguous segments.

3.3.13 get_antenna_whole_pattern

Get the antenna pattern.

Example:

```
ant_pattern = my_receiver.get_antenna_whole_pattern
```

Output variables:

- **ant_pattern** (double, 2D array of 181 x 360 elements): Antenna pattern as a function of θ and ϕ in the antenna frame in dB.

3.3.14 get_antenna_patterns

Get the E- and H-plane cuts of the antenna pattern.

Example:

```
[ant_pattern_E, ant_pattern_H] = my_receiver.get_antenna_patterns()
```

Output variables:

- **ant_pattern_E** (double, array of 360 elements): E-plane cut of the antenna pattern in dB.
- **ant_pattern_H** (double, array of 360 elements): H-plane cut of the antenna pattern in dB.

3.3.15 set_receiver_params

Set the main parameters of the receiver (by editing the corresponding private variables).

Example:

```
my_receiver.set_receiver_params(antenna_Gain_dB_in, antenna_T_in, noise_F_dB_in,
filter_BW_in, isotropic_antenna)
```

Input variables:

- **antenna_Gain_dB_in** (double): Antenna gain in dB.
- **antenna_T_in** (double): Antenna temperature in K.
- **noise_F_dB_in** (double): Noise figure in dB.
- **filter_BW_in** (double): Base-band bandwidth of the receiver in Hz.
- **isotropic_antenna** (char): 1 - Isotropic antenna, 0 - Non-isotropic antenna.

3.3.16 set_antenna_eff_area

Set the antenna effective area.

Example:

```
my_receiver.set_antenna_eff_area(antenna_Aeff_in)
```

Input variables:

- **antenna_Aeff_in** (double): Effective area of the antenna in meters^2.

3.3.17 set_noise_T

Set the noise temperature.

Example:

```
my_receiver.set_noise_T(noise_T_in)
```

Input variables:

- **noise_T_in** (double): Noise temperature in K.

3.3.18 set_noise_pow_dBW

Set the noise power.

Example:

```
my_receiver.set_noise_pow_dBW(noise_pow_dBW_in)
```

Input variables:

- **noise_pow_dBW_in** (double): Noise power in dBW.

3.3.19 set_frequency

Set the frequency of the receiver.

Example:

```
my_receiver.set_frequency(frequency_in)
```

Input variables:

- **frequency_in** (double): Frequency in Hz.

3.3.20 get_PhiTheta_gain_dB

Get the antenna gain as a function of ϕ and θ angles in the antenna frame.

Example:

```
my_receiver.get_PhiTheta_gain_dB(phi, theta)
```

Input variables:

- **phi** (double): Angle ϕ in the antenna frame in degrees.
- **theta** (double): Angle θ in the antenna frame in degrees.

3.3.21 get_incvector_gain_dB

Get the antenna gain as a function of the incidence or transmitting vector in the receiver body frame.

Example:

```
my_receiver.get_incvector_gain_dB(incvector)
```

Input variables:

- **incvector** (double, array of 3 elements): Incidence or transmitting vector in the receiver body frame.

3.3.22 get_frequency

Get the frequency of the receiver.

Example:

```
freq = my_receiver.get_frequency()
```

Output variables:

- **freq** (double): Frequency of the receiver in Hz.

3.3.23 get_antenna_Gain_dB

Get the antenna gain (for isotropic antennas).

Example:

```
gain = my_receiver.get_antenna_Gain_dB()
```

Output variables:

- **gain** (double): Antenna gain in dB.

3.3.24 get_antenna_Aeff

Get the effective area of the antenna.

Example:

```
eff_area = my_receiver.get_antenna_Aeff()
```

Output variables:

- **eff_area** (double): Effective area of the antenna in meters^2.

3.3.25 get_antenna_T

Get the antenna temperature.

Example:

```
ant_temp = my_receiver.get_antenna_T()
```

Output variables:

- **ant_temp** (double): Antenna temperature in K.

3.3.26 get_noise_T

Get the noise temperature.

Example:

```
noise_temp = my_receiver.get_noise_T()
```

Output variables:

- **noise_temp** (double): Noise temperature in K.

3.3.27 get_noise_pow_dBW

Get the noise power.

Example:

```
noise_pow = my_receiver.get_noise_pow_dBW()
```

Output variables:

- **noise_pow** (double): Noise power in dBW.

3.3.28 get_noise_F_dB

Get the noise figure.

Example:

```
noise_F = my_receiver.get_noise_F_dB()
```

Output variables:

- **noise_F** (double): Noise figure in dB.

3.3.29 get_filter_BB_BW

Get the base-band bandwidth of the receiver.

Example:

```
BW = my_receiver.get_filter_BB_BW()
```

Output variables:

- **BW** (double): Base-band bandwidth of the receiver in Hz.

3.3.30 set_antenna_elements_pos_AF

Set a distribution of antenna elements over the antenna frame to have a 2D planar array.

Example:

```
my_receiver.set_antenna_elements_pos_AF(element_pos_in, lambda_units)
```

Input variables:

- **element_pos_in** (double, 2D array of N x 2 elements): Positions of the array elements as 2D coordinates in plane XY of the antenna frame in meters or in units of lambda in **lambda_units** = 1.
- **lambda_units** (char): 1 - **element_pos_in** is given in lambda units, 0/else - **element_pos_in** is given in meters.

3.3.31 set_phase_delays

Set the phases applied to each element to obtain a desired array factor.

Example:

```
my_receiver.set_phase_delays(phase_delay_in)
```

Input variables:

- **phase_delay_in** (double, array of num_elements elements): Phases applied to each element to obtain a desired array factor in radians.

3.3.32 get_phase_delays

Get the phases applied to each element to obtain a desired array factor.

Example:

```
phase_delay_out = my_receiver.get_phase_delays(num_elements_out)
```

Input variables:

- **num_elements_out** (integer): Number of elements of the array (it has to be equal to **num_elements**).

Output variables:

- **phase_delay_out** (double, array of **num_elements_out** elements): Phases applied to each element to obtain a desired array factor in radians.

3.3.33 compute_array_factor

Compute and store the array factor based on the phases and the distribution of the antenna elements.

Example:

```
my_receiver.compute_array_factor()
```

3.3.34 get_array_factor

Get the array factor stored.

Example:

```
array_factor_out = my_receiver.get_array_factor()
```

Output variables:

- **array_factor_out** (double, 2D array of 90 x 360 elements): Array factor as a function of θ and ϕ in the antenna frame in dB.

3.3.35 compute_phase_delays_UPA

Compute the phases to be applied to each element to get an array factor with a desired pointing direction, based on uniformly-distributed planar array (UPA) theory.

Example:

```
my_receiver.compute_phase_delays_UPA(theta_max, phi_max)
```

Input variables:

- **theta_max** (double): Angle θ with maximum array factor gain in the antenna frame in degrees.

- **phi_max** (double): Angle ϕ with maximum array factor gain in the antenna frame in degrees.

3.3.36 compute_phase_delays_pos_ECEF_RT

Compute the phases to be applied to each element to get an array factor with a desired pointing direction, based on the ECEF position of a receiver (where the array is placed), a transmitter (where the array is pointing at) and the inertial information of the receiver.

Example:

```
my_receiver.compute_phase_delays_pos_ECEF_RT([roll_in, pitch_in, yaw_in],
posR_km, posT_km)
```

Input variables:

- **roll_in** (double): Inertial rotation of the X-axis (positive clockwise) of the receiver body frame in degrees.
- **pitch_in** (double): Inertial rotation of the Y-axis (positive clockwise) of the receiver body frame in degrees.
- **heading_in** (double): Inertial rotation of the Z-axis (positive clockwise) of the receiver body frame in degrees.
- **posR_km** (double, array of 3 elements): Position of the receiver in ECEF coordinates in km.
- **posT_km** (double, array of 3 elements): Position of the transmitter in ECEF coordinates in km.

4 Specular_geometry

This class provides functions to characterize a reflectometry scenario composed by a transmitter, a receiver and their corresponding specular point over a model of the Earth surface (ellipsoid WGS84 + a given undulation).

Example of object construction with arbitrary name *my_geom*:

```
my_geom = wavpy.Specular_geometry()
```

In this case, public variables and functions from object *my_geom* of class *Specular_geometry* can be respectively checked/modified or called with:

```
my_geom.variable
```

```
my_geom.function()
```

4.1 Additional information

Receiver frame: The receiver body frame is a Cartesian coordinates system of the structure containing the receiver, typically a satellite or an aircraft, where the X-axis points towards the front, the Y-axis points towards the right-side (XY define the horizontal plane) and the Z-axis towards Nadir. In abscense of inertial rotation of the body frame, we will assume that the X-axis points towards the Earth's North and the Z-axis points towards the Earth's center.

Local frame: The local frame is a Cartesian coordinates system with its origin at the specular point, X- and Y-axis defining the horizontal plane parallel to the surface, with the Y-axis pointing towards the transmitter, and the Z-axis pointing to Zenith by complying the right-hand rule.

4.2 Public variables

- **longitudeS** (double): Longitude coordinate of the specular point in degrees. Default value: 0.0
- **latitudeS** (double): Latitude coordinate of the specular point in degrees. Default value: 0.0
- **elevation** (double): Elevation angle of the transmitter at the specular point in degrees. Default value: 90.0

  a
- **azimuthR** (double): Azimuth angle of the receiver at the specular point in degrees. Default value: 0.0
- **azimuthT** (double): Azimuth angle of the transmitter at the specular point in degrees. Default value: 0.0
- **geometric_delay** (double): Delay path difference between transmitter-specular-receiver and transmitter-receiver in km. Default value: 6.0

4.3 Relevant private variables

- **posR_ECEF** (double, array of 3 elements): Position of the receiver in ECEF coordinates in km. Default value: [6381.137, 0.0, 0.0]
- **posT_ECEF** (double, array of 3 elements): Position of the transmitter in ECEF coordinates in km. Default value: [26378.137, 0.0, 0.0]
- **velR_ECEF** (double, array of 3 elements): Velocity vector of the receiver in ECEF coordinates in km/s. Default value: [0.0, 0.0, 0.0]
- **velT_ECEF** (double, array of 3 elements): Velocity vector of the transmitter in ECEF coordinates in km/s. Default value: [0.0, 0.0, 0.0]
- **posS_ECEF** (double, array of 3 elements): Position of the specular point in ECEF coordinates in km. Default value: [6378.137, 0.0, 0.0]
- **longitudeR** (double): Longitude coordinate of the receiver in degrees. Default value: 0.0
- **latitudeR** (double): Latitude coordinate of the receiver in degrees. Default value: 0.0

- **longitudeT** (double): Longitude coordinate of the transmitter in degrees. Default value: 0.0
- **latitudeT** (double): Latitude coordinate of the transmitter in degrees. Default value: 0.0
- **heightR** (double): Height of the receiver with respect to ellipsoid WGS84 in km. Default value: 3.0
- **heightT** (double): Height of the transmitter with respect to ellipsoid WGS84 in km. Default value: 20000.0
- **local_heightR** (double): Height of the receiver in the local frame in km. Default value: 3.0
- **local_heightT** (double): Height of the transmitter in the local frame in km. Default value: 3.0
- **undulation** (double): Vertical height offset of the specular point with respect to ellipsoid WGS84 in km. Default value: 0.0
- **roll** (double): Inertial rotation of the X-axis (positive clockwise) of the receiver body frame in degrees. Default value: 0.0
- **pitch** (double): Inertial rotation of the Y-axis (positive clockwise) of the receiver body frame in degrees. Default value: 0.0
- **heading** (double): Inertial rotation of the Z-axis (positive clockwise) of the receiver body frame with respect to North in degrees. Default value: 0.0

4.4 Functions

4.4.1 dump_parameters

Print relevant information about the object's content (public and private variables).

Example:

```
my_geom.dump_parameters()
```

4.4.2 set_ECEFpos_Receiver

Set the position of the receiver in ECEF coordinates. The orientation of the Z-axis of the receiver body frame with respect to North (**heading**) is updated with the receiver's position and flight direction.

Example:

```
my_geom.set_ECEFpos_Receiver(posR_in)
```

Input variables:

- **posR_in** (double, array of 3 elements): Position of the receiver in ECEF coordinates in km.

4.4.3 get_ECEFpos_Receiver

Get the position of the receiver in ECEF coordinates.

Example:

```
posR_out = my_geom.get_ECEFpos_Receiver()
```

Output variables:

- **posR_out** (double, array of 3 elements): Position of the receiver in ECEF coordinates in km.

4.4.4 set_ECEFvel_Receiver

Set the velocity vector of the receiver in ECEF coordinates. The orientation of the Z-axis of the receiver body frame with respect to North (**heading**) is updated with the receiver's position and flight direction.

Example:

```
my_geom.set_ECEFvel_Receiver(velR_in)
```

Input variables:

- **velR_in** (double, array of 3 elements): Velocity vector of the receiver in ECEF coordinates in km/s.

4.4.5 get_ECEFvel_Receiver

Get the position of the receiver in ECEF coordinates.

Example:

```
velR_out = my_geom.get_ECEFvel_Receiver()
```

Output variables:

- **velR_out** (double, array of 3 elements): Velocity vector of the receiver in ECEF coordinates in km/s.

4.4.6 set_ECEFpos_Transmitter

Set the position of the transmitter in ECEF coordinates.

Example:

```
my_geom.set_ECEFpos_Transmitter(posT_in)
```

Input variables:

- **posT_in** (double, array of 3 elements): Position of the transmitter in ECEF coordinates in km.

4.4.7 get_ECEFpos_Transmitter

Get the position of the transmitter in ECEF coordinates.

Example:

```
posT_out = my_geom.get_ECEFpos_Transmitter()
```

Output variables:

- **posT_out** (double, array of 3 elements): Position of the transmitter in ECEF coordinates in km.

4.4.8 set_ECEFvel_Transmitter

Set the velocity vector of the transmitter in ECEF coordinates.

