GaiaLab Documentation

Release 0.0.1

Alex Bombrun, Toby James, Maria Del Vallo, Luca Zampieri

Jan 31, 2019

Contents:

CHAPTER 1

README file

Just to check that everything is working.

Note that to add modules (e.g. with automodule) the path to the module should be in conf.py as done previoulsy.

CHAPTER 2

Installation

2.1 Linux

- Install dependencies (with pip or conda)
- git clone the repository
- Enjoy!

2.2 Windows

Not tested but should be similar than Linux

2.3 Mac

Not tested but should be similar than Linux

CHAPTER 3

Auxiliary code

Helpers functions necessary to the core code and plotting functions.

3.1 frame_transformations.py

File frame_transformations.py Contains functions that for frame transformations and rotations

Authors mdelvallevaro, LucaZampieri (2018)

Note: In this file, when there is a reference, unless explicitly stated otherwise, it refers to Lindegren main article: "The astronometric core solution for the Gaia mission - overview of models, algorithms, and software implementation" by L. Lindegren, U. Lammer, D. Hobbs, W. O'Mullane, U. Bastian, and J.Hernandez The reference is usually made in the following way: Ref. Paper *[LUDW2011]* eq. [1]

frame_transformations.**adp_to_cartesian**(*alpha*, *delta*, *parallax*)

Convert coordinates from (alpha, delta, parallax) format into the (x, y, z) format.

Parameters

- **azimuth** – [rad]
- **altitude** – [rad]
- **parallax** – [mas]

Returns [parsec](x, y, z) array in parsecs.

frame_transformations.**compute_ljk**(*epsilon*)

Ref. *[Lind2001]* (Lindegren, SAG-LL-35, Eq.1)

Calculates ecliptic triad vectors with respect to BCRS-frame.

Parameters **epsilon** – obliquity of the equator.

Returns

np.array, np.array, np.array of the ecliptic triad:

- **l**: is a unit vector toward (alpha, delta) = (0,0)‘)
- **n**: is a unit vector towards delta = 90°
- **m** = **n** x **l**

frame_transformations.**compute_pqr**(*alpha*, *delta*)

Ref. Paper *[LUDW2011]* eq. [5]
Can be used also with numpy arrays
Computes the p, q, r parameters

Parameters

- **alpha** – [rad] astronomic parameter alpha
- **delta** – [rad] astronomic parameter alpha

Returns p, q, r

frame_transformations.**lmn_to_xyz**(*attitude*, *vector*)
Ref. Paper *[LUDW2011]* eq. [9] Goes from the non-rotating (lmn) CoMRS frame to the rotating (xyz) SRS frame

---

Note: The attitude Quaternion q(t) gives the rotation from (lmn) to (xyz) (lmn) being the CoMRS (C), and (xyz) the SRS (S). The relation between the two frames is given by: {S’v,0} = q^-1 {C’v,0} q for an any vector v

---

Parameters

- **attitude** – Quaternion object
- **vector** – array of 3D

Returns the coordinates in XYZ-frame of the input vector.

frame_transformations.**polar_to_direction**(*alpha*, *delta*)
Convert polar angles to unit direction vector

Parameters

- **alpha** – [rad]
- **delta** – [rad]

Returns 3D np.array unit vector

frame_transformations.**quat_to_vector**(*quat*)

Parameters **quat** – [quaternion] Quaternion to transform into vector

Returns 3D array made with x,y,z components of the quaternion

frame_transformations.**rotate_by_quaternion**(*quaternion*, *vector*)
Ref. Paper *[LUDW2011]* eq. [9] rotate vector by quaternion

frame_transformations.**transform_twoPi_into_halfPi**(*deltas*)
Tranforms an angle in range [0-2*pi] to range [-pi/2, pi/2] by substracting 2pi to any angle greater than pi.

> **Warning:** The input angles have to be defined between [0,pi/2] and [3pi/2, 2pi]

Parameters **delta** – input angles

Returns modified angles

frame_transformations.**vector_to_alpha_delta**(*vector*, *two_pi=False*)
Ref. Paper *[LUDW2011]* eq. [96] Convert carthesian coordinates of a vector into its corresponding polar coordinates (0 - 2*pi)

Parameters

- **vector** – [array] X,Y,Z coordinates in CoMRS frame (non-rotating)
- **two_pi** – [bool] if True return delta in [0,2pi] instead of [-pi/2,pi/2]

Returns [rad][rad] alpha, delta

frame_transformations.**vector_to_quat**(*vector*)
Transform vector to quaternion by setting x,y,z components of the quaternion with x,y,z components of the vector.

Parameters **vector** – vector to transform to quaternion

Returns quaternion created from vector

frame_transformations.**xyz_to_lmn**(*attitude*, *vector*)
Ref. Paper *[LUDW2011]* eq. [9] Go from the rotating (xyz) SRS frame to the non-rotating (lmn) CoMRS frame

---

Note: The attitude Quaternion q(t) gives the rotation from (lmn) to (xyz) (lmn) being the CoMRS (C), and (xyz) the SRS (S). The relation between the two frames is given by: {C'v,0} = q {S'v,0} q^-1 for an any vector v

---

Parameters

- **attitude** – Quaternion object
- **vector** – array of 3D

Returns the coordinates in LMN-frame of the input vector.

frame_transformations.**zero_to_two_pi_to_minus_pi_pi**(*angle*, *unit='radians'*)
Tranforms an angle in range [0-2*pi] to range [-pi, pi] by substracting 2pi to any angle greater than pi.

