ASTROLABE

The ASTROLABE tests

Version 1.0

ABSTRACT

This document describes the test programs included in the distribution of ASTROLABE.

DOCUMENT STATUS

Document title:	The ASTROLABE tests
Document reference:	The_ASTROLABE_tests_1-0_NO
Document file name:	The ASTROLABE tests_1-0_NO.odt
Nature:	<report (RP) | specification (SP) | **note (NO)** | minutes (MI) | others (OT) >
Status:	<in preparation | checked | approved | **authorized**>
Author(s):	José A. Navarro (CTTC)
Version:	1.0
Last change:	2018-11-16T10:08:18
Distribution:	<team | project | organisation | **public**>
Security classification:	<**public** | internal | confidential | secret>
Number of pages:	16

DOCUMENT CHANGE RECORD

Version	Date	Section(s) affected	Description of change, change request reference or remarks.
1.0	2018-11-16	all	First public deliverable version.

TABLE OF CONTENTS

1 INTRODUCTION

1.1 Purpose

The purpose of this document is to describe the tests included in the distribution of the ASTROLABE library.

This document does **not** describe ASTROLABE. For an exhaustive, rigorous description of the ASTROLABE data model, please refer to [1]. A gentler introduction to ASTROLABE may be found in [2] and [3]. All these documents should be part of the distribution of ASTROLABE.

1.2 Organization of this document

Section 2 describe the set of tests included in the distribution of ASTROLABE. Subsections 2.1 to 2.4 explain general concepts about these tests that, however, must be known by those developers using the ASTROLABE library. Section 2.5 describe in detail each test.

2 THE TESTS

2.1 About the tests

A total of 20 tests in source code form have been included with the distribution of the ASTROLABE library. These are:

- Tests dealing with metadata
    - `test_astrolabe_header_file_parser`
    - `test_metadata_parser`
    - `test_navdir_parser`
- Tests dealing with data—high level.
    - `test_instrument_reader`
    - `test_instrument_writer`
    - `test_observation_reader`
    - `test_observation_writer`
    - `test_parameter_reader`
    - `test_parameter_writer`
    - `test_rmatrix_reader`
    - `test_rmatrix_writer`
- Tests dealing with data—low level.
    - `test_bin_to_reverse_bin`
    - `test_bin_to_txt`
    - `test_socket_to_txt`
    - `test_synthbinsplit`
    - `test_synthtxt`
    - `test_txt_to_bin`
    - `test_txt_to_reverse_txt`
    - `test_txt_to_socket`
- Other tests.
    - `test_navfile_manager`

Note the distinction between "high" and "low" level tests with data. Although ASTROLABE offers a high level API able to deal transparently with data stored or transmitted using different formats (text of binary files, sockets), such API relies on a series of low level classes implementing such specific formats, that is, the low level API.

Regular developers should never rely on the low level API; this is so because of several reasons. First, there is no need to dive into the details of the implementation of the high level classes – which, of course, use the low level ones; moreover, high and low level classes are as equally performant, taking the high level ones care of some extra details related only to the data header files described in section 3.7.1 of [1]; finally, but not less important, low level classes do not deal with data header files, thus not implementing completely the ASTROLABE specification.

However, low level tests are included for the sake of completeness and aiming at helping developers to understand how the classes on which ASTROLABE rely work.

2.2 Record structure. Record sequence

Many of the tests related to data include, besides the main.cpp file, extra source files named `RecordStructure[,hpp/.cpp]` and `RecordSequence[.hpp/.cpp]`.

When reading or writing ASTROLABE data, it is necessary (1) to read the metadata files characterizing data and therefore (2) read or write data themselves according to the description just obtained thanks to the aforementioned metadata.

Such description includes so relevant things as, for instance, the size of the observation vector or the number of extra tags accompanying actual data (see section 3.6 of [1] for details). However, and in order to simplify the data examples, metadata is never loaded. There exist, however a single test to show how to load metadata (`test_metadata_parser`).