Example:

```
my_geom.set_ECEFvel_Transmitter(velT_in)
```

Input variables:

- **velT_in** (double, array of 3 elements): Velocity vector of the transmitter in ECEF coordinates in km/s.

4.4.9 get_ECEFvel_Transmitter

Get the position of the transmitter in ECEF coordinates.

Example:

```
velT_out = my_geom.get_ECEFvel_Transmitter()
```

Output variables:

- **velT_out** (double, array of 3 elements): Velocity vector of the transmitter in ECEF coordinates in km/s.

4.4.10 get_ECEFpos_Specular

Get the position of the specular point in ECEF coordinates.

Example:

```
posS_out = my_geom.get_ECEFpos_Specular()
```

Output variables:

- **posS_out** (double, array of 3 elements): Position of the specular point in ECEF coordinates in km.

4.4.11 set_LongLatHeight_Receiver

Set the position of the receiver with Longitude-Latitude-Height coordinates. The orientation of the Z-axis of the receiver body frame with respect to North (**heading**) is updated with the receiver's position and flight direction.

Example:

```
my_geom.set_LongLatHeight_Receiver([lonR_in, latR_in, heightR_in])
```

Input variables:

- **lonR_in** (double): Longitude coordinate of the receiver in degrees.
- **latR_in** (double): Latitude coordinate of the receiver in degrees.
- **heightR_in** (double): Height of the receiver with respect to ellipsoid WGS84 in km.

4.4.12 get_LongLatHeight_Receiver

Get the position of the receiver with Longitude-Latitude-Height coordinates.

Example:

```
[lonR_out, latR_out, heightR_out] = my_geom.get_LongLatHeight_Receiver()
```

Output variables:

- **lonR_out** (double): Longitude coordinate of the receiver in degrees.
- **latR_out** (double): Latitude coordinate of the receiver in degrees.
- **heightR_out** (double): Height of the receiver with respect to ellipsoid WGS84 in km.

4.4.13 set_LongLatHeight_Transmitter

Set the position of the transmitter with Longitude-Latitude-Height coordinates.

Example:

```
my_geom.set_LongLatHeight_Transmitter([lonT_in, latT_in, heightT_in])
```

Input variables:

- **lonT_in** (double): Longitude coordinate of the transmitter in degrees.
- **latT_in** (double): Latitude coordinate of the transmitter in degrees.

- **heightT_in** (double): Height of the transmitter with respect to ellipsoid WGS84 in km.

4.4.14 get_LongLatHeight_Transmitter

Get the position of the transmitter with Longitude-Latitude-Height coordinates.

Example:

```
[lonT_out, latT_out, heightT_out] = my_geom.get_LongLatHeight_Transmitter()
```

Output variables:

- **lonT_out** (double): Longitude coordinate of the transmitter in degrees.
- **latT_out** (double): Latitude coordinate of the transmitter in degrees.
- **heightT_out** (double): Height of the transmitter with respect to ellipsoid WGS84 in km.

4.4.15 set_geometry_from_ElevHeightsSpec

Set the geometry of the different elements from their heights, the elevation and azimuth angles, and the location of the specular point in Longitude-Latitude coordinates. The orientation of the Z-axis of the receiver body frame with respect to North (**heading**) is updated with the receiver's position and flight direction.

Example:

```
my_geom.set_geometry_from_ElevHeightsSpec(elev_in, heightR_in, heightT_in,
lonS_in, latS_in, azimT_in, heightS_in)
```

Input variables:

- **elev_in** (double): Elevation angle of the transmitter at the specular point in degrees.
- **heightR_in** (double): Height of the receiver with respect to ellipsoid WGS84 in km.

- **heightT_in** (double): Height of the transmitter with respect to ellipsoid WGS84 in km.
- **lonS_in** (double): Longitude coordinate of the specular point in degrees.
- **latS_in** (double): Latitude coordinate of the specular point in degrees.
- **azimT_in** (double): Azimuth angle of the transmitter at the specular point in degrees.
- **heightS_in** (double): Vertical height offset of the specular point with respect to ellipsoid WGS84 in km.

4.4.16 set_tangEarthVel_Receiver

Set a velocity vector for the receiver tangential to the Earth surface. The orientation of the Z-axis of the receiver body frame with respect to North (**heading**) is updated with the receiver's position and flight direction.

Example:

```
my_geom.set_tangEarthVel_Receiver(velocity, specAzim)
```

Input variables:

- **velocity** (double): Speed of the receiver in km/s.
- **specAzim** (double): Clockwise azimuth angle with respect to the pointing direction towards the specular point in degrees.

4.4.17 set_tangEarthVel_Transmitter

Set a velocity vector for the transmitter tangential to the Earth surface.

Example:

```
my_geom.set_tangEarthVel_Transmitter(velocity, specAzim)
```

Input variables:

- **velocity** (double): Speed of the transmitter in km/s.
- **specAzim** (double): Clockwise azimuth angle with respect to the pointing direction towards the specular point in degrees.

4.4.18 set_Undulation

Set a vertical height offset of the specular point with respect to ellipsoid WGS84.

Example:

```
my_geom.set_Undulation(undu_in)
```

Input variables:

- **undu_in** (double): Vertical height offset of the specular point with respect to ellipsoid WGS84 in km.

4.4.19 get_Undulation

Get the vertical height offset of the specular point with respect to ellipsoid WGS84.

Example:

```
undu_out = my_geom.get_Undulation()
```

Output variables:

- **undu_out** (double): Vertical height offset of the specular point with respect to ellipsoid WGS84 in km.

4.4.20 read_ECEFpos_Receiver

Set the ECEF position and velocity of the receiver from an ASCII file of five columns containing the following variables: GPS_week - Second_of_week - Position_X_km - Position_Y_km - Position_Z_km. Interpolation is applied when required. The orientation of the Z-axis of the receiver body frame with respect to North (**heading**) is updated with the receiver's position and flight direction.

Example:

```
my_geom.read_ECEFpos_Receiver(file, week, sow)
```

Input variables:

- **file** (string): Filename of the ASCII file containing the time series of the receiver's ECEF position.
- **week** (integer): GPS week.
- **sow** (double): GPS second of the week.

4.4.21 read_ECEFpos_Transmitter

Set the ECEF position and velocity of the transmitter from an ASCII file of five columns containing the following variables: GPS_week - Second_of_week - Position_X_km - Position_Y_km - Position_Z_km. Interpolation is applied when required.

Example:

```
my_geom.read_ECEFpos_Transmitter(file, week, sow)
```

Input variables:

- **file** (string): Filename of the ASCII file containing the time series of the transmitter's ECEF position.
- **week** (integer): GPS week.
- **sow** (double): GPS second of the week.

4.4.22 read_ECEFpos_GNSS_Transmitter

Set the ECEF position and velocity of a GNSS transmitter from a SP3 file. If several reads have to be done at the same SP3 file, it is better to use *load_sp3File*, *read_ECEFpos_GNSS_Transmitter_sp3Loaded* and *free_sp3File*.

Example:

```
my_geom.read_ECEFpos_GNSS_Transmitter(sp3_file, week, sow, prn, gnss_ident)
```

Input variables:

- **sp3_file** (string): Filename of the SP3 file containing ECEF positions of several GNSS satellites.
- **week** (integer): GPS week.

- **sow** (double): GPS second of the week.
- **prn** (integer): PRN of the desired GNSS satellite.
- **gnss_ident** (char): GNSS identifier ('G' for GPS, 'E' for Galileo, 'C' for Beidou, 'R' for GLONASS and 'J' for QZSS).

4.4.23 load_sp3File

Load the orbits of a single GNSS from a SP3 file.

Example:

```
my_geom.load_sp3File(sp3_file, gnss_ident)
```

Input variables:

- **sp3_file** (string): Filename of the SP3 file containing ECEF positions of several GNSS satellites.
- **gnss_ident** (char): GNSS identifier ('G' for GPS, 'E' for Galileo, 'C' for Beidou, 'R' for GLONASS and 'J' for QZSS).

4.4.24 free_sp3File

Free from memory previous loaded orbits of a single GNSS from a SP3 file.

Example:

```
my_geom.free_sp3File()
```

4.4.25 read_ECEFpos_GNSS_Transmitter_sp3Loaded

Set the ECEF position and velocity of a GNSS transmitter from a previously loaded SP3 file.

Example:

```
my_geom.read_ECEFpos_GNSS_Transmitter_sp3Loaded(week, sow, prn, gnss_ident)
```

Input variables:

- **week** (integer): GPS week.
- **sow** (double): GPS second of the week.
- **prn** (integer): PRN of the desired GNSS satellite.
- **gnss_ident** (char): GNSS identifier ('G' for GPS, 'E' for Galileo, 'C' for Beidou, 'R' for GLONASS and 'J' for QZSS).

4.4.26 compute_specular_point

Compute the specular point from the positions of transmitter and receiver over the ellipsoid WGS84 plus a given undulation by increasing both semi-axis with such value.

Example:

```
my_geom.compute_specular_point(compute_undu)
```

Input variables:

- **compute_undu** (char): "1" to interpolate undulation from EGM96 stored grid and "0" to do not compute undulation.

4.4.27 compute_specular_point_Undu_Spherical_Earth

Compute the specular point from the positions of transmitter and receiver over the ellipsoid WGS84 plus a given undulation by increasing the radius of curvature at initial specular position over WGS84.

Example:

```
my_geom.compute_specular_point_Undu_Spherical_Earth()
```

4.4.28 compute_ElevAzimT_from_receiver

Compute elevation and azimuth angles of the transmitter as seen from the receiver's point of view.

Example:

```
[elevT_R, azimT_R] = my_geom.compute_ElevAzimT_from_receiver()
```

Output variables:

- **elevT_R** (double): Elevation angle of the transmitter from the receiver's point of view in degrees.
- **azimT_R** (double): Azimuth angle of the transmitter from the receiver's point of view in degrees.

4.4.29 set_inertials

Set the inertial rotation of the receiver, including its orientation with respect to North.

Example:

```
my_geom.set_inertials(roll_in, pitch_in, heading_in)
```

Input variables:

- **roll_in** (double): Inertial rotation of the X-axis (positive clockwise) of the receiver body frame in degrees.
- **pitch_in** (double): Inertial rotation of the Y-axis (positive clockwise) of the receiver body frame in degrees.
- **heading_in** (double): Inertial rotation of the Z-axis (positive clockwise) of the receiver body frame with respect to North in degrees.

4.4.30 get_inertials

Get the stored inertial rotation of the receiver.

Example:

```
[roll_out, pitch_out, heading_out] = my_geom.get_inertials()
```

Output variables:

- **roll_out** (double): Inertial rotation of the X-axis (positive clockwise) of the receiver body frame in degrees.

- **pitch_out** (double): Inertial rotation of the Y-axis (positive clockwise) of the receiver body frame in degrees.
- **heading_out** (double): Inertial rotation of the Z-axis (positive clockwise) of the receiver body frame with respect to North in degrees.

4.4.31 rotate_vector_BF_to_local

Rotate an input vector in the receiver body frame to the local frame's orientation (it is not a change of coordinates).

Example:

```
vector_local_out = my_geom.rotate_vector_BF_to_local(vector_BF_in)
```

Input variables:

- **vector_BF_in** (double, array of 3 elements): Input vector in the receiver body frame.

Output variables:

- **vector_local_out** (double, array of 3 elements): Output vector with the local frame's orientation.

4.4.32 rotate_vector_BF_to_ECEF

Rotate an input vector in the receiver body frame to ECEF orientation (it is not a change of coordinates).

Example:

```
vector_ECEF_out = my_geom.rotate_vector_BF_to_ECEF(vector_BF_in)
```

Input variables:

- **vector_BF_in** (double, array of 3 elements): Input vector in the receiver body frame.

Output variables:

- **vector_ECEF_out** (double, array of 3 elements): Output vector with ECEF orientation.

4.4.33 compute_inertial_delay

Compute the projection of an input vector in the receiver body frame into the reflected signal's delay path.

Example:

```
inertdel = my_geom.compute_inertial_delay(vector_BF_in)
```

Input variables:

- **vector_BF_in** (double, array of 3 elements): Input vector in the receiver body frame.

Output variables:

- **inertdel** (double): Projection of the input vector into the reflected signal's delay path in the same units as vector_BF_in.

4.4.34 read_Inertials_Receiver

Set the inertial rotation of the receiver from an ASCII file of five columns containing the following variables: GPS_week - Second_of_week - roll_deg - pitch_deg - yaw_deg. Interpolation is applied when required.

Example:

```
my_geom.read_Inertials_Receiver(file, week, sow)
```

Input variables:

- **file** (string): Filename of the ASCII file containing the time series of the receiver's inertial rotation.
- **week** (integer): GPS week.
- **sow** (double): GPS second of the week.

4.4.35 compute_Beyerle_windup_direct

Compute the carrier phase wind-up of the up-looking antenna (collecting direct GNSS signals) based on [**Beyerle, 09**].

Example:

```
[windup_R, windup_L] = my_geom.compute_Beyerle_windup_direct(vector_r_a_BF,
double vector_r_t_BF, int week, double sow)
```

Input variables:

- **vector_r_a_BF** (double, array of 3 elements): Antenna vector r_a (as in [**Beyerle, 09**]) in the receiver body frame.
- **vector_r_t_BF** (double, array of 3 elements): Antenna vector r_t (as in [**Beyerle, 09**]) in the receiver body frame.
- **week** (integer): GPS week.
- **sow** (double): GPS second of the week.

Output variables:

- **windup_R** (double): Carrier phase wind-up for RHCP polarization in radians.
- **windup_L** (double): Carrier phase wind-up for LHCP polarization in radians.

4.4.36 compute_Beyerle_windup_reflected

Compute the carrier phase wind-up of the down-looking antenna (collecting reflected GNSS signals) based on [**Beyerle, 09**].