Info: Can be used with numpy arrays

Parameters

- **angle** – [rad] angle or array of angles in [0-2*pi] format
- **unit** – [str] specify if the input data is in radians or degrees

Returns angle in the [-pi, pi] format

3.2 helpers.py

File helpers.py

Helper functions for the analytic scanner

Contains: (at least)

- compute_intersection
- compute_angle

Author LucaZampieri (2018)

helpers.**check_symmetry**(*a*, *tol=1e-12*)
Check the symmetry of array a. True if symmetric up to tolerance

helpers.**compute_angle**(*v1*, *v2*)
Computes the angle between two Vectors :param vi: vector between which you want to compute the angle for each i=1:2 :returns: [float] [deg] angle between the vectors

helpers.**compute_intersection**(*x1*, *y1*, *x2*, *y2*, *x3*, *y3*, *x4*, *y4*, *segment=True*)
Return intersection of two lines (or segments) if it exists, raise an error otherwise. :param xi: x-coordinate of segment i for i=1:4 :param yi: y-coordinate of segment i for i=1:4 :param segment: [bool]

Returns

- (x, y) tuple with x and y coordinartes of the intersection point
- [list] error_msg list

helpers.**get_rotation_matrix**(*v1*, *v2*)
Get the rotation matrix necessary to go from v1 to v2 :param vi: 3D vector as np.array To rotate vector v1 into v2 then do r@v1

helpers.**get_sparse_diagonal_matrix_from_half_band**(*half_band*)
it assumes we have the upper hal of the band, i.e. we some zeros at the bottom of band for column index greater than 0

helpers.**normalize**(*v*, *tol=1e-10*)
return normalized version of v

helpers.**plot_sparsity_pattern**(*A*, *tick_frequency*)

Parameters **A** – np array containing the matrix

helpers.**sec**(*x*)
Stable if x close to 0

helpers.**symmetrize_triangular_matrix**(*a*)
Symmetrize an already triangular matrix (lower or upper) :param a: upper ot lower triangular matrix :returns: corresponding symmetric matrix

3.3 analytic_plots.py

CHAPTER 4

Core code

Here is described the list of files documented and corresponding functions for the core part of the code. Helper functions and plotting functions are documented apart, under auxiliary code.

4.1 satellite.py

Satellite class implementation in Python

Todo:

- repair attitude for t_init != 0

Author mdelvallevaro

satellite.**gaia_orbit**(*t*, *epsilon*)

Parameters

- **t** – time at which we want the position
- **epsilon** – obiquity of equator

Returns the gaia position at time **t**, assuming it is a circle around the sun tilted by epsilon:

class satellite.**Satellite**(*ti=0*, *tf=1825*, *dt=0.041666666666666664*, *k=3*, **args*)

Class Object, that represents a satellite, e.g. Gaia.
Creates spline from attitude data for 5 years by default.

Note: (see e.g. Lindegren, SAG-LL-35)

The Nominal Scanning Law (NSL) for gaia is descibed by two constant angles:

- Epsilon: obliquity of equator
- Xi (greek letter): revolving angles

and 3 angles that increase continuously but non-uniformly:

- _lambda(t): nominal longitude of the sun
- nu(t): revolving phase
- omega(t): spin phase

With also initial values nu(0), omega(0) at time t_0 the NSL is completely specified.

__init__ (*ti=0*, *tf=1825*, *dt=0.041666666666666664*, *k=3*, **args*)

Parameters

- **ti** – initial time, float [day]
- **tf** – final time, float [day]
- **dt** – time step for creation of discrete data fed to spline, float [day].
- **S** – change in the z-axis of satellite wrt solar longitudinal angle. [float]
- **epsilon** – ecliptical angle [rad]
- **xi** – revolving angle [rad]
- **wz** – z component of inertial spin vector [arcsec/s]

Action Sets satellite to initialization status.

orbital_period = None
orbital_period [days]

orbital_radius = None
orbital_radius

spline_degree = None
degree of the interpolating polynomial. spline_degree = spline_order - 1

init_parameters (*S=4.035*, *epsilon=0.4090928042223289*, *xi=0.9599310885968813*, *wz=60*)
Init parameters with values in file contants.py

ephemeris_bcrs (*t*)
Defines the orbit of the satellite around the sun Returns the barycentric ephemeris of the Gaia satellite at time t. Equivalently written b_G(t)

Parameters **t** – float [days]

Returns [np.array] 3D [AU] BCRS position-vector of the satellite

reset (*ti*, *tf*, *dt*)

Returns reset satellite to initialization status

4.2 source.py

Source class implementation in Python

Authors mdelvallevaro LucaZampier (modifications)

source.**compute_topocentric_direction**(*astro_parameters*, *sat*, *t*)

Compute the topocentric_function direction i.e. ũ The horizontal coordinate system, also known as topocentric coordinate system, is a celestial coordinate system that uses the observer's local horizon as the fundamental plane. Coordinates of an object in the sky are expressed in terms of altitude (or elevation) angle and azimuth.

