PASSATA introduction

Guido Agapito* and Alfio Puglisi
*guido.agapito@inaf.it
INAF Osservatorio Astrofisico di Arcetri, Largo E. Fermi 5, Firenze, Italy

CONTENTS

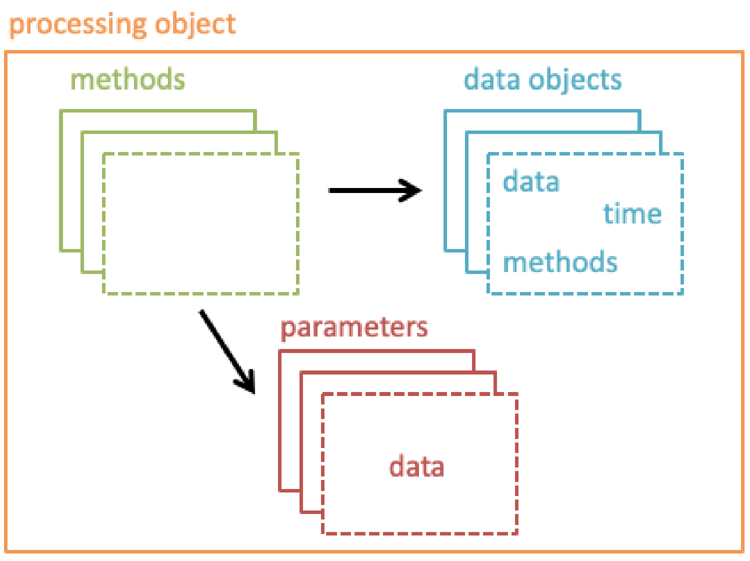
1 Introduction

Here we report an introduction to PASSATA (Agapito et al. 2016). PASSATA is based on object oriented programming and as a general introduction please refer to the IDL Object-Oriented Programming, to `IDL_Object` class and to other classes like `DICTIONARY`, `HASH`, `LIST`, `ORDEREDHASH`, ...

2 PASSATA objects

Most objects belong to one of two main categories: *data objects* and *processing objects*. *Data objects* hold static data used as input, output or intermediate data for the simulation (i.e. a ccd pixel frame or a phase screen). *Processing objects* perform calculations, and pass data objects around to communicate with each other. A few more specialized objects help in retrieving calibration data and control simulation timing.
Data objects
All simulation data is stored using data objects. The only special feature of data objects is that, in addition to holding their data, they also store the simulated time at which they were generated or updated. To do so, the routine that generates or updates the object must update the `generation_time` property passing the current simulation time. Every data object has a `time()` function method that returns the last set time. Routines that use the object as input can thus judge whether the data is current, or how much time has passed since its generation. Basic objects are provided for simple scalar or array values, lists and dictionaries. Specialized data objects must derive from the `base_data_obj` class. Specialized objects may provide functions to manipulate their data like sum, subtraction, wavelength-based rescaling where applicable, etc.

2.1 Processing objects



All calculations are done in processing objects. Each of these objects have several features:

- Data inputs and outputs are implemented with data objects. Each processing object defines one or more outputs, and receives object references for input data.
- Processing is done in a procedure with the conventional name `trigger`. This procedure has a single argument, the simulated time *t*, and will be called once for each increment of the simulated time.
- Objects are configured using *parameters*, either from a configuration file or set in simulation code, as detailed below.

Output data is represented with a data object allocated in the processing object's __*define* structure. The processing object initializes the output data object as soon as it has enough information to do so, fills it with new data and timestamp in the `trigger` procedure, and defines a function to return a reference to it.

Input data is not stored in the processing object. Since each input will be an output of some other object, the processing object will only receive a reference to it.

Processing of data is performed at discrete time intervals. At each time interval, the object's `trigger` procedure is called, with a single parameter *t*, which contains the currently simulated time. The object can act on the trigger, filling its data output objects, or can choose to do nothing if not enough time has passed (i.e. a processing object that produces output only once per second will do nothing until the *t* parameter has incremented by at least one second since the last time an output was generated). Reading of data inputs is always allowed, and their `time()` function can be used to check whether the data has been updated or not.