Example:

```
[windup_R, windup_L] = my_geom.compute_Beyerle_windup_reflected(vector_r_a_BF,
vector_r_t_BF, rvv, rhh, week, sow)
```

Input variables:

- **vector_r_a_BF** (double, array of 3 elements): Antenna vector r_a (as in [**Beyerle, 09**]) in the receiver body frame.
- **vector_r_t_BF** (double, array of 3 elements): Antenna vector r_t (as in [**Beyerle, 09**]) in the receiver body frame.

- **rvv** (double, array of 2 elements): Complex reflection coefficient for vertical polarization (real and imaginary parts).
- **rhh** (double, array of 2 elements): Complex reflection coefficient for horizontal polarization (real and imaginary parts).
- **week** (integer): GPS week.
- **sow** (double): GPS second of the week.

Output variables:

- **windup_R** (double): Carrier phase wind-up for RHCP polarization in radians.
- **windup_L** (double): Carrier phase wind-up for LHCP polarization in radians.

5 Reflecting_surface

This class provides functions to characterize a reflecting surface in a radar scenario, in particular regarding its dielectric properties (based on **[Ulaby et al, 90]**) and its roughness.

Example of object construction with arbitrary name *my_surface*:

```
my_surface = wavpy.Reflecting_surface()
```

In this case, public variables and functions from object *my_surface* of class *Reflecting_surface* can be respectively checked/modified or called with:

```
my_surface.variable
```

```
my_surface.function()
```

5.1 Public variables

- **epsilon_real** (double): Real part of relative permittivity without units. Default value: 73.423
- **epsilon_imag** (double): Imaginary part of relative permittivity without units. Default value: 56.067
- **mss_x** (double): Mean square slope at upwind direction over the reflecting surface without units. Default value: 0.0075
- **mss_y** (double): Mean square slope at crosswind direction over the reflecting surface without units. Default value: 0.0075
- **sigma_z** (double): Standard deviation of surface height in meters. Default value: 0.069
- **c21_coeff** (double): Gram-Charlier C21 coefficient for 5 m/s wind. Default value: -0.033

- **c03_coeff** (double): Gram-Charlier C03 coefficient for 5 m/s wind. Default value: -0.125
- **wind_U10_speed** (double): Wind speed magnitude at 10 meters above the surface in m/sec. Default value: 5.0
- **wind_U10_azimuth** (double): Wind azimuth (clockwise starting from North) at 10 meters above the surface in degrees. The direction follows the meteorological convention (0° means wind coming from North). Default value: 0.0
- **medium** (string): Brief description of the medium. Default value: "Sea water with T=15C and sal=35psu"

5.2 Relevant private variables

- **freq_GHz** (double): Frequency of the system in GHz. Default value: 1.57542 (GPS L1)
- **surface_spectrum** (double, 2D array of **nx_spec** x **ny_spec** elements): Spectrum of the surface. Default value: void
- **kx_spec** (double, array of **nx_spec** elements): Wavenumber's range of stored spectrum at upwind direction over the reflecting surface in meter^{-1}. Default value: void
- **ky_spec** (double, array of **ny_spec** elements): Wavenumber's range of stored spectrum at crosswind direction over the reflecting surface in meter^{-1}. Default value: void
- **nx_spec** (integer): Number of samples of stored spectrum at upwind direction over the reflecting surface. Default value: 0
- **ny_spec** (integer): Number of samples of stored spectrum at crosswind direction over the reflecting surface. Default value: 0
- **k_threshold** (double): Wavenumber's limit for the computation of MSS from the stored spectrum. Default value: $2\pi/3\lambda_{\mathrm{L1}}$
- **wind_U10_speed_grid** (double, 2D array of **size_lon_wgrid** x **size_lat_wgrid** elements): Grid of wind speeds at 10 meters above the surface in m/sec. Default value: void

- **wind_U10_azimuth_grid** (double, 2D array of **size_lon_wgrid** x **size_lat_wgrid** elements): Grid of wind azimuths at 10 meters above the surface in degrees. Default value: void
- **lon_wgrid** (double, array of **size_lon_wgrid** elements): Longitude values of wind grid in degrees. Default value: void
- **lat_wgrid** (double, array of **size_lat_wgrid** elements): Latitude values of wind grid in degrees. Default value: void
- **size_lon_wgrid** (integer): Number of samples of longitudes in stored wind grid. Default value: 0
- **size_lat_wgrid** (integer): Number of samples of latitudes in stored wind grid. Default value: 0
- **use_wind_grid** (boolean): True - A wind grid is stored, False - There is no wind grid stored. Default value: False

5.3 Functions

5.3.1 dump_parameters

Print relevant information about the object's content (public and private variables).

Example:

```
my_surface.dump_parameters()
```

5.3.2 set_frequency

Set frequency of the system in GHz.

Example:

```
my_surface.set_frequency(freq_GHz)
```

Input variables:

- **freq_GHz** (double): Frequency in GHz.

5.3.3 set_k_threshold

Set wavenumber's limit for the computation of MSS from the stored spectrum ($2\pi/3\lambda_{\mathrm{L1}}$ by default).

Example:

```
my_surface.set_k_threshold(k_lim_in)
```

Input variables:

- **k_lim_in** (double): Wavenumber's limit in meters^{-1}.

5.3.4 set_k_threshold_Brown

Set wavenumber's limit for the computation of MSS from the stored spectrum using [**Brown, 78**].

Example:

```
my_surface.set_k_threshold_Brown(incidence)
```

Input variables:

- **incidence** (double): Incidence angle of incoming signal in degrees.

5.3.5 get_k_threshold

Get wavenumber's limit internally stored.

Example:

```
k_lim_out = my_surface.get_k_threshold()
```

Output variables:

- **k_lim_out** (double): wavenumber's limit internally stored in meters^{-1}.

5.3.6 epsilon_sea_water

Set relative permittivity for sea water as a function of salinity and temperature.

Example:

```
my_surface.epsilon_sea_water(sal, temp)
```

Input variables:

- **sal** (double): Salinity of sea water in psu.
- **temp** (double): Temperature of sea water in C-degrees.

5.3.7 epsilon_sea_ice

Set relative permittivity of sea ice as a function of brine.

Example:

```
my_surface.epsilon_sea_ice(brine)
```

Input variables:

- **brine** (double): Brine volume of sea ice in 1/1000 units.

5.3.8 epsilon_dry_snow

Set relative permittivity of dry snow as a function of snow density.

Example:

```
my_surface.epsilon_dry_snow(density)
```

Input variables:

- **density** (double): Snow density of dry snow in gr/cm^3.

5.3.9 epsilon_wet_snow

Set relative permittivity of wet snow as a function of snow density and water volume.

Example:

```
my_surface.epsilon_wet_snow(density, water_vol)
```

Input variables:

- **density** (double): Snow density of wet snow in gr/cm^3.
- **water_vol** (double): Water volume of wet snow in %.

5.3.10 compute_Rfresnel_linear

Compute the reflection Fresnel coefficients for linear polarizations.

Example:

```
[rvv, rhh] = my_surface.compute_Rfresnel_linear(incidence, epsilon_up_layer)
```

Input variables:

- **incidence** (double): Incidence angle of incoming signal in degrees.
- **epsilon_up_layer** (double, array of 2 elements): Complex relative permittivity (real and imaginary parts) of layer above the reflecting surface. For air, simply set **epsilon_up_layer** = [1.0, 0.0].

Output variables:

- **rvv** (double, array of 2 elements): Complex reflection coefficient for vertical polarization (real and imaginary parts).
- **rhh** (double, array of 2 elements): Complex reflection coefficient for horizontal polarization (real and imaginary parts).

5.3.11 compute_Rfresnel_circular

Compute the reflection Fresnel coefficients for circular polarizations.

Example:

```
[rco, rcross] = my_surface.compute_Rfresnel_circular(incidence, epsilon_up_layer)
```

Input variables:

- **incidence** (double): Incidence angle of incoming signal in degrees.

- **epsilon_up_layer** (double, array of 2 elements): Complex relative permittivity (real and imaginary parts) of layer above the reflecting surface. For air, simply set **epsilon_up_layer** = [1.0, 0.0].

Output variables:

- **rco** (double, array of 2 elements): Complex reflection coefficient for co-polar [RHCP for GPS] (real and imaginary parts).
- **rcross** (double, array of 2 elements): Complex reflection coefficient for cross-polar [LHCP for GPS] (real and imaginary parts).

5.3.12 compute_Tfresnel_linear

Compute the transmission Fresnel coefficients for linear polarizations.

Example:

```
[tvv, thh] = my_surface.compute_Tfresnel_linear(incidence, epsilon_up_layer)
```

Input variables:

- **incidence** (double): Incidence angle of incoming signal in degrees.
- **epsilon_up_layer** (double, array of 2 elements): Complex relative permittivity (real and imaginary parts) of layer above the reflecting surface. For air, simply set **epsilon_up_layer** = [1.0, 0.0].

Output variables:

- **tvv** (double, array of 2 elements): Complex transmission coefficient for vertical polarization (real and imaginary parts).
- **thh** (double, array of 2 elements): Complex transmission coefficient for horizontal polarization (real and imaginary parts).

5.3.13 compute_Tfresnel_circular

Compute the transmission Fresnel coefficients for circular polarizations.

Example:

```
[tco, tcross] = my_surface.compute_Tfresnel_circular(incidence, epsilon_up_layer)
```

Input variables:

- **incidence** (double): Incidence angle of incoming signal in degrees.
- **epsilon_up_layer** (double, array of 2 elements): Complex relative permittivity (real and imaginary parts) of layer above the reflecting surface. For air, simply set **epsilon_up_layer** = [1.0, 0.0].

Output variables:

- **tco** (double, array of 2 elements): Complex transmission coefficient for co-polar [RHCP for GPS] (real and imaginary parts).
- **tcross** (double, array of 2 elements): Complex transmission coefficient for cross-polar [LHCP for GPS] (real and imaginary parts).

5.3.14 compute_sea_spectrum

Compute the sea spectrum based on [**Elfouhaily et al, 97**].

Example:

```
my_surface.compute_sea_spectrum(num_samples, delta_k, theta, omega)
```

Input variables:

- **num_samples** (integer): Number of samples of the generated spectrum in a single dimension.
- **delta_k** (double): Wavenumber's resolution of the generated spectrum in meter^{-1}.
- **theta** (double): Wind waves angle in degrees.
- **omega** (double): Wave age.

5.3.15 set_surf_spectrum

Set sea surface spectrum with a constant wavenumber's resolution.

Example:

```
my_surface.set_surf_spectrum(spectrum, kx_in, ky_in)
```

Input variables:

- **spectrum** (double, 2-D array of MxN elements): Sea surface spectrum.
- **kx_in** (double, array of M elements): Wavenumber's range of spectrum at upwind direction over the reflecting surface in meter^{-1} (a non-repetitive ascending sequence is required for a proper computation of MSS).
- **ky_in** (double, array of N elements): Wavenumber's range of spectrum at crosswind direction over the reflecting surface in meter^{-1} (a non-repetitive ascending sequence is required for a proper computation of MSS).

5.3.16 set_surf_spectrum_omnidir

Set omnidirectional sea surface with a constant wavenumber's resolution.

Example:

```
my_surface.set_surf_spectrum_omnidir(spectrum, kr_in)
```

Input variables:

- **spectrum** (double, array of N elements): Sea surface spectrum.
- **kr_in** (double, array of N elements): Wavenumber's range of omnidirectional spectrum in meter^{-1} (a non-repetitive ascending sequence is required for a proper computation of MSS).

5.3.17 get_surf_spectrum

Get value from stored sea surface spectrum as a function of grid's position.

Example:

```
[kx, ky, spec_val] = my_surface.get_surf_spectrum(x, y)
```

Input variables:

- **x** (integer): Sample index at upwind direction over the reflecting surface.
- **y** (integer): Sample index at crosswind direction over the reflecting surface.

Output variables:

- **kx** (double): Wavenumber at sample **x** in meter^{-1}.
- **ky** (double): Wavenumber at sample **y** in meter^{-1}.
- **spec_val** (double): Spectrum value at sample (**x**, **y**).

5.3.18 get_surf_spectrum_omnidir

Get value from stored sea surface spectrum as a function of array's position.

Example:

```
[kr, spec_val] = my_surface.get_surf_spectrum_omnidir(r)
```

Input variables:

- **r** (integer): Sample index in the omnidirectional array.

Output variables:

- **kr** (double): Wavenumber at sample **r** in meter^{-1}.
- **spec_val** (double): Spectrum value at sample **r**.

5.3.19 compute_mss_from_spectrum

Compute MSS from the stored spectrum.

Example:

```
my_surface.compute_mss_from_spectrum()
```

5.3.20 compute_mss_from_wind

Compute MSS from the wind speed parameters based on [**Katzberg et al, 2006**].

Example:

```
my_surface.compute_mss_from_wind()
```

5.3.21 set_wind_grid

Store an inhomogeneous wind grid as a function of latitude and longitude.

Example:

```
my_surface.set_wind_grid(wind_speed_grid, wind_azim_grid, longitudes,
latitudes)
```

Input variables:

- **wind_speed_grid** (double, 2-D array of MxN elements): Grid of wind speeds at 10 meters above the surface in m/sec.
- **wind_azim_grid** (double, 2-D array of MxN elements): Grid of wind azimuths at 10 meters above the surface in degrees.
- **longitudes** (double, array of M elements): Longitude values of wind grid in degrees.
- **latitudes** (double, array of N elements): Latitude values of wind grid in degrees.

5.3.22 interp_wind_grid

Interpolate wind speed and wind azimuth from stored wind grid and store the results obtained in public variables **wind_U10_speed** and **wind_U10_azimuth** respectively.

```
my_surface.interp_wind_grid(lon_in, lat_in)
```

Input variables:

- **lon_in** (double): Longitude coordinate for interpolation of the wind grid in degrees.
- **lat_in** (double): Latitude coordinate for interpolation of the wind grid in degrees.