Parameters

- **astro_parameters** – [alpha, delta, parallax, mu_alpha_dx, mu_delta, mu_radial] [rads] the astrometric parameters
- **sat** – [Satellite]
- **t** – [float][days] time at which we want the topocentric function

Returns [array] (x,y,z) direction-vector of the star from the satellite's lmn frame. (CoMRS)

class source.**Source**(*name*, *alpha0*, *delta0*, *parallax*, *mu_alpha*, *mu_delta*, *radial_velocity*, *func_color=<function Source.<lambda>>*, *mean_color=0*)

Source class implemented to represent a source object in the sky

Examples:

```
>>> vega = Source("vega", 279.2333, 38.78, 128.91, 201.03, 286.23, -13.9)
>>> proxima = Source("proxima",217.42, -62, 768.7, 3775.40, 769.33, 21.7)
>>> sirio = Source("sirio", 101.28, -16.7161, 379.21, -546.05, -1223.14, -7.6)
```

__init__(*name*, *alpha0*, *delta0*, *parallax*, *mu_alpha*, *mu_delta*, *radial_velocity*, *func_color=<function Source.<lambda>>*, *mean_color=0*)

Parameters

- **alpha0** – [deg]
- **delta0** – [deg]
- **parallax** – [mas]
- **mu_alpha** – [mas/yr]
- **mu_delta** – [mas/yr]
- **radial_velocity** – [km/s]
- **func_color** – function representing the color of the source in nanometers
- **mean_color** – mean color observed by satellite

Transforms in rads/day or rads, i.e. we got:

- [alpha] = rads
- [delta] = rads
- [parallax] = rads
- [mu_alpha_dx] = rads/days
- [mu_delta] = rads/days
- [mu_radial] = rads/days

get_parameters(*t=0*)

Returns astrometric parameters at time t (t=0 by default)

reset()
Reset star position to t=0

set_time(*t*)
Sets star at position wrt bcrs at time t.

Parameters **t** – [float][days] time

barycentric_direction(*t*)
Direction unit vector to star from bcrs.

Parameters **t** – [float][days]

Returns ndarray 3D vector of [floats]

barycentric_coor(*t*)
Vector to star wrt bcrs-frame.

alpha: [float][rad] delta: [float][rad] parallax: [float][rad] :param t: [float][days] :return: ndarray, length 3, components [floats][parsecs]

unit_topocentric_function(*satellite*, *t*)
Compute the topocentric_function direction

Parameters **satellite** – satellite [class object]

Returns [array] (x,y,z) direction-vector of the star from the satellite's lmn frame.

topocentric_angles(*satellite*, *t*)
Calculates the angles of movement of the star from bcrs.

Parameters

- **satellite** – satellite object
- **t** – [days]

Returns alpha, delta, delta alpha, delta delta [mas]

4.3 scanner.py

Scanner class implementation in Python

Authors LucaZampieri (2018) mdelvallevaro (2018)

scanner.**eta_angle**(*t*, *sat*, *source*, *FoV='centered'*)

Function to minimize in the scanner.
See *agis_functions.observed_field_angles()*

Parameters **FoV** – [string] specify which Field of View to use

scanner.**get_etas_from_phis**(*phi_a*, *phi_b*, *FoV*)
Tranform phis into etas using the field of view parameter

Parameters

- **phi_a** – phi at the beginning of the interval
- **phi_b** – phi at the end of the interval
- **FoV** – [string] specify which Field of View we are using

Returns eta at the beginning and end of the inteval

scanner.**violated_contraints**(*eta_a*, *zeta_a*, *eta_b*, *zeta_b*, *zeta_limit*)

Returns True if the contraints are violated and False otherwise

class scanner.**Scanner**(*zeta_limit=0.00872664625997164 8*, *double_telescope=True*)

__init__(*zeta_limit=0.00872664625997164 8*, *double_telescope=True*)

Parameters

- **zeta_limit** – [rads] limitation of the Field of View in the across scan direction
- **double_telescope** – [bool] if true implements the scanner version with two telescopes (gaia-like)

reset(*verbose=False*)

Action empty all attribute lists from scanner before beginning new scanning period.

Parameters **verbose** – [bool] If true will print messages

scan(*sat*, *source*, *ti*, *tf*)
Find the exact time in which the source is seen.

Action Find the observation time of the sources

Parameters

- **sat** – [Satellite object]
- **source** – [Source object]
- **& tf** (*ti*) – [days] initial and end dates

Returns [float] time it took for the scan

compute_angles_eta_zeta(*sat*, *source*)
Compute angles and remove 'illegal' observations ($|zeta| > zeta_lim$)

scanner_error()

Returns mean error in the Along-scan direction

4.4 agis.py

File agis.py

purpose Contains implementation of classes Calc_source and Agis

Authors LucaZampieri

4.4.1 Description of what we are trying to achieve in this file

It is a simplification of what is in *[LUDW2011]* . **If in doubt, check with the paper.**

The goal would be to minimize:

$$min_{s,a}Q = \sum_{l \in AL} (R_l^{AL})^2 + \sum_{l \in AC} (R^{AC_l})^2$$

In the following we might forget about R^{AL}, R^{AC} and just use R_l (for simplicity)

The necessary condition for optimality would be that the derivative is null. So given that we want to optimize with respect to variable **x** we would have:

$$\frac{dQ}{dx} = \sum_l 2R_l \frac{dR_l}{dx} = 0$$

Later we will see that x will correspond to the source parameters and the coefficient describing the attitude. (i.e. $x \in \{s, a\}$, **s** being the the set of 5 source parameter and a being the coeffeicients of the splines for each of the four quaternions parameters describing the attitude of the satellite at each time)

This condition being sufficient if Q is convex.