Configuration is done using parameters. Parameters are grouped into structures or dictionaries, each structure holding one or more parameters for a single object. Each parameter corresponds to a property and an entry in the configuration file. For example, an object with a parameter called "integration time" will implement a property called "integration_time" and the configuration structure will accept a definition like "integration time: 1.0". Usually the parameter will correspond to a single variable in the object __*define* structure.

The base object processing class implements an `apply_params` procedure that is automatically called after object creation. This procedure receives a structure or dictionary containing the object's parameters, and will execute a series of set_ calls to set their values. Parameters are conventionally separated into *static* and *runtime* sets: the static set holds parameters that are rarely changed (i.e. the primary mirror diameter). The runtime sets those that are likely to change at each simulation run (i.e. the source magnitude or seeing). Each set is written into its own file and can be loaded when necessary. Note that a set does not need to specify all object parameters: if a parameter is missing, the relevant set_* procedure will not be called, and the parameter will retain its previous value, or the default value if it was never set.

3 Setting up data transfer

Use properties instead of set/get.

Once all processing objects have been created and parameterized, the data objects needed for data exchange have already been created inside the processing objects (see the definition of *output data* above). The remaining task is to connect each input data with the corresponding data output object.

4 Loop control

The `loop_control` object coordinates the simulation, incrementing the simulated time and calling each object `trigger` procedure at each increment (`run` method). Each processing object must be added to the loop control list (with the `add` method), and they will be triggered in that order. The simulation will stop when the simulated time has reached an end point, or when a stop condition is satisfied (i.e. a certain kind of data is produced or is equal to a certain value).

5 Calibration

Generation of calibration data can be implemented as a partial loop: to generate for example a slope offset frame, the relevant processing objects are created and connected as in a real simulation, and the loop does not start until the desired data has been produced. Any special configuration can be done with a runtime parameter file as in any other simulation run.

6 Time representation

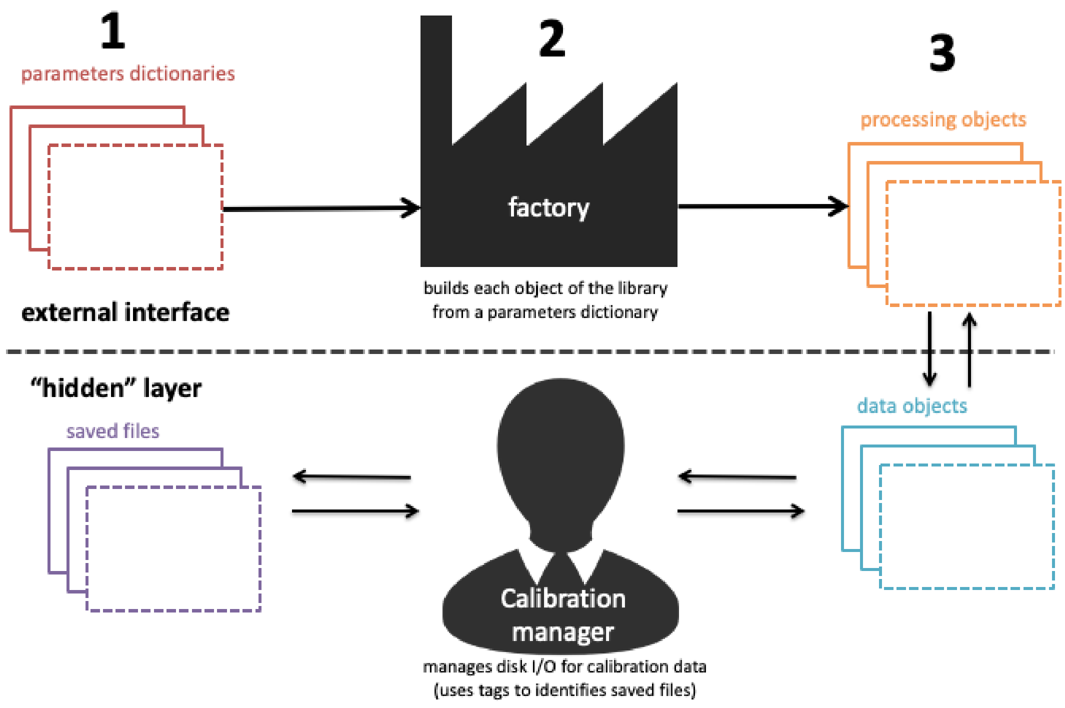
In order to avoid rounding errors, time is represented with an integer value with 64 bits (IDL type *long64*). Time resolution is by default one nanosecond, and can be optionally changed when setting up the simulation. With the default resolution, the *long64* type allows up to 9x10^9 seconds of simulated time (about three years). Note that all objects must agree on the time resolution, so changing the resolution after object creation is not allowed. All data and processing objects have a *_time_resolution* member and functions to convert from/to seconds.