5.3.23 disable_wind_grid

Remove stored wind grid.

Example:

```
my_surface.disable_wind_grid()
```

5.3.24 get_wind_grid_status

Return current status of stored wind grid.

Example:

```
status = my_surface.get_wind_grid_status()
```

Output variables:

- **status** (boolean): True - There is a wind grid stored, False - There is no wind grid stored.

6 ZaVoModel_GNSSR

This class models GNSS-R waveforms and DDM's based on [**Zavorotny and Voronovich, 00**] and their corresponding covariance matrix based on [**Li et al, 17**], which allows the simulation of realistic noise (thermal and speckle) realizations with a proper statistical characterization in both range and Doppler domains.

Example of object construction with arbitrary name *my_model*:

```
my_model = wavpy.ZaVoModel_GNSSR()
```

In this case, public variables (including objects) and functions from object *my_model* of class *ZaVoModel_GNSSR* can be respectively checked/modified or called with:

```
my_model.variable

my_model.function()

my_model.object_A.variable_from_object_A

my_model.object_A.function_from_object_A()
```

6.1 Additional information

Local frame: The local frame is a Cartesian coordinates system with its origin at the specular point, X- and Y-axis defining the horizontal plane parallel to the surface, with the Y-axis pointing towards the transmitter, and the Z-axis pointing to Zenith by complying the right-hand rule.

6.2 Public variables

- **interferometric_flag** (boolean): True - Computation of noise level based on the interferometric approach. False - Computation of noise level based on the clean-replica approach. Default value: False
- **curvature_approx_flag** (boolean): True - Apply Earth curvature approximation when integrating the reflected power over the surface (recommended when the receiver is at high altitude). False - Apply planar reflected surface. Default value: False
- **covariance_wav_flag** (boolean): True - Computation of mean waveform based on the covariance approach (the corresponding covariance matrix is also stored). False - Computation of mean waveform based on the regular approach (if **covariance_ddm_flag** is also at False). Default value: False
- **covariance_ddm_flag** (boolean): True - Computation of mean waveform and DDM based on the covariance approach (the corresponding covariance matrix is also stored). False - Computation of mean DDM based on the regular approach. Default value: False
- **recompute_Lambda_flag** (boolean): True - Computation of autocorrelation function from **gnss_signal** before simulation. False - Use of already stored autocorrelation function in **gnss_signal** during simulation. Default value: True
- **sigma0_to_one_flag** (boolean): True - Computation of mean waveform or DDM by setting *sigma_0* to 1. False - Computation of mean waveform or DDM based on the regular approach. Default value: False
- **polarization** (char): Polarization of the reflected signal. Valid values: 'R' for RHCP and 'L' for LHCP. Default value: 'L'
- **num_angles** (integer): Number of integration points over the surface ellipse for each range sample. It determines the angular resolution of the surface integral during the simulation. Default value: 120
- **wav_length** (integer): Length of modelled waveform. Default value: 256
- **ddm_half_dopplers** (integer): Number of additional Doppler slices. The total number of Doppler lines in the DDM will be: 2x **ddm_half_dopplers** + 1. Default value: 0

- **sampling_rate** (double): Sampling rate of the modelled waveform in samples/sec. Default value: 80000000.0
- **delta_doppler** (double): Doppler increment between consecutive slices of the DDM in Hz. Default value: 0.0
- **delta_freq** (double): Doppler offset of the DDM in Hz. Default value: 0.0
- **coherent_integration** (double): Coherent integration time of the modelled waveform in secs. Default value: 0.001
- **weight_cohPower** (double): Weight applied to the coherent component of the reflected power. Default value: 0.0
- **geometry** (object of class *Specular_geometry* [4]): Object characterizing the geometry used during the simulation.
- **surface** (object of class *Reflecting_surface* [5]): Object characterizing the reflecting surface used during the simulation.
- **receiver_Up** (object of class *RF_FrontEnd* [3]): Object characterizing the up-looking receiver front-end (collecting direct GNSS signals) used during the simulation.
- **receiver_Down** (object of class *RF_FrontEnd* [3]): Object characterizing the down-looking receiver front-end (collecting reflected GNSS signals) used during the simulation.
- **gnss_signal** (object of class *GNSS_composite* [2]): Object characterizing the GNSS signal used during the simulation.
- **waveform_POW** (object of class *Waveform_power* [8]): Object that stores the simulated waveform with functions for data analysis.

6.3 Relevant private variables

- **dump_isolines_data** (boolean): True - Store a binary file containing information of range, Doppler, σ_0 and gain of the antenna over the surface. False - Do not store surface data. Default value: False
- **stare_processing_flag** (boolean): True - Make simulation for a stare processing case, which will produce a DDM with a PDF of slopes set to

1. False - Make the simulation using the standard computation of the PDF of slopes. Default: False

- **use_transmitter_eirp** (boolean): True - Use **txr_eirp** for the computation of transmitted power. False - Standard computation of transmitted power by applying the code weights in **gnss_signal** to the nominal values of the different code-components. Default: False
- **double txr_eirp** (double): Equivalent Isotropic Radiated Power (EIRP) in dB units employed for the computation of transmitted power when **use_transmitter_eirp** is True. Default: 0.0
- **isolines_data_namefile** (string): Name of the binary output file containing the information of range, Doppler, σ_0 and gain of the antenna over the surface when making the simulation. Default value: void
- **stareProc_data_namefile** (string): Name of the text output file containing the information related to stare processing in an eight-columns format: range — Doppler — mean X-component of scattering vector — mean Y-component of scattering vector — mean Z-component of scattering vector — mean longitude — mean latitude — ambiguity status ("1" if there is no ambiguity, "0" else). Default value: void
- **size_ddm_stored** (integer, array of 2 elements): Number of samples of the stored DDM without taking into account the central Doppler. Default value: [0, 0]
- **ddm** (double, 2D array of **size_ddm_stored[0]** x **size_ddm_stored[1]** elements): Simulated power DDM. Default value: void
- **len_cov_stored** (integer): Number of samples (single dimension) of the stored covariance. Default value: 0
- **cov** (double, 2D array of **len_cov_stored** x **len_cov_stored** elements): Simulated waveform or DDM covariance. Default value: void

6.4 Functions

6.4.1 enable_isolines_data_dump

Enable the storage of a binary file containing information of range, Doppler, σ_0 and gain of the antenna over the surface during the simulation.

Example:

```
my_model.enable_isolines_data_dump(binfile)
```

Input variables:

- **binfile** (string): Name of the binary output file containing the information of range, Doppler, σ_0 and gain of the antenna over the surface when making the simulation.

Additional information:

The binary file contains blocks of 8 float numbers: [*x, y, lon, lat, tau, sigma_0, gain, doppler*]. The first two elements (*x, y*) are a point location at the local frame in meters, (*lon, lat*) are the longitude and latitude coordinates in degrees of the location, *tau* is the corresponding range in meters, *sigma_0* is the σ_0 in dB at that point, *gain* is the projection of the receiver antenna in dB and *doppler* contains the corresponding Doppler frequency in Hz.

The next lines show a python example on how to read the contents of file **binfile** and load them into a set of numpy arrays:

```
import numpy as np

data_surf = np.fromfile(binfile, dtype=np.float32)
x = np.zeros(len(data_surf)/8, dtype=np.float32)
y = np.zeros(len(data_surf)/8, dtype=np.float32)
lon = np.zeros(len(data_surf)/8, dtype=np.float32)
lat = np.zeros(len(data_surf)/8, dtype=np.float32)
tau = np.zeros(len(data_surf)/8, dtype=np.float32)
sigma_0 = np.zeros(len(data_surf)/8, dtype=np.float32)
gain = np.zeros(len(data_surf)/8, dtype=np.float32)
doppler = np.zeros(len(data_surf)/8, dtype=np.float32)

for index in range(len(data_surf)/8):
   x[index] = data_surf[index*8]
   y[index] = data_surf[index*8 + 1]
   lon[index] = data_surf[index*8 + 2]
   lat[index] = data_surf[index*8 + 3]
   tau[index] = data_surf[index*8 + 4]
```

```
sigma_0[index] = data_surf[index*8 + 5]
gain[index] = data_surf[index*8 + 6]
doppler[index] = data_surf[index*8 + 7]
```

6.4.2 disable_isolines_data_dump

Disable the storage of a binary file containing information of range, Doppler, σ_0 and gain of the antenna over the surface during the simulation.

Example:

```
my_model.disable_isolines_data_dump()
```

6.4.3 set_stare_processing_mode

Set simulation into stare processing mode. Under this coditions, the waveform/DDM is simulated by setting the PDF of the slopes to 1. In addition, a text file is stored with further information of the DDM process, such as mean scattering vector and ambiguity status of each delay-Doppler cell.

Example:

```
my_model.set_stare_processing_mode(textfile)
```

Input variables:

- **textfile** (string): Name of the text output file containing the information related to stare processing in an eight-columns format: (1) range, (2) Doppler, (3) mean X-component of scattering vector, (4) mean Y-component of scattering vector, (5) mean Z-component of scattering vector, (6) mean longitude, (7) mean latitude and (8) ambiguity status ("1" if there is no ambiguity, "0" else).

6.4.4 unset_stare_processing_mode

Unset the stare processing mode from the simulation.

Example:

```
my_model.unset_stare_processing_mode()
```

6.4.5 set_transmitter_EIRP

Use of a given value for the computation of transmitted power.

Example:

```
my_model.set_transmitter_EIRP(txr_eirp_in)
```

Input variables:

- **txr_eirp_in** (double): Equivalent Isotropic Radiated Power (EIRP) in dB units employed for the computation of transmitted power.

6.4.6 unset_transmitter_EIRP

Unset the use of a previously loaded value for the computation of transmitted power. Then, such computation is made by applying the code weights in **gnss_signal** to the nominal values of the different code-components.

Example:

```
my_model.unset_transmitter_EIRP()
```

6.4.7 compute_waveform

Simulate the waveform and the DDM (if **ddm_half_dopplers** and **delta_doppler** are greater than zero) for the given characterization.

Example:

```
my_model.compute_waveform()
```

6.4.8 get_DDM_doppler_slice

Get a Doppler slice from the stored DDM (if it has been computed).

Example:

```
ddm_slice = my_model.get_DDM_doppler_slice(doppler_index, range_size)
```

Input variables:

- **doppler_index** (integer): Index of the Doppler slice (0 for the central frequency).
- **range_size** (integer): Number of range samples of the stored DDM (it has to be equal to **size_ddm_stored[0]**).

Output variables:

- **ddm_slice** (double, array of **size_ddm_stored[0]** elements): Doppler slice from the stored DDM.

6.4.9 get_cov_slice

Get a covariance matrix slice from the stored covariance (if it has been computed).

Example:

```
cov_slice = my_model.get_cov_slice(cov_index, range_size)
```

Input variables:

- **cov_index** (integer): Index of the covariance slice.
- **range_size** (integer): Number of range samples of the stored covariance (it has to be equal to **len_cov_stored**). For a waveform covariance, this number should be equal to the number of range samples of the corresponding waveform. In the case of a DDM, the previous number should be additionally multiplied by the number of Doppler samples.

Output variables:

- **cov_slice** (double, array of **len_cov_stored** elements): Covariance slice from the stored covariance.

6.4.10 get_noisy_waveform

Get a noisy waveform computed from the stored covariance and the mean waveform.

Example:

```
wav_out = my_model.get_noisy_waveform(wav_len, seed_in)
```

Input variables:

- **wav_len** (integer): Number of samples of the stored mean waveform (it has to be equal to **wav_length**). Recommendation: use `my_model.waveform_POW.get_wav_length()` as **wav_len**.
- **seed_in** (unsigned long integer): Seed used for the internal random Gaussian noise generator.

Output variables:

- **wav_out** (double, array of wav_len elements): Power noisy waveform in arbitrary units.

6.4.11 get_noisy_DDM

Get a noisy DDM computed from the stored covariance and the mean DDM.

Example:

```
ddm_out = my_model.get_noisy_DDM(ddm_len, seed_in)
```

Input variables:

- **ddm_len** (integer): Number of samples of the stored mean DDM (it has to be equal to **size_ddm_stored[0]** multiplied by **size_ddm_stored[1] + 1**). Recommendation: use `my_model.waveform_POW.get_wav_length()*(my_model.ddm_half_dopplers*2 + 1)` as **ddm_len**.
- **seed_in** (unsigned long integer): Seed used for the internal random Gaussian noise generator.

Output variables:

- **ddm_out** (double, array of ddm_len elements): Power noisy DDM in arbitrary units. Recommendation: use `ddm_out.reshape((my_model.ddm_half_dopplers*2 + 1), my_model.wav_length)` to convert it into a 2-dimensional array.

7 MRSR_Model

This class models GNSS-R complex waveforms in a scenario with several reflecting layers. This model, referred as Multiple Ray - Single Reflection (MRSR), constructs the reflected signal as a complex sum of single coherent reflections coming from a set of layer interfaces (one reflection from each layer). Such approach was successfully tested in an Antarctic campaign [**Cardellach et al, 12**].

Example of object construction with arbitrary name *my_model_layers*:

```
my_model_layers = wavpy.MRSR_Model()
```

In this case, public variables (including objects) and functions from object *my_model_layers* of class *MRSR_Model* can be respectively checked/modified or called with:

```
my_model_layers.variable

my_model_layers.function()

my_model_layers.object_A.variable_from_object_A

my_model_layers.object_A.function_from_object_A()
```

7.1 Additional information

Surface frame: The surface frame is a Cartesian coordinates system with its origin at the vertical projection of the receiver's location towards the surface level, X- and Y-axis defining the horizontal plane parallel to the surface, with the X-axis pointing towards North, and the Z-axis pointing to Zenith by complying the right-hand rule.