Q being non-linear we are gonna linearize it. Assuming our initial guess is good enough we can write the optimal set of parameters $x^* = x^{ref} + x$

We would get:

$$R_l(x^*) = R_l(x^{ref}) + \frac{dR_l(x^{ref})}{dx}(x^* - x^{ref}) = R_l(x^{ref}) + \frac{dR(x^{ref})}{dx}x$$
$$Q(x) = [R_l(x^{ref})^2 + 2R_l(x^{ref})\frac{dR(x^{ref})}{dx}x + (\frac{dR_l(x^{ref})}{dx}x)^2]$$

Thus at optimality we should have ($\frac{dQ}{dx} = 0$):

$$[\frac{dR_l(x^{ref})}{dx}]^T[\frac{dR_l(x^{ref})}{dx}]x = -\frac{dR_l(x^{ref})}{dx}R_l(x^{ref})$$

Note: Maily for next contributors:

- Chowleski factorization (LL^T) is about twice as efficient as LU factorization (default used by numpy.linalg.solve/scipy) if the matrix is hermitian which is the case for us (we have a symmetric matrix)

---

class agis.**Calc_source**(*name=None*, *obs_times=[]*, *source_params=None*, *mu_radial=None*, *mean_color=0*, *source=None*)

Contains the calculated parameters per source

__init__(*name=None*, *obs_times=[]*, *source_params=None*, *mu_radial=None*, *mean_color=0*, *source=None*)

Data structure containing our computed parameters for the source in question.

Parameters

- **name** – [string] the name of the source
- **obs_times** – [list or array] of the observed times for this source
- **source_params** – [list or array] alpha, delta, parallax, mu_alpha, mu_delta
- **mu_radial** – [float] radial velocity of the source (appart since we do not solve for radial velocity)
- **mean_color** – [float] mean color of the source if we want to use explore chromatic aberration
- **source** – [source] instead of most of the above parameters we can provide a source object instead and take the data from it. Manually providing the parameters will override the source parameter

see *source.Source*

```
>>> calc_source = Calc_source('calc_sirio', [1, 2.45, 12], [1, 2, 3, 4, 5], 6)
>>> calc_source = Calc_source(obs_times=[1, 2, 3], source=sirio)  # where␣
↪sirio is a source object
```

class agis.**Agis**(*sat*, *calc_sources=[]*, *real_sources=[]*, *attitude_splines=None*, *verbose=False*, *spline_degree=3*, *attitude_regularisation_factor=0*, *updating='attitude'*, *degree_error=0*, *double_telescope=False*)

__init__(*sat*, *calc_sources=[]*, *real_sources=[]*, *attitude_splines=None*, *verbose=False*, *spline_degree=3*, *attitude_regularisation_factor=0*, *updating='attitude'*, *degree_error=0*, *double_telescope=False*)

Parameters

- **sat** – [Satellite]
- **calc_sources** – [list of Calc_sources]
- **real_sources** – [liste of Sources]
- **attitude_splines** – list of splines, only if we update attitude
- **verbose** – [bool] if True prints intermediate results
- **spline_degree** – degree of the splines
- **attitude_regularization_factor** – [float] lambda in the paper. Describes how much we take into account the regularization into our minimization problem
- **updating** – [string] defines what we are currently updating can be: *'source', 'scanned source' or 'attitude'*
- **degree_error** –
- **double_telescope** – [bool] if True uses two telescopes (gaia-like)

real_sources = None
List of the sources objects

calc_sources = None
List of calculated sources with 1-1 correspondance to the real sources

sat = None
Satellite object that we are using to solve the problem

k = None
Degree of the interpolating polynomial

M = None
Order of the spline (degree+1)

time_dict = None
[dict] containig the observed times assiciated with their source index

all_obs_times = None
[array] containing the combined observation time of all the sources

reset_iterations()
Resetting the iteration counter

error_function()

Ref. Paper *[LUDW2011]* eq. [24]

Compute the error function Q

$$Q = \sum_{l \in AL} (R^{AL})^2 + \sum_{l \in AC} (R^{AC})^2$$

get_field_angles(*source_index*, *t*)

Uses functions *get_attitude_for_source()*, *get_attitude()*, `observed_field_angles()`, `calculated_field_angles()`, `satellite.Satellite.func_attitude()`

Parameters

- **source_index** – [int] index of the source
- **t** – [float] [days] time at which

Returns eta_obs, zeta_obs, eta_calc, zeta_calc

deviate_field_angles_color_aberration(*source_index*, *t*, *angles*)

apply color aberration deviation to field angles (eta, zeta)

Note: If we the color aberration is not given in the sources, nothing happens and the same angles are returned.