7 Data storing

Use the `datastorage` class to store data during a simulation, save them to disk and restore them from disk. Then methods `values`, `times`, `mean`, `stddev`, `variance` and `plot` can be used to manage the data. Other useful methods are `keys` and `HasKey` (see IDL help of `dictionary` class).

8 Factory and Calibration Manager

There are a couple of objects that help the user to set up and manage processing and data objects: they are `factory` and `calib_manager`.


1
parameters dictionaries
2
factory
3
processing objects
builds each object of the library from a parameters dictionary
external interface
"hidden" layer
saved files
data objects
Calibration manager
manages disk I/O for calibration data (uses tags to identifies saved files)

8.1 Factory

`factory` is an object which has one method to build each object of the library. All these method names have the same prefix, *get_*, and a suffix that is the name of the class. The method needs one or more dictionaries of parameters and returns the object. If the parameters of the dictionary refer to data saved on disk the factory uses the calibration manager object to restore the data.

8.1.1 Example of get_ method for the atmo_evolution processing object

```
;+
; Builds an `atmo_evolution` processing object.
;
; :params:
; params: in, required, type=dictionary or struct
; dictionary or struct of parameters
; source_list: in, required, type=list
; list of `source` objects
;
; :returns:
; a new `atmo_evolution` processing object
;-

function factory::get_atmo_evolution, params, source_list

; checks the parameters and convert it to a dictionary if needed
params = self.ensure_dictionary(params)

; extracts some parameters from the main dictionary
pixel_pup = self._main.pixel_pupil
```

```
pixel_pitch = self._main.pixel_pitch
precision = self._main.precision

; removes some parameters from the dictionary (some have default values if they are not
defined)
L0 = params.remove('L0')
wavelengthInNm = params.remove('wavelengthInNm')
heights = params.remove('heights')
Cn2 = params.remove('Cn2')
pixel_phasescreens = self.extract(params, 'pixel_phasescreens', default=!NULL)
seed = self.extract(params, 'seed', default=1)

; extracts the directory name from the calibration manager
directory = self._cm.root_subdir('phasescreen')

; makes the object
atmo_evolution = obj_new('atmo_evolution', L0, wavelengthInNm, pixel_pitch, heights, Cn2, $
                         pixel_pup, directory, source_list, $
                         pixel_phasescreens = pixel_phasescreens, $
                         precision=precision, seed=seed)

; applies other parameters, not previously extracted/removed, to the new object
self.apply_global_params, atmo_evolution
atmo_evolution.apply_properties, params

; returns the new object
return, atmo_evolution
end
```

8.2 Calibration manager

`calib_manager` object manages disk I/O for calibration data. It uses a directory tree to distinguish different data, and read/read/restore methods of the data objects.