7.2 Public variables

- **receiver** (object of class *RF_FrontEnd* [3]): Object characterizing the receiver front-end (collecting reflected GNSS signals) used during the simulation.
- **gnss_signal** (object of class *GNSS_composite* [2]): Object characterizing the GNSS signal used during the simulation.
- **waveforms** (object of class *Waveform_complex_cluster* [9]): Object that stores the simulated cluster of complex waveforms with functions for data analysis.

7.3 Relevant private variables

- **num_layers** (integer): Number of layers. Default value: 0
- **height_z** (integer): Height of the receiver with respect of the surface level (or its position in the Z-axis of the surface frame) in meters. Default value: 0
- **depth_layer** (double, array of **num_layers** elements): Initial depth level of the different layers at the Z-axis of the surface frame in meters. Note that positive values refer to negative values in the Z-axis' positions (e.g. 10 meters depth means -10 meters in the Z-axis of the surface frame). Default value: void
- **alpha_x** (double, array of **num_layers** elements): Rotation angle in the X-axis of the surface frame of the different layers in degrees (following right-hand rule with the surface frame). Default value: void
- **alpha_y** (double, array of **num_layers** elements): Rotation angle in the Y-axis of the surface frame of the different layers in degrees (following right-hand rule with the surface frame). Default value: void
- **epsilon_r** (double, array of **num_layers** elements): Real part of relative permittivity of the different layers without units. Default value: void
- **epsilon_i** (double, array of **num_layers** elements): Imaginary part of relative permittivity of the different layers without units. Default value: void

7.4 Functions

7.4.1 set_general_scenario

Characterize a multiple-layer scenario.

Example:

```
my_model_layers.set_general_scenario(height_in, depths_in, alpha_x_in,
alpha_y_in, epsilon_r_in, epsilon_i_in)
```

Input variables:

- **height_in** (double): Height of the receiver with respect of the surface level (or its position in the Z-axis of the surface frame) in meters.
- **depths_in** (double, array of N elements): Initial depth level of the different layers at the Z-axis of the surface frame in meters.
- **alpha_x_in** (double, array of N elements): Rotation angle in the X-axis of the surface frame of the different layers in degrees (following right-hand rule with the surface frame).
- **alpha_y_in** (double, array of N elements): Rotation angle in the Y-axis of the surface frame of the different layers in degrees (following right-hand rule with the surface frame).
- **epsilon_r_in** (double, array of N elements): Real part of relative permittivity of the different layers without units.
- **epsilon_i_in** (double, array of N elements): Imaginary part of relative permittivity of the different layers without units.

7.4.2 set_planar_layers_scenario

Characterize a multiple-layer scenario where all layers are parallel to the XY-plane of the surface frame.

Example:

```
my_model_layers.set_planar_layers_scenario(height_in, depths_in, epsilon_r_in,
epsilon_i_in)
```

Input variables:

- **height_in** (double): Height of the receiver with respect of the surface level (or its position in the Z-axis of the surface frame) in meters.
- **depths_in** (double, array of N elements): Initial depth level of the different layers at the Z-axis of the surface frame in meters.
- **epsilon_r_in** (double, array of N elements): Real part of relative permittivity of the different layers without units.
- **epsilon_i_in** (double, array of N elements): Imaginary part of relative permittivity of the different layers without units.

7.4.3 set_dry_snow_planar_layers_scenario

Characterize a dry snow multiple-layer scenario where all layers are parallel to the XY-plane of the surface frame.

Example:

```
my_model_layers.set_dry_snow_planar_layers_scenario(height_in, depths_in,
snow_dens_in)
```

Input variables:

- **height_in** (double): Height of the receiver with respect of the surface level (or its position in the Z-axis of the surface frame) in meters.
- **depths_in** (double, array of N elements): Initial depth level of the different layers at the Z-axis of the surface frame in meters.
- **snow_dens_in** (double, array of N elements): Snow density of the different layers in gr/cm^3 units.

7.4.4 mod_height_depths

Modify the height of the receiver and the depths of the layers in the already characterized scenario.

Example:

```
my_model_layers.mod_height_depths(height_in, depths_in)
```

Input variables:

- **height_in** (double): Height of the receiver with respect of the surface level (or its position in the Z-axis of the surface frame) in meters.
- **depths_in** (double, array of **num_layers** elements): Initial depth level of the different layers at the Z-axis of the surface frame in meters.

7.4.5 mod_alphas

Modify the rotation angles of the layers in the already characterized scenario.

Example:

```
my_model_layers.mod_alphas(alpha_x_in, alpha_y_in)
```

Input variables:

- **alpha_x_in** (double, array of N elements): Rotation angle in the X-axis of the surface frame of the different layers in degrees (following right-hand rule with the surface frame).
- **alpha_y_in** (double, array of N elements): Rotation angle in the Y-axis of the surface frame of the different layers in degrees (following right-hand rule with the surface frame).

7.4.6 mod_epsilon

Modify the permittivity of the layers in the already characterized scenario.

Example:

```
my_model_layers.mod_epsilon(epsilon_r_in, epsilon_i_in)
```

Input variables:

- **epsilon_r_in** (double, array of **num_layers** elements): Real part of relative permittivity of the different layers without units.
- **epsilon_i_in** (double, array of **num_layers** elements): Imaginary part of relative permittivity of the different layers without units.

7.4.7 compute_GNSS_wavcluster

Simulate a cluster of complex waveforms for the characterized scenario.

Example:

```
my_model_layers.compute_GNSS_wavcluster(wav_lags, lag_direct_pos, sampling_rate,
elevations, azimuths)
```

Input variables:

- **wav_lags** (integer): Size of the waveforms stored in the simulated cluster.
- **lag_direct_pos** (integer): Lag position corresponding to the direct signal's peak in the simulated waveform's range.
- **sampling_rate** (double): Sampling rate of the waveforms from the simulated cluster in samples/sec.
- **elevations** (double, array of N elements): Elevation of the incident GNSS signal at the surface level for each sample of the simulated cluster in degrees.
- **azimuths** (double, array of N elements): Azimuth of the incident GNSS signal at the surface level for each sample of the simulated cluster in degrees.

7.4.8 compute_LH_freqs_and_depths

Compute the relationship between interferometric frequency and depth in the resultant lag-hologram (obtained from a simulated waveforms cluster) for the characterized scenario.

Example:

```
[freq_LH, depth_LH] = my_model_layers.compute_LH_freqs_and_depths(elev_range,
azim_range, time_range, samples_freq_LH, samples_depth_LH)
```

Input variables:

- **elev_range** (double, array of 2 elements): Elevation range (initial and final values) of the incident GNSS signal at the surface level for the simulated cluster in degrees.

- **azim_range** (double, array of 2 elements): Azimuth range (initial and final values) of the incident GNSS signal at the surface level for the simulated cluster in degrees.
- **time_range** (double, array of 2 elements): Time range (initial and final values) of the simulated cluster in seconds.
- **samples_freq_LH** (integer): Number of frequency samples of the resultant lag-hologram.
- **samples_depth_LH** (integer): Number of depths samples of the resultant lag-hologram (it has to be equal than **samples_freq_LH**).

Output variables:

- **freq_LH** (double, array of **samples_freq_LH** elements): Interferometric frequencies (in cycles/degree of elevation units) of the resultant lag-hologram (obtained from a simulated waveforms cluster) for the characterized scenario.
- **depth_LH** (double, array of **samples_freq_LH** elements): Depths (in meters) of the resultant lag-hologram (obtained from a simulated waveforms cluster) for the characterized scenario.

7.4.9 compute_pow_linearPol

Compute the relative received power (not waveforms) assuming an incoming signal (with power = 1 at the surface level) at a given frequency and linear polarizations for the characterized scenario.

Example:

```
[pow_H, pow_V] = my_model_layers.compute_pow_linearPol(elevation,
azimuth, freq)
```

Input variables:

- **elevation** (double): Elevation of the incident signal at the surface level in degrees.
- **azimuth** (double): Azimuth of the incident signal at the surface level in degrees.
- **freq** (double): Frequency of the incident signal in Hz.

Output variables:

- **pow_H** (double): Relative received power at horizontal polarization.
- **pow_V** (double): Relative received power at vertical polarization.

8 Waveform_power

This class provides means to analyze and characterize power (real values) GNSS+R waveforms.

Example of object construction with arbitrary name *my_wav*:

```
my_wav = wavpy.Waveform_power()
```

In this case, public variables and functions from object *my_wav* of class *Waveform_power* can be respectively checked/modified or called with:

```
my_wav.variable

my_wav.function()
```

8.1 Public variables

- **positionMax** (double): Range position of waveform's peak (computed by means of a linear fit at the first waveform's derivative) in meters. Default value: 0.0

- **posSampleMax** (double): Range position of waveform's peak (computed by taking the maximum sample value of the interpolated waveform) in meters. Default value: 0.0

- **sigma_posMax** (double): Formal standard deviation of the position of the waveform's peak (as a result of the linear fir applied) in meters. Default value: -1.0

- **powerMax** (double): Waveform's peak power (computed by taking the maximum sample value of the interpolated waveform) in the stored waveform's units. Default value: 0.0

- **positionDer** (double): Range position of waveform's peak derivative (computed by means of a linear fit at the second waveform's derivative) in meters. Default value: 0.0
- **posSampleDer** (double): Range position of waveform's peak derivative (computed by taking the maximum sample value of the interpolated waveform's first derivative) in meters. Default value: 0.0
- **sigma_posDer** (double): Formal standard deviation of the position of the waveform's peak derivative (as a result of the linear fir applied) in meters. Default value: -1.0
- **power_posDer** (double): Waveform's power at position of maximum's first derivative (computed by taking the corresponding sample value of the interpolated waveform) in the stored waveform's units. Default value: 0.0
- **powerDer_posDer** (double): Waveform's first derivative power at position of maximum's first derivative (computed by taking the corresponding sample value of the interpolated waveform) in the stored waveform's units divided by meters. Default value: 0.0
- **floorNoise** (double): Waveform's noise level (computed by averaging the first samples) in the stored waveform's units. Default value: 0.0
- **positionRel** (double): Range position of waveform's peak multiplied by **rel_factor** at the leading edge (computed by means of a linear fit at the waveform) in meters. Default value: 0.0
- **posSampleRel** (double): Range position of waveform's peak multiplied by **rel_factor** at the leading edge (computed by taking the corresponding sample value of the interpolated waveform) in meters. Default value: 0.0
- **sigma_posRel** (double): Formal standard deviation of the position of waveform's peak multiplied by **rel_factor** at the leading edge (as a result of the linear fir applied) in meters. Default value: -1.0
- **slope_normTail** (double): Slope of the trailing edge of the normalized waveform (computed by means of a linear fit at the waveform) in meters^{-1}. Default value: 0.0
- **sigma_slope_normTail** (double): Formal standard deviation of the slope of the trailing edge of the normalized waveform (as a result of the linear fir applied) in meters^{-1}. Default value: -1.0

8.2 Relevant private variables

- **waveform** (double, array of **wav_length** elements): Stored power waveform in the given units. Default value: void
- **wav_length** (integer): Number of samples of stored waveform. Default value: 0
- **tail_factor** (integer): Integer factor (to be multiplied by **wav_length**) used to compute the extended length of the waveform for connecting both ends before FFT-based interpolation. Default value: 1
- **noise_lags** (integer): Number of lags, starting from zero, employed to compute the noise floor. Default value: 0
- **tail_lags** (integer): Number of lags employed to extrapolate the tail of the waveform until the noise floor. Default value: 0
- **sampling_rate** (double): Sampling rate of the waveform in samples/sec. Default value: 20000000.0
- **rel_factor** (double): Scaling factor used for the computation of the relative delay located at the leading edge with a power equal to the product of such factor with the peak of the waveform. Default value: 0.5
- **init_range** (double): Initial range value of the stored waveform in meters. Default value: 0.0
- **min_resolution_fft_interp** (double): Minimum range resolution of the FFT interpolation for the computation of the delays in meters. Default value: 0.15
- **fit_length** (double): Range length of the segment employed for the linear fitting when computing the delays in meters. Default value: 10.0
- **normTail_length** (double): Range length of the segment employed for the linear fitting when computing the slope of the trailing edge in meters. Default value: 50.0

8.3 Functions

8.3.1 set_waveform

Set a power waveform.

Example:

```
my_wav.set_waveform(wav_in)
```

Input variables:

- **wav_in** (double, array of N elements): Power waveform in arbitrary units.

8.3.2 set_float_waveform

Set a power waveform of type float.

Example:

```
my_wav.set_float_waveform(wav_in)
```

Input variables:

- **wav_in** (double, array of N elements): Power waveform in arbitrary units.

8.3.3 set_norm_waveform

Set a normalized power waveform of type float.

Example:

```
my_wav.set_norm_waveform(norm_wav_in, max_val)
```

Input variables:

- **norm_wav_in** (double, array of N elements): Normalized power waveform.
- **max_val** (double): Maximum value of the corresponding de-normalized waveform in arbitrary units.

8.3.4 set_amp_waveform

Set a power waveform from in-phase and quadrature amplitude components.

Example:

```
my_wav.set_amp_waveform(wav_i_in, wav_q_in)
```

Input variables:

- **wav_i_in** (double, array of N elements): Amplitude waveform (in-phase component) in arbitrary units.
- **wav_q_in** (double, array of N elements): Amplitude waveform (quadrature component) in arbitrary units.

8.3.5 get_waveform

Get the stored power waveform.

Example:

```
wav_out = my_wav.get_waveform(wav_len)
```

Input variables:

- **wav_len** (integer): Number of samples of the stored waveform (it has to be equal to **wav_length**). Recommendation: use `my_wav.get_wav_length()` as **wav_len**.

Output variables:

- **wav_out** (double, array of **wav_len** elements): Power waveform in arbitrary units.