---

compute_R_L(*source_index*, *t*)

Ref. Paper *[LUDW2011]* eq. [25]-[26].
$R = eta_{obs} + zeta_{obs} - eta_{calc} - zeta_{calc}$

Note R_AL = R_eta, R_AC = R_zeta

Returns [tuple] R_eta, R_zeta

iterate(*num*, *use_sparse=False*, *verbosity=0*)

Do _num_ iterations

update_S_block()

Performs the update of the source parameters

update_block_S_i(*source_index*)

Ref. Paper *[LUDW2011]* eq. [57]
Uses function *block_S_error_rate_matrix()*

Parameters **source_index** – [int] Index of the source that will be updated

Action update source number *source_index*

compute_h(*source_index*)

Ref. Paper *[LUDW2011]* eq. [59]
Source update Right hand side
Uses functions *compute_R_L()*

Warning: here we sum along-scan and across-scan observations. It should be good for this update, but be carefull in future implementations

Parameters **source_index** – [int] Index of the source that will be updated

block_S_error_rate_matrix(*source_index*)

Ref. Paper *[LUDW2011]* eq. [58]
error matrix for the block update S

uses *compute_du_ds()* and *dR_ds()*

Parameters **source_index** – [int] Index of the source that will be updated

dR_ds(*source_index*, *du_ds*)
Ref. Paper eq. [71] Computes the derivative of the error (R_l) wrt the 5 astronomic parameters s_i transposed.

Parameters

- **source_index** – [int] Index of the source that will be updated
- **du_ds** – [numpy array] derivative of the topocentric function wrt astronometric parameters

Returns [tuple of numpy arrays] with derivatives of observations in the Along-scan (AL) and Across-scan (AC) directions

compute_du_ds(*source_index*)

Ref. Paper *[LUDW2011]* eq. [73]
Compute dũ_ds for a given source

Note:

- b_G(t) barycentric position of Gaia at the time of observation, also called barycentric ephemeris of the Gaia Satellite
- t_B barycentric time (takes into account the Römer delay)
- t_ep in the paper is not used since we assume t_ep=0 and start counting the time from J2000

Uses *CoMRS_to_SRS_for_source_derivatives()*, *satellite.Satellite.ephemeris_bcrs()* and *agis_functions.compute_du_dparallax()*

Parameters **source_index** – [int] Index of the source that will be updated

Returns du_ds - source derivatives (wrt astrometric parameters) in SRS

CoMRS_to_SRS_for_source_derivatives(*CoMRS_derivatives*, *calc_source*, *t_L*, *source_index*)

Ref. Paper *[LUDW2011]* eq. [72]
rotate the frame from CoRMS (lmn) to SRS (xyz) for the given derivatives

Parameters

- **CoMRS_derivatives** – [list] of deriatives [du_dalpha, du_ddelta, du_dparallax, du_dmualpha, du_dmudelta]
- **calc_source** – [Calc_source]
- **t_L** – [float] time of observation
- **source_index** – [int]

get_attitude_for_source(*source_index*, *t*)
For only source updating with **color aberration**. Change if condition to decide which sources are affected by that aberration

Parameters

- **source_index** – [int] Index of the source that will be updated
- **t** – [float] [days] time at which we want the attitude

get_attitude(*t*, *unit=True*)
Get attitude from the attitude coefficients at time *t*. If *unit* is True, the return normalized attitude.

Parameters

- **t** – [float] time at which we desire the attitude
- **unit** – [bool] if true normalize the quaternion

Returns [Quaternion object] attitude

actualise_splines()

Action Update the splines re-creating them from the new coefficients

update_A_block(*use_sparse=False*)

Ref. Paper *[LUDW2011]* Section 5.2
Solve for the attitude

update_A_block_bis()

updates the four components separately (wrong but not by much)
It should work if we update just some of the blocks (like one or two)

compute_attitude_LHS()
Ref. Paper *[LUDW2011]* eq. [82]

compute_attitude_RHS()
Ref. Paper *[LUDW2011]* eq. [82]

get_source_index(*t*)
get the index of the source corresponding to observation time t

Parameters **t** – [float] observation time

Returns [int] source index

compute_attitude_RHS_n(*n_index*)
Ref. Paper *[LUDW2011]* eq. [82]

Parameters **n_index** –

Returns entry **n** of the Right Hand Side column vector for the attitude update

compute_Naa_mn(*m_index*, *n_index*)

Ref. Paper *[LUDW2011]* eq. [82]
compute dR/da (i.e. wrt coeffs)

Parameters

- **m_index** –
- **n_index** –

Returns entry [n, m] of the attitude normal matrix N_a

compute_attitude_banded_derivative_and_regularisation_matrices()

Ref. Paper *[LUDW2011]* eq. [82]
Computes the bands of the banded matrices of the derivatives for the normal matrix and the band for the regularization of the normal matrix.

Returns (der_band, reg_band)

compute_sparses_matrices(*der_band*, *reg_band*)

Action Creates sparse banded matrices from bands and stores them in class properties

Parameters

- **der_band** – band for the derivatives part of the normal attitude matrix
- **reg_band** – band for the regularization part of the normal attitude matrix

compute_matrix_reg_mn(*m_index*, *n_index*)

Ref. Paper *[LUDW2011]* eq. [82]
compute $\lambda^2 * \frac{dD_l}{da_m} * \frac{dD_l}{/} da_n^T$ (i.e. wrt coeffs)

Parameters

- **m_index** – [int]
- **n_index** – [int]

Returns reg_mn [n, m] entry of the regularisation part of the attitude normal matrix

compute_matrix_der_mn(*m_index*, *n_index*)

Ref. Paper *[LUDW2011]* eq. [82]
Compute $\frac{dR_l}{da_n} \frac{dR_l}{da_m^T}$ (i.e. wrt coeffs)

Parameters

- **m_index** – [int]
- **n_index** – [int]

Returns der_mn, entry [n ,m] of the derivative parts of the attitude normal matrix

4.5 agis_functions.py

File agis_functions.py

purpose Functions that uses the classes source, satellite but don't belong to a given file yet

used by (at least) agis.py & scanner.py

author LucaZampieri

The file is divided in sections, one for each purpose they serve.