8.2.1 Example of sub-directories dictionary

```
self._subdirs = dictionary({ $
 phasescreen: 'phasescreens/', $ ; phase-screens
 slopenull  : 'slopenulls/', $ ; reference slope vector
 pupils     : 'pupils/', $ ; pyramid WFS pupil
 subaps     : 'subaps/', $ ; SH WFS sub-apertures index
 rec        : 'rec/', $ ; reconstruction matrix
 im         : 'im/', $ ; interaction matrix
 ifunc      : 'ifunc/', $ ; influence functions matrix (modal or zonal)
 m2c        : 'm2c/', $ ; modes-to-commands matrix
 filter     : 'filter/', $ ; IIR filters
 pupilstop  : 'pupilstop/', $ ; pupil mask
 vibrations : 'vibrations/' $ ; vibrations PSD
 })
```

8.2.2 Example of Read/Write methods

```
pro calib_manager::write_im, tag, intmat

; produces the filename with the complete path
```

```
filename = self.filename( 'im', tag, /makedirs)
intmat.tag = tag

; uses save method of the data object
intmat.save, filename

end
function calib_manager::read_im, tag

; produces the filename with the complete path
filename = self.filename( 'im', tag)

; returns !NULL if the file does not exist
if not file_test(filename) then return, !NULL

; uses restore method of the data object
return, intmat.restore(filename)

end
```

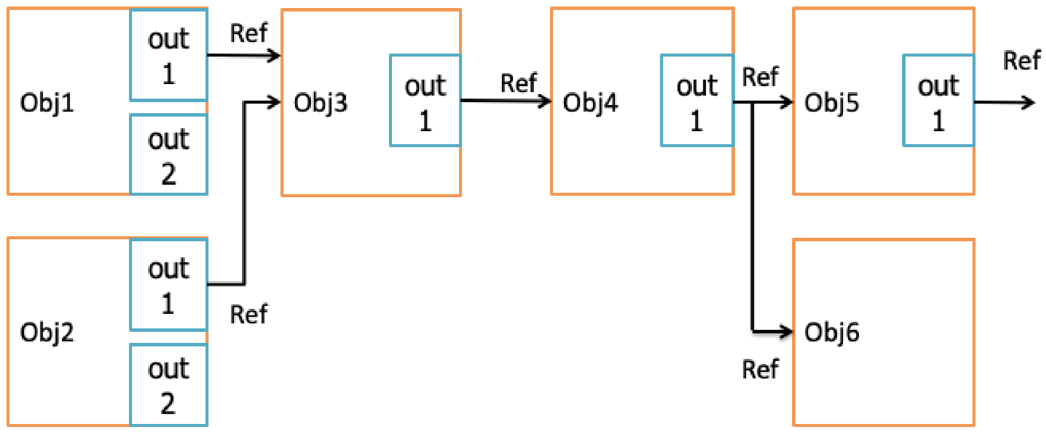
---

9 Base objects

There are some base objects in the libraries which can/must be inherited by other object to implement some general functionalities.

- `base_data_obj`: The only purpose of this class is to inherit from both `base_time_obj` and `IDL_Object`, so that data classes do not need to inherit from then, but just from this one.
- `base_dict`: Base data object for dictionaries which inherits from `base_time_obj` and dictionary.
- `base_gpu_value`: Basic data object in GPU memory. Use this object to store simple arrays in GPU memory.
- `base_list`: Base data object for lists which inherits from base_time_obj and list.
- `base_parameter_obj`: Base parameter object with `apply_properties` method.
- `base_processing_obj`: Base class for processing objects. A processing object is the basic building block of a simulation. Each processing object defines a `trigger` procedure that is called at regular intervals by the `loop_control` main object.
- `base_time_obj`: Do not use this class directly. Derive from base_data_obj or `base_processing_obj`. Class for objects that need to keep track of time generation or resolution (typically both processing and data objects). Needs to be inherited together with `IDL_Object` in order to work
- `base_value`: Basic data object. Use this object to store simple scalars or arrays without further qualifications.

10 Basic simulation scheme



A simulation is made of a list of processing objects connected together by data objects. The loop control object coordinates the simulation, and the data store object collects the data from the data objects.