8.3.6 add_waveform_retracking

Update the stored waveform by weighty averaging its values with an input waveform after being re-tracked a given delay (by means of FFT).

Example:

```
my_wav.add_waveform_retracking(wav_in, retrack_delay, wav_weight)
```

Input variables:

- **wav_in** (double, array of N elements): Power waveform in arbitrary units.
- **retrack_delay** (double): Retracking delay applied to **wav_in** before being averaged with the stored waveform in meters.

- **wav_weight** (double): Weight applied to **wav_in** for its averaging with the stored waveform.

A more clear explanation:

new_stored_waveform[range] = (1.0 - **wav_weight**) * **waveform**[range] + **wav_weight** * **wav_in**[range - **retrack_delay**]

8.3.7 set_sampling_rate

Set the sampling rate of the stored waveform.

Example:

```
my_wav.set_sampling_rate(sampling_rate_in)
```

Input variables:

- **sampling_rate_in** (double): Sampling rate of the waveform in samples/sec.

8.3.8 set_rel_factor

Set the scaling factor used for the computation of the relative delay located at the leading edge with a power equal to the product of such factor with the peak of the waveform.

Example:

```
my_wav.set_rel_factor(factor_in)
```

Input variables:

- **factor_in** (double): Scaling factor used for the computation of the relative delay located at the leading edge with a power equal to the product of such factor with the peak of the waveform.

8.3.9 get_rel_factor

Get the scaling factor used for the computation of the relative delay located at the leading edge with a power equal to the product of such factor with the peak of the waveform.

Example:

```
factor_out = my_wav.get_rel_factor()
```

Output variables:

- **factor_out** (double): Scaling factor used for the computation of the relative delay located at the leading edge with a power equal to the product of such factor with the peak of the waveform.

8.3.10 set_min_resolution_fft_interp

Set the minimum range resolution of the FFT interpolation for the computation of the delays.

Example:

```
my_wav.set_min_resolution_fft_interp(resolution_in)
```

Input variables:

- **resolution_in** (double): Minimum range resolution of the FFT interpolation for the computation of the delays in meters.

8.3.11 set_fit_length

Set the range length of the segment employed for the linear fitting when computing the delays.

Example:

```
my_wav.set_fit_length(fit_length_in)
```

Input variables:

- **fit_length_in** (double): Range length of the segment employed for the linear fitting when computing the delays in meters.

8.3.12 set_normtail_length

Set range length of the segment employed for the linear fitting when computing the slope of the trailing edge in meters.

Example:

```
my_wav.set_normtail_length(normtail_length_in)
```

Input variables:

- **normtail_length_in** (double): Range length of the segment employed for the linear fitting when computing the slope of the trailing edge in meters.

8.3.13 set_tail_factor

Set factor to increase waveform's length with the purpose of connecting both ends during computation of delays.

Example:

```
my_wav.set_tail_factor(tail_factor_in)
```

Input variables:

- **tail_factor_in** (integer): Integer factor (to be multiplied by **wav_length**) used to compute the extended length of the waveform for connecting both ends before FFT-based interpolation.

8.3.14 set_noise_lags

Set the number of lags employed to compute the noise floor.

Example:

```
my_wav.set_noise_lags(noise_lags_in)
```

Input variables:

- **noise_lags_in** (integer): Number of lags, starting from zero, employed to compute the noise floor.

8.3.15 set_tail_lags

Set the number of lags employed to extrapolate the tail of the waveform until the noise floor.

Example:

```
my_wav.set_tail_lags(tail_lags_in)
```

Input variables:

- **tail_lags_in** (integer): Number of lags employed to extrapolate the tail of the waveform until the noise floor.

8.3.16 compute_delays

Compute the delays that characterize the stored waveform based on [**Rius et al, 10**]. It updates the contents of all the public variables.

Example:

```
my_wav.compute_delays()
```

8.3.17 compute_delays_wspeckle

Compute the delays that characterize the stored waveform based on [**Rius et al, 10**]. It updates the contents of all the public variables. The difference with `compute_delays()` is that in this case the standard deviations resulting from the linear fits are computed with weights for each sample.

Example:

```
my_wav.compute_delays_wspeckle(num_incoh)
```

Input variables:

- **num_incoh** (integer): Number of incoherent integration samples.

8.3.18 compute_delays_wlimits

Compute the delays that characterize the stored waveform based on [**Rius et al, 10**] after applying range limits. It updates the contents of all the public variables. This function is recommended for those cases where the waveform shows unexpected fluctuations or has complex autocorrelation function shapes (such as Galileo signals).

Example:

```
my_wav.compute_delays_wlimits(limits_center, limits_width, apriori_scattdel)
```

Input variables:

- **limits_center** (double): Central range location for limiting the search of the waveform's maximum derivative in meters.
- **limits_width** (double): Range interval around **limits_center** and **limits_center + apriori_scattdel** for limiting the search of waveform's maximum derivative and waveform's peak respectively. Meter units.
- **apriori_scattdel** (double): A-priori estimation of the range distance between waveform's maximum derivative and waveform's peak used for limiting the search of the position of the peak. Meter units.

8.3.19 compute_delays_wlimits_LPF

Compute the delays that characterize the stored waveform based on [**Rius et al, 10**] after applying range limits and a box low-pass filter. It updates the contents of all the public variables. This function is recommended for those cases where the waveform shows unexpected fluctuations or has complex autocorrelation function shapes (such as Galileo signals).

Example:

```
my_wav.compute_delays_wlimits_LPF(limits_center, limits_width, apriori_scattdel,
bandwidth_LPF_Hz)
```

Input variables:

- **limits_center** (double): Central range location for limiting the search of the waveform's maximum derivative in meters.
- **limits_width** (double): Range interval around **limits_center** and **limits_center + apriori_scattdel** for limiting the search of waveform's maximum derivative and waveform's peak respectively. Meter units.

- **apriori_scattdel** (double): A-priori estimation of the range distance between waveform's maximum derivative and waveform's peak used for limiting the search of the position of the peak. Meter units.
- **bandwidth_LPF_Hz** (double): Bandwidth of the low-pass filter applied. Hz units.

8.3.20 set_init_range

Set the initial range value of the stored waveform.

Example:

```
my_wav.set_init_range(init_range_in)
```

Input variables:

- **init_range_in** (double): Initial range value of the stored waveform in meters.

8.3.21 get_range_waveform

Get the range of the stored waveform.

Example:

```
range_wav = my_wav.get_range_waveform(range_len)
```

Input variables:

- **range_len** (integer): Number of samples of the waveform stored (it has to be equal to **wav_length**). Recommendation: use `my_wav.get_wav_length()` as **range_len**.

Output variables:

- **range_wav** (integer, array of **range_len** elements): Range of the stored waveform in meters.

8.3.22 get_size_deriv_waveform_tracks

Get size (number of samples) of the intervals where the fits are applied on first and second waveform's derivatives.

Example:

```
size_deriv = my_wav.get_size_deriv_waveform_tracks()
```

Output variables:

- **size_deriv** (integer): Number of samples of the intervals where the fits are applied on first and second waveform's derivatives.

8.3.23 get_deriv_waveform_tracks

Get the intervals where the fits are applied (around specular delay) from waveform and its first and second derivatives after computing relevant delay positions and power levels.

Example:

```
[tau_der, wav_der, wav_der1, wav_der2] = my_wav.get_deriv_waveform_tracks(size_tau_de
size_wav_der, size_wav_der1, size_wav_der2)
```

Input variables:

- **size_tau_der** (integer): Number of samples of range interval (it has to be equal to **size_deriv**). Recommendation: use `my_wav.get_size_deriv_waveform_tracks()` as **size_tau_der**.

- **size_wav_der** (integer): Number of samples of waveform interval (it has to be equal to **size_deriv**). Recommendation: use `my_wav.get_size_deriv_waveform_tracks()` as **size_wav_der**.

- **size_wav_der1** (integer): Number of samples of waveform's first derivative interval (it has to be equal to **size_deriv**). Recommendation: use `my_wav.get_size_deriv_waveform_tracks()` as **size_wav_der1**.

- **size_wav_der2** (integer): Number of samples of waveform's second derivative interval (it has to be equal to **size_deriv**). Recommendation: use `my_wav.get_size_deriv_waveform_tracks()` as **size_wav_der2**.

Output variables:

- **tau_der** (double, array of **size_deriv** elements): Range of the waveform and its first and second derivatives for the interval around the specular delay.
- **wav_der** (double, array of **size_deriv** elements): Waveform interval around the specular delay where the fits are applied.
- **wav_der1** (double, array of **size_deriv** elements): Waveform's first derivative interval around the specular delay where the fits are applied.
- **wav_der2** (double, array of **size_deriv** elements): Waveform's second derivative interval around the specular delay where the fits are applied.

8.3.24 get_deriv_waveform_tracks_wlimits

Get the intervals where the fits are applied (around specular delay) from waveform and its first and second derivatives after computing relevant delay positions and power levels after applying limits.

Example:

```
[tau_der, wav_der, wav_der1, wav_der2] = my_wav.get_deriv_waveform_tracks_wlimits(lim
limits_width, apriori_scattdel, size_tau_der, size_wav_der, size_wav_der1,
size_wav_der2)
```

Input variables:

- **limits_center** (double): Center location of the range interval to constraint the initial estimation of the specular delay in meters.
- **limits_width** (double): Range interval around **limits_center** to constraint the initial estimation of the specular delay in meters.
- **apriori_scattdel** (double): A-priori estimation of the scatterometric delay (delay distance from specular to peak of the waveform) in meters.
- **size_tau_der** (integer): Number of samples of range interval (it has to be equal to **size_deriv**). Recommendation: use `my_wav.get_size_deriv_waveform_tracks()` as **size_tau_der**.

- **size_wav_der** (integer): Number of samples of waveform interval (it has to be equal to **size_deriv**). Recommendation: use `my_wav.get_size_deriv_waveform_tracks()` as **size_wav_der**.
- **size_wav_der1** (integer): Number of samples of waveform's first derivative interval (it has to be equal to **size_deriv**). Recommendation: use `my_wav.get_size_deriv_waveform_tracks()` as **size_wav_der1**.
- **size_wav_der2** (integer): Number of samples of waveform's second derivative interval (it has to be equal to **size_deriv**). Recommendation: use `my_wav.get_size_deriv_waveform_tracks()` as **size_wav_der2**.

Output variables:

- **tau_der** (double, array of **size_deriv** elements): Range of the waveform and its first and second derivatives for the interval around the specular delay.
- **wav_der** (double, array of **size_deriv** elements): Waveform interval around the specular delay where the fits are applied.
- **wav_der1** (double, array of **size_deriv** elements): Waveform's first derivative interval around the specular delay where the fits are applied.
- **wav_der2** (double, array of **size_deriv** elements): Waveform's second derivative interval around the specular delay where the fits are applied.

8.3.25 dump_norm_waveform

Print the normalized power waveform stored.

Example:

```
my_wav.dump_norm_waveform()
```

8.3.26 dump_delays

Print the information of the delays computed from the power waveform stored.

Example:

```
my_wav.dump_delays()
```

8.3.27 get_wav_length

Get the number of samples of the waveform stored.

Example:

```
wav_len_out = my_wav.get_wav_length()
```

Output variables:

- **wav_len_out** (integer): Number of samples of the waveform stored.

9 Waveform_complex_cluster

This class provides means to analyze and characterize time series of complex GNSS+R waveforms.

Example of object construction with arbitrary name *my_wavcluster*:

```
my_wavcluster = wavpy.Waveform_complex_cluster()
```

In this case, public variables and functions from object *my_wavcluster* of class *Waveform_complex_cluster* can be respectively checked/modified or called with:

```
my_wavcluster.variable
```

```
my_wavcluster.function()
```

9.1 Public variables

- **num_valid_wavs** (integer): Number of valid waveforms stored in the cluster. Default value: 0
- **num_phasor_iter** (integer): Number of operations made with the stored phasor. Default value: 0

9.2 Relevant private variables

- **Icomponents** (double, 2D array of **cluster_length** x **wav_length** elements): In-phase components of waveforms stored in the cluster. Default value: void
- **Qcomponents** (double, 2D array of **cluster_length** x **wav_length** elements): Quadrature components of waveforms stored in the cluster. Default value: void

- **valid_wavs** (boolean, array of **cluster_length** elements): Boolean array indicating the validity of the stored complex waveforms in the cluster. Default value: void
- **phasorI** (double, array of **cluster_length** elements): In-phase component of a complex phasor stored in the cluster. Default value: void
- **phasorQ** (double, array of **cluster_length** elements): Quadrature component of a complex phasor stored in the cluster. Default value: void
- **valid_phasor** (boolean, array of **cluster_length** elements): Boolean array indicating the validity of the stored complex phasor in the cluster. Default value: void
- **wav_length** (integer): Size of the waveforms stored in the cluster. Default value: 0
- **cluster_length** (integer): Length of the cluster. Default value: 0

9.3 Functions

9.3.1 initialize

Initialize the size of the cluster to allocate the required memory.

Example:

```
my_wavcluster.initialize(in_cluster_length, wav_in_length)
```

Input variables:

- **in_cluster_length** (integer): Length of the cluster.
- **wav_in_length** (integer): Length of the waveforms to be stored in the cluster.

9.3.2 add_waveform

Add a complex waveform to the cluster at a given position.

Example:

```
my_wavcluster.add_waveform(Icomp_in, Qcomp_in, cluster_pos)
```

Input variables:

- **Icomp_in** (double, array of **wav_length** elements): In-phase components of complex waveform in arbitrary units.
- **Qcomp_in** (double, array of **wav_length** elements): Quadrature components of complex waveform in arbitrary units.
- **cluster_pos** (integer): Position to store the complex waveform inside the cluster (from 0 to **cluster_length** - 1).

9.3.3 add_waveform_scale

Add a complex waveform to the cluster at a given position and multiplied by a given scaling factor.