When cleaning this file search for ???, LUCa, warning , error, debug, print?

Note: In this file, when there is a reference, unless explicitly stated otherwise, it refers to Lindegren main article: "The astronometric core solution for the Gaia mission - overview of models, algorithms, and software implementation" by L. Lindegren, U. Lammer, D. Hobbs, W. O'Mullane, U. Bastian, and J.Hernandez The reference is usually made in the following way: Ref. Paper *[LUDW2011]* eq. [1]

Todo:

- [DONE] Rotate the attitude
- [DONE] attitude i.e. generate observations (with scanner object but without scanning)
- [DONE] with scanner
- [DONE] two telescope
- Attitute with scanner
- scaling

Todo: (bis)

- implement chromatic aberration
- add acceleration to proper motion
- add noise to observation
- QSO
- signal
- binary source

---

agis_functions.**add_noise_to_calc_source**(*s*, *noise=1e-12*)

Action Adds noise to given calc_source (inline)

Parameters

- **s** – [calc_source object] that we wish to make noisy
- **noise** – [numpy array] of five elements, one for each parameter of the source

agis_functions.**attitude_from_alpha_delta**(*source*, *sat*, *t*, *vertical_angle_dev=0*)

Parameters

- **source** – [Source object]
- **sat** – [satellite object]
- **t** – [float] time
- **vertical_angle_dev** – how much we deviate from zeta

agis_functions.**calculated_field_angles**(*calc_source*, *attitude*, *sat*, *t*, *double_telescope=False*)

Ref. Paper *[LUDW2011]* eq. [12]-[13]
Return field angles according to Lindegren eq. 12. See *compute_field_angles()*

Parameters

- **source** – [Calc_source]
- **attitude** – [quaternion] attitude at time t
- **sat** – [Satellite]
- **t** – [float] time at which we want the angles
- **double_telescope** – [bool] If true, uses the model with two telescopes

Returns

- eta: along-scan field angle (== phi if double_telescope = False)
- zeta: across-scan field angle

agis_functions.**compare_attitudes**(*gaia*, *solver*, *my_times*)

Parameters

- **gaia** – [satellite object]
- **solver** – [Solver object]
- **my_times** – [list][days] list of times at which we want to compare the attitudes

Returns figure object

agis_functions.**compute_DL_da_i**(*coeff_basis_sum*, *bases*, *time_index*, *i*)

Ref. Paper *[LUDW2011]* eq. [80]
Compute derivative of the attitude deviation wrt attitude params. See *compute_coeff_basis_sum()*

Parameters

- **coeff_basis_sum** – the sum $\sum_{n=L-M+1}^{L} a_n B_n(t_L)$
- **bases** – Bspline basis, B_n(t_L) in the equation above.
- **time_index** – [int] index that will get us to return B_n(t_L). Since we stored only B_n for all the observed times t_L, it is possible to access them only with the index
- **i** – number of the base that we want (**n in the equations above**)

agis_functions.**compute_DL_da_i_from_attitude**(*attitude*, *bases*, *time_index*, *i*)

Ref. Paper *[LUDW2011]* eq. [83]
Compute derivative of the attitude deviation wrt attitude params. See *compute_coeff_basis_sum()*

Parameters

- **attitude** – [quaternion]
- **bases** – Bspline basis, B_n(t_L) in the equation above.
- **time_index** – [int] index that will get us to return B_n(t_L). Since we stored only B_n for all the observed times t_L, it is possible to access them only with the index
- **i** – number of the base that we want (**n in the equations above**)

agis_functions.**compute_attitude_deviation**(*coeff_basis_sum*)

Ref. Paper *[LUDW2011]* eq. [80]
Compute the attitude deviation from unity:

$$D_l = 1 - || \sum_{n=L-M+1}^{L} a_n B_n(t_L)||^2$$

Action Compute the attitude deviation from unity

Parameters **coeff_basis_sum** – the sum $\sum_{n=L-M+1}^{L} a_n B_n(t_L)$

Returns attitude deviation from unity

agis_functions.**compute_coeff_basis_sum**(*coeffs*, *bases*, *L*, *M*, *time_index*)

Ref. Paper *[LUDW2011]* eq. [80]
Computes the sum:

$$\sum_{n=L-M+1}^{L} (a_n \cdot B_n(t_L))$$

Parameters

- **coeffs** – [numpy array] splines coefficients
- **bases** – [numpy array] B-spline bases
- **L** – [int] left_index
- **M** – [int] spline order (= spline degree + 1)
- **time_index** – [float] time index where we want to evaluate the spline

Returns [numpy array] vector of the

agis_functions.**compute_dR_dq**(*calc_source*, *sat*, *attitude*, *t*)