Example:

```
my_wavcluster.add_waveform_scale(Icomp_in, Qcomp_in, cluster_pos,
scale_factor)
```

Input variables:

- **Icomp_in** (double, array of **wav_length** elements): In-phase components of complex waveform in arbitrary units.
- **Qcomp_in** (double, array of **wav_length** elements): Quadrature components of complex waveform in arbitrary units.
- **cluster_pos** (integer): Position to store the complex waveform inside the cluster (from 0 to **cluster_length** - 1).
- **scale_factor** (double): Scaling factor applied to the input complex waveform before being stored in the cluster.

9.3.4 add_waveform_GOLD

Add a complex waveform with the format of IEEC's GOLD-RTR dataset to the cluster at a given position.

Example:

```
my_wavcluster.add_waveform_GOLD(Icomponents_in, Qcomponents_in, cluster_pos)
```

Input variables:

- **Icomponents_in** (signed char, array of **wav_length** elements): In-phase components of complex waveform in arbitrary units.
- **Qcomponents_in** (signed char, array of **wav_length** elements): Quadrature components of complex waveform in arbitrary units.
- **cluster_pos** (integer): Position to store the complex waveform inside the cluster (from 0 to **cluster_length** - 1).

9.3.5 add_waveform_PIR

Add a complex waveform with the format of IEEC's PIR dataset to the cluster at a given position.

Example:

```
my_wavcluster.add_waveform_PIR(XiYi, XqYq, XiYq, XqYi, cluster_pos)
```

Input variables:

- **XiYi** (short integer, array of **wav_length** elements): Correlation between in-phase components of direct and reflected signals in arbitrary units.
- **XqYq** (short integer, array of **wav_length** elements): Correlation between quadrature components of direct and reflected signals in arbitrary units.
- **XiYq** (short integer, array of **wav_length** elements): Correlation between in-phase component of direct signal and quadrature component of reflected signal in arbitrary units.
- **XqYi** (short integer, array of **wav_length** elements): Correlation between quadrature component of direct signal and in-phase component of reflected signal in arbitrary units.
- **cluster_pos** (integer): Position to store the complex waveform inside the cluster (from 0 to **cluster_length** - 1).

9.3.6 load_ITF_waveforms_SPIR

Load a cluster of 999 complex interferometric waveforms from a 1-second IEEC's SPIR binary file.

Example:

```
start_window_delay = my_wavcluster.load_ITF_waveforms_SPIR(namefile,
peak_delay_estimate, BF_phases_UP, BF_phases_DW, filter_num)
```

Input variables:

- **namefile** (string): Filename of SPIR 1-second binary file.
- **peak_delay_estimate** (double): Raw estimation of the range delay of the waveform's peak in meters.
- **BF_phases_UP** (double, array of 8 elements): Beamformer phases (in degrees) to be applied at each of the 8 up-looking antenna elements of the SPIR setup to get maximum antenna gain towards a desired direction.
- **BF_phases_DW** (double, array of 8 elements): Beamformer phases (in degrees) to be applied at each of the 8 down-looking antenna elements of the SPIR setup to get maximum antenna gain towards a desired direction.
- **filter_num** (integer): Code to select the frequency-domain filter to be applied during the processing. 1 - Galileo at E1, 2 - GPS/Galileo at L5, other - GPS at L1 removing the C/A code band.

Output variables:

- **start_window_delay** (double): Range delay of the first waveform sample in meters.

9.3.7 load_ITF_waveforms_SPIR_selected_signals

Load a cluster of 999 complex interferometric waveforms from a 1-second IEEC's SPIR binary file by selecting a set of input signals (16 available).

Example:

```
start_window_delay = my_wavcluster.load_ITF_waveforms_SPIR_selected_signals(namefile,
peak_delay_estimate, BF_phases, signal_elements_1, signal_elements_2,
```

```
filter_num)
```

Input variables:

- **namefile** (string): Filename of SPIR 1-second binary file.
- **peak_delay_estimate** (double): Raw estimation of the range delay of the waveform's peak in meters.
- **BF_phases** (double, array of 16 elements): Beamformer phases (in degrees) to be applied at each of the 16 antenna elements of the SPIR setup to get maximum antenna gain towards a desired direction. Note that only those elements indicated on input variables **signal_elements_1** and **signal_elements_2** will be employed.
- **signal_elements_1** (signed char, array of 16 elements): Array indicator of which antenna elements (those indexes with a value equal to '1') will be combined to generate signal 1 (to be cross-correlated against signal 2).
- **signal_elements_2** (signed char, array of 16 elements): Array indicator of which antenna elements (those indexes with a value equal to '1') will be combined to generate signal 2 (to be cross-correlated against signal 1).
- **filter_num** (integer): Code to select the frequency-domain filter to be applied during the processing. 1 - Galileo at E1, 2 - GPS/Galileo at L5, other - GPS at L1 removing the C/A code band.

Output variables:

- **start_window_delay** (double): Range delay of the first waveform sample in meters.

9.3.8 load_CR_waveforms_SPIR

Load a cluster of 999 complex GPS L1-CA clean-replica waveforms from a 1-second IEEC's SPIR binary file.

Example:

```
start_window_delay = my_wavcluster.load_CR_waveforms_SPIR(namefile,
peak_delay_estimate, doppler_estimate, delta_doppler, BF_phases, uplooking_channel,
```

```
code_ref)
```

Input variables:

- **namefile** (string): Filename of SPIR 1-second binary file.
- **peak_delay_estimate** (double): Raw estimation of the range delay of the waveform's peak in meters.
- **doppler_estimate** (double): Estimation of the Doppler frequency of the direct signal in Hz (also required when computing waveforms from reflected signals).
- **delta_doppler** (double): Estimation of the Doppler frequency difference of the reflected signal with respect to the direct one in Hz (set to zero in case of computing waveforms from direct signals).
- **BF_phases** (double, array of 16 elements): Beamformer phases (in degrees) to be applied at each of the 16 antenna elements of the SPIR setup to get maximum antenna gain towards a desired direction.
- **uplooking_channel** (boolean): True - Cross-correlation against direct signals using up-looking antenna elements. False - Cross-correlation against direct or reflected signals using down-looking antenna elements.
- **code_ref** (integer): GPS PRN number.

Output variables:

- **start_window_delay** (double): Range delay of the first waveform sample in meters.

9.3.9 load_CR_waveforms_SPIR_selected_signals

Load a cluster of 999 complex GPS L1-CA clean-replica waveforms from a 1-second IEEC's SPIR binary file by selecting a set of input signals (16 available).

Example:

```
start_window_delay = my_wavcluster.load_CR_waveforms_SPIR_selected_signals(namefile,
peak_delay_estimate, doppler_estimate, assist_doppler, BF_phases, signal_elements,
code_ref)
```

Input variables:

- **namefile** (string): Filename of SPIR 1-second binary file.
- **peak_delay_estimate** (double): Raw estimation of the range delay of the waveform's peak in meters.
- **doppler_estimate** (double): Estimation of the Doppler frequency of the direct signal in Hz (also required when computing waveforms from reflected signals).
- **assist_doppler** (boolean): True - Enable search of Doppler frequency (having **doppler_estimate** as initial reference). False - Disable search of Doppler frequency (rely on **doppler_estimate**).
- **BF_phases** (double, array of 16 elements): Beamformer phases (in degrees) to be applied at each of the 16 antenna elements of the SPIR setup to get maximum antenna gain towards a desired direction. Note that only those elements indicated on input variable **signal_elements** will be employed.
- **signal_elements** (signed char, array of 16 elements): Array indicator of which antenna elements (those indexes with a value equal to '1') will be combined to generate the signal.
- **code_ref** (integer): GPS PRN number.

Output variables:

- **start_window_delay** (double): Range delay of the first waveform sample in meters.

9.3.10 searching_CR_waveforms_SPIR

Search of GPS L1-CA clean-replica signals from a 1-second IEEC's SPIR binary file by selecting a set of input signals (16 available). Basically, it scans the Doppler domain cross-correlating against a clean-replica model of each PRN in order to get the maximum response. Then, it prints on the screen the corresponding Doppler frequency and peak-to-noise ratio values obtained.

Example:

```
my_wavcluster.searching_CR_waveforms_SPIR(namefile, signal_elements)
```

Input variables:

- **namefile** (string): Filename of SPIR 1-second binary file.
- **signal_elements** (signed char, array of 16 elements): Array indicator of which antenna elements (those indexes with a value equal to '1') will be evaluated.

9.3.11 get_waveform

Get one of the stored complex waveforms.

Example:

```
[wavI_out, wavQ_out] = my_wavcluster.get_waveform(len_I, len_Q, cluster_pos)
```

Input variables:

- **len_I** (integer): Number of samples of the in-phase component of the stored waveform (it has to be equal to **wav_length**). Recommendation: use `my_wavcluster.get_wav_length()` as **len_I**.
- **len_Q** (integer): Number of samples of the quadrature component of the stored waveform (it has to be equal to **wav_length**). Recommendation: use `my_wavcluster.get_wav_length()` as **len_Q**.
- **cluster_pos** (integer): Position of the complex waveform to be got.

Output variables:

- **wavI_out** (double, array of **wav_len** elements): In-phase component of the complex waveform in arbitrary units.
- **wavQ_out** (double, array of **wav_len** elements): Quadrature component of the complex waveform in arbitrary units.

9.3.12 integrate_waveforms

Integrate the cluster of complex waveforms, both coherent and incoherent, to get a power waveform.

Example:

```
wav_out = my_wavcluster.integrate_waveforms(coherent_int, wav_len)
```

Input variables:

- **coherent_int** (integer): Number of samples of coherent integration (from 1 to **cluster_length**). The complex waveforms are coherently integrated in blocks of **coherent_int** samples and the results are incoherently averaged.

- **wav_len** (integer): Number of samples of the stored waveforms (it has to be equal to **wav_length**). Recommendation: use `my_wavcluster.get_wav_length()` as **wav_len**.

Output variables:

- **wav_out** (double, array of **wav_len** elements): Integrated power waveform in arbitrary units.

9.3.13 integrate_waveforms_remdir

Integrate the cluster of complex waveforms, both coherent and incoherent, to get a power waveform after removing a longer coherent component to mitigate direct signal interference.

Example:

```
wav_out = my_wavcluster.integrate_waveforms_remdir(coherent_int, coherent_int_dir,
wav_len)
```

Input variables:

- **coherent_int** (integer): Number of samples of coherent integration (from 1 to **cluster_length**). The complex waveforms are coherently integrated in blocks of **coherent_int** samples and the results are incoherently averaged.

- **coherent_int_dir** (integer): Number of samples of coherent integration for mitigation of direct signal interference (from 1 to **cluster_length**).

- **wav_len** (integer): Number of samples of the stored waveforms (it has to be equal to **wav_length**). Recommendation: use `my_wavcluster.get_wav_length()` as **wav_len**.

Output variables:

- **wav_out** (double, array of **wav_len** elements): Integrated power waveform in arbitrary units.

9.3.14 integrate_waveforms_retracking

Integrate the cluster of complex waveforms, both coherent and incoherent, to get a power waveform after applying a given complex retracking along the cluster. In this context, "to apply a *retracking*" means to rotate and move in delay each complex waveform given a range value with the purpose of better align them before integration.

Example:

```
wav_out = my_wavcluster.integrate_waveforms_retracking(coherent_int,
sampling_rate, retracking_meters, wav_len)
```

Input variables:

- **coherent_int** (integer): Number of samples of coherent integration (from 1 to **cluster_length**). The complex waveforms are coherently integrated in blocks of **coherent_int** samples and the results are incoherently averaged.
- **sampling_rate** (double): Sampling rate of the waveforms from the cluster in samples/sec.
- **retracking_meters** (double, array of **cluster_length** elements): Retracking (in meters) to be applied to each waveform of the cluster before the integration.
- **wav_len** (integer): Number of samples of the stored waveforms (it has to be equal to **wav_length**). Recommendation: use `my_wavcluster.get_wav_length()` as **wav_len**.

Output variables:

- **wav_out** (double, array of **wav_len** elements): Integrated power waveform in arbitrary units.

9.3.15 dump_phase

Print a time series of the phase from the stored waveform's cluster at a given lag position (only where there are valid values).

Example:

```
my_wavcluster.dump_phase(lag_pos)
```

Input variables:

- **lag_pos** (integer): Lag position at the waveform's cluster selected to print out a time series of its phase.

9.3.16 dump_phase_peak

Print a time series of the phase from the stored waveform's cluster at a the peak position (only where there are valid values).

Example:

```
my_wavcluster.dump_phase_peak()
```

9.3.17 store_phasor_wavs

Store the contents of the waveform's cluster at a given lag in a parallel phasor for analysis purposes.

Example:

```
my_wavcluster.store_phasor_wavs(lag_pos)
```

Input variables:

- **lag_pos** (integer): Lag position at the waveform's cluster selected to store its contents in a parallel phasor.

9.3.18 get_phasor

Get the stored phasor.

Example:

```
[phasorI_out, phasorQ_out, valid_phasor_out] = my_wavcluster.get_phasor(len_Iphasor,
len_Qphasor, len_Vphasor)
```

Input variables:

- **len_Iphasor** (integer): Number of samples of the in-phase component of the stored phasor (it has to be equal to **cluster_length**). Recommendation: use `my_wavcluster.get_cluster_length()` as **len_Iphasor**.
- **len_Qphasor** (integer): Number of samples of the quadrature component of the stored phasor (it has to be equal to **cluster_length**). Recommendation: use `my_wavcluster.get_cluster_length()` as **len_Qphasor**.
- **len_Vphasor** (integer): Number of samples of the array that indicates the validity of the stored phasor (it has to be equal to **cluster_length**). Recommendation: use `my_wavcluster.get_cluster_length()` as **len_Vphasor**.

Output variables:

- **phasorI_out** (double, array of **len_Iphasor** elements): In-phase component of the stored phasor.
- **phasorQ_out** (double, array of **len_Qphasor** elements): Quadrature component of the stored phasor.
- **valid_phasor_out** (signed char, array of **len_Vphasor** elements): Array that indicates the validity of the stored phasor (1 - valid, 0 - non-valid).

9.3.19 get_sigma_phase_phasor

Compute the standard deviation of the phase from the stored phasor with the requirement of a given minimum number of valid samples (returns -1 if not enough valid samples).

Example:

```
my_wavcluster.get_sigma_phase_phasor(min_valid_samples)
```

Input variables:

- **min_valid_samples** (integer): Minimum number of valid samples in stored phasor to compute the standard deviation of the phase.

9.3.20 get_sigma_phase_phasor_interv

Compute the standard deviation of the phase from the stored phasor within a given interval and with the requirement of a given minimum number of valid samples (returns -1 if not enough valid samples).

Example:

```
my_wavcluster.get_sigma_phase_phasor_interv(init_sample, interv_samples,
min_valid_samples)
```

Input variables:

- **init_sample** (integer): Initial sample of the selected interval in the stored phasor to compute the standard deviation of the phase.
- **interv_samples** (integer): Number of samples of the selected interval in the stored phasor to compute the standard deviation of the phase.
- **min_valid_samples** (integer): Minimum number of valid samples within the selected interval of the stored phasor to compute the standard deviation of the phase.

9.3.21 counterrot_phasor

Counter-rotate the stored phasor with a given array of phases.

Example:

```
my_wavcluster.counterrot_phasor(phases_rad, valid_phases)
```

Input variables:

- **phases_rad** (double, array of **cluster_length** elements): Time series of input phases (in radians) employed to counter-rotate the stored phasor.
- **valid_phases** (signed char, array of **cluster_length** elements): Array that indicates the validity of **phases_rad** (1 - valid, 0 - non-valid).

9.3.22 counterrot_waveforms

Counter-rotate the stored complex waveforms with a given array of phases.

Example:

```
my_wavcluster.counterrot_phasor(phases_rad, valid_phases)
```

Input variables:

- **phases_rad** (double, array of **cluster_length** elements): Time series of input phases (in radians) employed to counter-rotate the stored complex waveforms.

- **valid_phases** (signed char, array of **cluster_length** elements): Array that indicates the validity of **phases_rad** (1 - valid, 0 - non-valid).

9.3.23 correct_navigation_bit

Apply an algorithm to correct the navigation bit (for GPS L1 C/A code) in all the waveforms from the stored cluster. Such algorithm analyzes the phase variations at a given lag and decides where there is a bit transition. Since coherence and proper SNR are required, this function is recommended for waveform clusters containing direct GPS L1 signals; however, the information obtained could be later applied to the clusters containing their corresponding reflections (if available).

Example:

```
my_wavcluster.correct_navigation_bit(lag_pos, store_navbit_phasorI)
```

Input variables:

- **lag_pos** (integer): Lag position where the algorithm is applied (peak is recommended).

- **store_navbit_phasorI** (integer): 1 - store the time series of the navigation bit estimated at the in-phase component of the stored phasor, 0 - do not store the navigation bit.

9.3.24 compute_coherence_time

Apply an algorithm to estimate the coherence time from the stored cluster. Such algorithm checks the accumulated phase variation at a given lag and sets the coherence time at the moment when such variation reaches 1 radian.

Example:

```
coh_time = my_wavcluster.compute_coherence_time(lag_pos, store_acdiff_phasorQ)
```

Input variables:

- **lag_pos** (integer): Lag position where the algorithm is applied (peak is recommended).
- **store_acdiff_phasorQ** (integer): 1 - store the time series of the accumulated phase difference at the quadrature component of the stored phasor, 0 - do not store the navigation bit.

Output variables:

- **coh_time** (double): Coherence time obtained in cluster sample units.

9.3.25 compute_singlefreq_DDM

Compute a DDM's Doppler slice from the waveform cluster. The procedure is the same as doing an integration, but after modulating the waveforms using a phasor at the given frequency.

Example:

```
freq_ddm_out = my_wavcluster.compute_singlefreq_DDM(coherent_int,
doppler_freq, ddm_lag_len)
```

Input variables:

- **coherent_int** (integer): Number of samples of coherent integration (from 1 to **cluster_length**). The complex waveforms are coherently integrated in blocks of **coherent_int** samples and the results are incoherently averaged.
- **doppler_freq** (double): Doppler frequency of the output DDM slice in inverse cluster sample units.
- **ddm_lag_len** (integer): Number of delay samples of the DDM (it has to be equal to **wav_length**). Recommendation: use `my_wavcluster.get_wav_length()` as **ddm_lag_len**.

Output variables:

- **freq_ddm_out** (double, array of **ddm_lag_len** elements): DDM's Doppler slice in arbitrary units.

9.3.26 compute_singlefreq_DDM_remdir

Compute a DDM's Doppler slice from the waveform cluster after removing a longer coherent component to mitigate direct signal interference. The procedure is the same as doing an integration, but after modulating the waveforms using a phasor at the given frequency.

Example:

```
freq_ddm_out = my_wavcluster.compute_singlefreq_DDM_remdir(coherent_int,
coherent_int_dir, doppler_freq, ddm_lag_len)
```

Input variables:

- **coherent_int** (integer): Number of samples of coherent integration (from 1 to **cluster_length**). The complex waveforms are coherently integrated in blocks of **coherent_int** samples and the results are incoherently averaged.
- **coherent_int_dir** (integer): Number of samples of coherent integration for mitigation of direct signal interference (from 1 to **cluster_length**).
- **doppler_freq** (double): Doppler frequency of the output DDM slice in inverse cluster sample units.
- **ddm_lag_len** (integer): Number of delay samples of the DDM (it has to be equal to **wav_length**). Recommendation: use `my_wavcluster.get_wav_length()` as **ddm_lag_len**.

Output variables:

- **freq_ddm_out** (double, array of **ddm_lag_len** elements): DDM's Doppler slice in arbitrary units.

9.3.27 compute_singlelag_DDM

Compute a DDM's delay slice from the waveform cluster. The procedure is the same as doing an integration, but after modulating the waveforms using a phasor at a given set of frequencies.

Example:

```
lag_ddm_out = my_wavcluster.compute_singlelag_DDM(coherent_int, lag_pos,
delta_freq, ddm_freq_len)
```

Input variables:

- **coherent_int** (integer): Number of samples of coherent integration (from 1 to **cluster_length**). The complex waveforms are coherently integrated in blocks of **coherent_int** samples and the results are incoherently averaged.
- **lag_pos** (integer): Delay lag of the output DDM slice.
- **delta_freq** (double): Doppler frequency resolution of the DDM in inverse cluster sample units.
- **ddm_freq_len** (integer): Number of Doppler bins in the DDM, starting from (-**ddm_freq_len**/2)*__delta_freq__.

Output variables:

- **lag_ddm_out** (double, array of **ddm_freq_len** elements): DDM's delay slice in arbitrary units.

9.3.28 compute_singlelag_DDM_remdir

Compute a DDM's delay slice from the waveform cluster after removing a longer coherent component to mitigate direct signal interference. The procedure is the same as doing an integration, but after modulating the waveforms using a phasor at a given set of frequencies.

Example:

```
lag_ddm_out = my_wavcluster.compute_singlelag_DDM_remdir(coherent_int,
coherent_int_dir, lag_pos, delta_freq, ddm_freq_len)
```

Input variables:

- **coherent_int** (integer): Number of samples of coherent integration (from 1 to **cluster_length**). The complex waveforms are coherently integrated in blocks of **coherent_int** samples and the results are incoherently averaged.
- **coherent_int_dir** (integer): Number of samples of coherent integration for mitigation of direct signal interference (from 1 to **cluster_length**).

- **lag_pos** (integer): Delay lag of the output DDM slice.
- **delta_freq** (double): Doppler frequency resolution of the DDM in inverse cluster sample units.
- **ddm_freq_len** (integer): Number of Doppler bins in the DDM, starting from (-**ddm_freq_len**/2)***delta_freq**.

Output variables:

- **lag_ddm_out** (double, array of **ddm_freq_len** elements): DDM's delay slice in arbitrary units.

9.3.29 compute_DopplerMap_BW

Compute the 3 dB bandwidth from the DDM at the Doppler domain for a given lag.

Example:

```
[pos_max, pow_max] = my_wavcluster.compute_DopplerMap_BW(coherent_int,
lag_pos, ddm_freq_len, delta_freq)
```

Input variables:

- **coherent_int** (integer): Number of samples of coherent integration (from 1 to **cluster_length**). The complex waveforms are coherently integrated in blocks of **coherent_int** samples and the results are incoherently averaged.
- **lag_pos** (integer): Delay lag of the DDM slice.
- **ddm_freq_len** (integer): Number of Doppler bins in the DDM, starting from (-**ddm_freq_len**/2)***delta_freq**.
- **delta_freq** (double): Doppler frequency resolution of the DDM in inverse cluster sample units.

Output variables:

- **pos_max** (double): Peak location of the DDM's delay slice in inverse sample units.
- **pow_max** (double): Peak value of the DDM's delay slice in arbitrary units.

9.3.30 compute_DopplerMap_BW_remdir

Compute the 3 dB bandwidth from the DDM at the Doppler domain for a given lag and after removing a longer coherent component to mitigate direct signal interference.

Example:

```
[pos_max, pow_max] = my_wavcluster.compute_DopplerMap_BW_remdir(coherent_int,
coherent_int_dir, lag_pos, ddm_freq_len, delta_freq)
```

Input variables:

- **coherent_int** (integer): Number of samples of coherent integration (from 1 to **cluster_length**). The complex waveforms are coherently integrated in blocks of **coherent_int** samples and the results are incoherently averaged.
- **coherent_int_dir** (integer): Number of samples of coherent integration for mitigation of direct signal interference (from 1 to **cluster_length**).
- **lag_pos** (integer): Delay lag of the DDM slice.
- **ddm_freq_len** (integer): Number of Doppler bins in the DDM, starting from (-**ddm_freq_len**/2)*__delta_freq__.
- **delta_freq** (double): Doppler frequency resolution of the DDM in inverse cluster sample units.

Output variables:

- **pos_max** (double): Peak location of the DDM's delay slice in inverse cluster sample units.
- **pow_max** (double): Peak value of the DDM's delay slice in arbitrary units.

9.3.31 compute_LagHologram

Compute a lag-hologram delay slice from the waveform cluster. The procedure consist in computing the FFT algorithm from the time series at the given lag position from the cluster of waveforms.

Example:

```
lag_hologram_out = my_wavcluster.compute_LagHologram(lag_pos, fft_len)
```

Input variables:

- **lag_pos** (integer): Delay lag of the output lag-hologram slice.
- **fft_len** (integer): Number of FFT samples (from 2 to **cluster_length**, being a power of 2).

Output variables:

- **lag_hologram_out** (double, array of **fft_len** elements): Lag-hologram delay slice in arbitrary units. The corresponding frequencies range from -(int(**fft_len**/2) - 1) to int(**fft_len**/2) in inverse cluster sample units.

9.3.32 get_wav_length

Get the length of the waveforms stored.

Example:

```
wav_len_out = my_wavcluster.get_wav_length()
```

Output variables:

- **wav_len_out** (integer): Length of the waveforms stored.

9.3.33 get_cluster_length

Get the number of waveforms stored in the cluster.

Example:

```
cluster_len_out = my_wavcluster.get_cluster_length()
```

Output variables:

- **cluster_len_out** (integer): Number of waveforms stored in the cluster.

References

[**Beyerle, 09**]: Beyerle, G., Carrier phase wind-up in GPS reflectometry, *GPS Solutions*, 13, 3, 191-198, doi: 10.1007/s10291-008-0112-1.

[**Brown, 78**]: Brown, G.S., Backscattering from a Gaussian-Distributed Perfectly Conducting Rough Surface, *IEEE Trans. Antennas and Propagation*, AP-26, No.3, May 1978.

[**Cardellach et al, 12**]: Cardellach, E., F. Fabra, A. Rius, S. Pettinato and S. D'Addio, Characterization of Dry snow Substructure using GNSS Reflected Signals, *Remote Sensing Environment*, 124, 2012.

[**Elfouhaily et al, 97**]: Elfouhaily, T., B. Chapron, K. Katsaros and D. Vandemark, A unified directional spectrum for long and short wind-driven waves, *J. Geosphys. Res.*, 102, 15781-15796, 1997.

[**Katzberg et al, 06**]: Katzberg, S.J., O. Torres, and G. Ganoe (2006), Calibration of reflected GPS for tropical storm wind speed retrievals, *Geophys. Res. Lett.*, vol.33, N.L18602, doi:10.1029/2006GL026825.

[**Li et al, 17**]: Li, W., A. Rius, F. Fabra, E. Cardellach, S. Rib and M. Martn-Neira, Revisiting the GNSS-R Waveform Statistics and its Impact on Altimetric Retrievals, submitted to *IEEE Trans. Geosc. Remote Sensing*, 2017.

[**Rius et al, 10**]: Rius, A., E. Cardellach and M. Martn-Neira, Altimetric Analysis of the Sea-Surface GPS-Reflected Signals, *Geoscience and Remote Sensing, IEEE Transactions on*, vol. 48, no. 4, pp. 2119-2127, Apr 2010.

[**Ulaby et al, 90**]: F. T. Ulaby, R. K. Moore, and A. K. Fung. Microwave Remote Sensing Active and Passive, volume III. Artech House, Inc., 1990.

[**Zavorotny and Voronovich, 00**]: Zavorotny, V.U., and A.G. Voronovich, Scattering of GPS signals from the ocean with wind remote sensing application, *IEEE Trans. Geosci. Remote Sens.*, 38, 951964, 2000.