Ref. Paper *[LUDW2011]* eq. [79].
Computes the derivative of the cost-function w.r.t. quaternion q i.e. the tuple of equations:

- $\frac{dR_l^{AL}}{dq_l} = 2 \cdot sec(\zeta_l) q_l * \{S' n_l, 0\}$ which is Along_scan w.r.t. observation number l
- $\frac{dR_l^{AC}}{dq_l} = -2 q_l * \{S' m_l, 0\}$

where $*$ is a quaternion multiplication

Parameters

- **calc_source** – [calc_source object]

- **sat** – [sat object]
- **attitude** – [numpy quaternion]
- **t** – [float][days] time

Returns [tuple of numpy arrays] (dR^AL/dq, dR^AC/dq)

agis_functions.**compute_deviated_angles_color_aberration**(*eta*, *zeta*, *color*, *error*)

Implementation of chromatic aberration

Parameters

- **eta** –
- **zeta** –
- **color** –
- **error** –

Returns

- eta deviated by the aberration
- zeta deviated by the aberration

agis_functions.**compute_du_dparallax**(*r*, *b_G*)

Ref. Paper *[LUDW2011]* eq. [73]
Computes $\frac{du}{d\omega}$

Parameters

- **r** – barycentric coordinate direction of the source at time t. Equivalently it is the third column vector of the "normal triad" of the source with respect to the ICRS.
- **b_G** – Spatial coordinates in the BCRS.

Returns [array] the derivative du_dw

agis_functions.**compute_field_angles**(*Su*, *double_telescope=False*)

Ref. Paper *[LUDW2011]* eq. [12]-[13]
Return field angles according to eq. [12]

Parameters

- **Su** – array with the proper direction in the SRS reference system
- **double_telescope** – [bool] If true, uses the model with two telescopes

Returns

- eta: along-scan field angle (== phi if double_telescope = False)
- zeta: across-scan field angle

agis_functions.**compute_mnu**(*phi*, *zeta*)

Ref. Paper *[LUDW2011]* eq. [69]
$S'm_l = [-sin(\phi_l), cos(\phi_l), 0]^T$

$S'n_l = [-sin(\zeta_l)cos(\phi_l), -sin(\zeta_l)\cos(\phi_l), cos(\zeta_l)]^T$
$S'u_l = [cos(\zeta_l)cos(\phi_l), cos(\zeta_l)sin(\phi_l), sin(\zeta_l)]^T$

Parameters

- **phi** – [float]
- **zeta** – [float]

Returns [array] column vectors of the S'[m_l, n_l, u_l] matrix

agis_functions.**dR_da_i**(*dR_dq*, *bases_i*)
See *compute_dR_dq()*

Parameters

- **dR_dq** – Derivative of the cost funtions w.r.t. the quaternion q
- **basis_i** – B-spline basis of index i

agis_functions.**error_between_func_attitudes**(*my_times*, *func_att1*, *func_att2*)
Computes the sum of the relative difference of the quaternion components between two attitudes

Parameters

- **my_times** – times at which the difference will be computed
- **func_att1** – [function] that returns an attitude quaternion to compare
- **func_att2** – [function] that returns an attitude quaternion to compare

Returns [float] The error in attitude (only qualitative) just for visualization

agis_functions.**extend_knots**(*internal_knots*, *k*)
Extend the knots sequence to add the external knots. This is done because *extract_coeffs_knots_from_splines()* returns only the internal knots. Therefore they should be extended on both sides by **k** knots.

Parameters

- **internal_knots** – [array] containing the internal knots
- **k** – [int] spline degree

Returns [list] containing the extended knots

agis_functions.**extract_coeffs_knots_from_splines**(*attitude_splines*, *k*)
Extract spline characteristics (coeffs, knots, splines). The spline being defined as

$$S(t) = \sum_{n=0}^{N-1} a_n B_n(t)$$

where c_n are the spline coefficients and $B_n(t)$ is the spline basis evaluated at time t. **N** is the number of coefficients. The knots are the time discritization used for the spline.

Parameters

- **attitude_splines** – list or array of splines of scipy.interpolate.InterpolatedUnivariateSpline
- **k** – [int] Spline degree

Returns

- [array] coeff

- [array] knots
- [array] splines (Bspline interpolating with degree k)

agis_functions.**generate_angles_of_sources**(*times_for_source*, *sat*, *noise_factor=1e-12*)
Create source along the path of the telescopes. For each time in times_for_source create one source on PFoV, one between the two telescope and one on the FFoV returns a number *num_sources* x3 of ICRS coordinates (right ascension, declination)

Parameters

- **num_source** – [list of floats] times where we want to create the sources
- **noise_factor** – [float]

Returns list of alphas and deltas

agis_functions.**generate_observation_wrt_attitude**(*attitude*)
Create coordinates of a star in the position of the x-axis of the attitude of the satellite

Parameters **attitude** – The attitude of the satellite

Returns [tuple of floats] [rads]right ascention and declination corresponding to the direction in which the x-vector rotated to *attitude* is pointing

agis_functions.**generate_scanned_times_intervals**(*day_list*, *time_step*)
Given a list of days, it will return the list of time that will define the intervals to be scanned

Parameters

- **day_list** – [list of floats][days]
- **time_step** – [float][days] length of the time interval

Returns [list][days] list of times to create scanned intervals

agis_functions.**get_angular_FFoV_PFoV**(*sat*, *t*)
Computes angular positions (righ ascension α, declination δ) of the fields of view as a function of time. The angles are as seen from the satellite (Co-Moving Reference System).

Parameters

- **sat** – [Satellite object]
- **t** – time at which we want the FoVs pointing directions

Returns $\alpha_{PFoV}, \delta_{PFoV}, \alpha_{FFoV}, \delta_{FFoV}$

agis_functions.**get_basis_Bsplines**(*knots*, *coeffs*, *k*, *obs_times*)

Parameters

- **knots** – [array] knots intervals for the time discretization of the spline (for one (any) of the four quaternions parameters)
- **coeffs** – [array] of the coefficients to create a spline (for one (any) of the four quaternions parameters)
- **k** – [int] spline degree
- **obs_times** – [list][days] Times of observation

Returns arrays of size (#coeffs, #obs_times)

agis_functions.**get_interesting_days**(*ti*, *tf*, *sat*, *source*, *zeta_limit*)
Computes the days in which the Fields of View may see some sources.

Parameters

- **ti** – [float] initial time at which we want to evaluate the interesting days
- **tf** – [float] final time
- **sat** – [Satellite object]
- **source** – [Source object]
- **zeta_limit** – [float][rads] zeta_limit for the field of view

Returns [list of floats] containing the days in which the source may be in the field of view of the telescopes

agis_functions.**get_left_index**(*knots*, *t*, *M*)

Parameters

- **knots** – knots for the spline
- **t** – [float][days] time at which
- **M** – [int] spline order (M=k+1)

Returns left_index the left_index corresponding to t i.e. *i* s.t. $t_i < t <= t_{i+1}$

agis_functions.**get_obs_in_CoMRS**(*source*, *sat*, *t*)

Get observation in the Co-Moving Reference System.

Parameters

- **source** – [source object]
- **sat** – [Satellite object]
- **t** –

Returns (α, δ) of the observation in CoMRS

agis_functions.**get_times_in_knot_interval**(*time_array*, *knots*, *index*, *M*)

Parameters **time_array** – [numpy array]

Returns times in knot interval defined by [index, index+M]. I.e. all the times t such that $\tau_n < t < \tau_{n+M}$ where **M** is the order of the spline (M=k+1) and τ_n is the knot number n.

agis_functions.**multi_compare_attitudes**(*gaia*, *Solver*, *my_times*)

Compares the attitudes in four different plots, one for each attitude component.

Parameters

- **gaia** – [satellite object]
- **solver** – [Solver object]
- **my_times** – [list][days] list of times at which we want to compare the attitudes

Returns figure object

agis_functions.**multi_compare_attitudes_errors**(*gaia*, *solver*, *my_times*)

Parameters

- **gaia** – [Satellite object]
- **solver** – [Solver object]
- **my_times** – [list][days] times at which we want to compare the attitudes

Returns figure

agis_functions.**observed_field_angles**(*source*, *attitude*, *sat*, *t*, *double_telescope=False*)

Ref. Paper *[LUDW2011]* eq. [12]-[13]
Return field angles according to Lindegren eq. 12. See *compute_field_angles()*

Parameters

- **source** – [Source]
- **attitude** – [quaternion] attitude at time t
- **sat** – [Satellite]
- **t** – [float] time at which we want the angles
- **double_telescope** – [bool] If true, uses the model with two telescopes

Returns

- eta: along-scan field angle (== phi if double_telescope = False)
- zeta: across-scan field angle

agis_functions.**scanning_direction**(*source*, *sat*, *t*)
Computes the y-coordinates of the SRS frame, which is an approximation of the scanning direction. Use for plotting alone

4.6 constants.py

File constants.py

Purpose File containing the constants that will be used in the other files. This will allow to avoid "magic numbers" in the code and also to easily change these constants if we later need them more or less precises

Author Luca Zampieri 2018

CHAPTER 5

Known Issues

Too many to be listed for the moment but we will try to make a good list before Christmas.

Problems in class Satellite:

- Attitude not constructed if t_init != 0

Note: If you find any issue that is not on the list, you can contact us by email and we will: * Add it to the list * Try to fix it

CHAPTER 6

Notations

Table 1: Description of the symbols used in the code

Symbol	Description
α or ra	Righ ascension
δ or dec	Declination
η or eta	Along-scan field angle see picture
ζ or zeta	Across-scan field angle see picture

6.1 Testing tables

test	Values	Examples	Description
SCI_EXTENSION	integer	2	Index of science extens

Other option Simple table

Inputs		Output
A	B	A or B
False	False	False
True	False	True
False	True	True
True	True	True

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Bibliography

[LUDW2011] The astrometric core solution for the Gaia mission *Overview of models, algorithms, and software implementation* L. Lindegren, U.Lammers, D. Hobbs, W. O'Mullane, U. Bastian and J. Hernandez link to paper

[Lind2001] Calculating the GAIA nominal scanning Law, L. Lindegren SAG-LL-35 19 february 2001 http://www.astro.lu.se/~lennart/Astrometry/TN/Gaia-LL-035-20010219-Calculating-the-GAIA-Nominal-Scanning-Law.pdf

Python Module Index

Index

Symbols

A

B

C

D

E

X

Z