Since metadata is indispensable to read data, a simpler mechanism to simulate such important information has been provided. The `RecordStructure` class, present in many of the tests, perform such task. In this class, a series of (fake) types of observations (or instruments or parameters) are minimally defined, including the indispensable characterization to let the data tests work.

Therefore, the tests needing such information are able to retrieve the structural definition of some kind of data just providing its code (or type).

Note that this class, `RecordStructure`, is not part of the ASTROLABE library, but a simple method to make the examples more understandable, relieving these from the responsibility of loading metadata.

Some of the tests produce data files. The structure of the several kind of observations (or instruments, or parameters…) written in this files is, once more, obtained by means of the `RecordSequence` class. However, what kind (types) of data are written by these tests? ASTROLABE data files (see section 3.7 of [1] for details) are made of an arbitrary series of `<l>` and `<o>` records, which may refer to several kinds of observations. What types should be written? When an `<l>` record should be written? And `<o>` records? How meny of these?

`RecordSequence` provides with a synthetic sequence of `<l>` and `<o>` records, including the type of data to write (and, given a type, it is possible to discover its structure using `RecordSequence`, as stated above). Test (writing) programs may, therefore, ask the `RecordSequence` class for the kind (`<l>` or `<o>`, data kind, etc) of the next record to write. Programs writing data series longer than the maximum length of the sequence provided by `RecordSequence` may keep asking for more until no more data need to be written. Note that the sequence provided by `RecordSequence` adheres to the rules set to write ASTROLABE data files (see section 3.7.2.1 of [1] for details).

Once more, `RecordSequence` is a mechanism to simplify the tests and not a component of the ASTROLABE library.

2.3 Building the tests

For detailed instructions on how to build the tests, please refer to [4].

2.4 Data for the tests. Schemas. Units.

Many of the tests – the “writers” ones – create synthetic data sets that may be used afterwards by other ones (the classic combination of producer / consumer), so, in mots cases, no data samples are provided. In some other cases, it is mandatory to use an actual data (or metadata) file as input

All tests state clearly what are they data needs and how such data may be obtained. When actual data file samples are needed, these may be found in the test subfolder of the ASTROLABE distribution.

Tests reading data (as any application developed by the ASTROLABE user that reads ASTROLABE data) need “schemas” as input. A schema is a formal description of the syntax of some XML-based language.

ASTROLABE provides a variety of these languages, as the ones used to describe navigation metadata, navigation directories or data header files. Therefore, schemas to describe – and validate – such languages are part of the distribution.

The tests reading data, in most occasions, will want to guarantee that the data files they read are syntactically correct. To do so, they will ask for the **full** path to the corresponding schema. Note that it is **indispensable** to type the full path to the schema file. For instance:

- `C:/some_user/development/ASTROLABE/data/nav-directory_file.xsd` (Windows)
- `/home/some_user/development/ASTROLABE/data/nav-directory_file.xsd` (Linux)

would be examples of such full paths.

The schemas available in ASTROLABE are:

- `astrolabe-header_file.xsd` – Syntax of the ASTROLABE data header files (section 3.7.1 of [1]).
- `nav-directory_file.xsd` – Syntax of the ASTROLABE navigation directory (section 3.6.2 of [1]).
- `nav_metadata.xsd` – Syntax of the ASTROLABE navigation medatada files (section 3.6.1 of [1]).

All these are located in the `data` folder of the ASTROLABE distribution.

Finally, the test program `test_metadata_parser` (as well as any program developed by the ASTROLABE user that reads metadata) relies on the `udunits2` library. When this happens, it is mandatory to specify the full path to the file `udunits2.xml`.

This file, as well as:

- `udunits2-accepted.xml`
- `udunits2-base.xml`
- `udunits2-common.xml`
- `udunits2-derived.xml`
- `udunits2-prefixes.xml`

are part of the distribution of the `udunits2` library and must reside in some accessible place so any application relying on it may find them. Therefore, it is necessary to identify the precise location of the `udunits2.xml` file to provide it to any tests asking for it.

Details about building ASTROLABE and the libraries it relies on may be found in [4].

2.5 Test dealing with metadata

2.5.1 test_astrolabe_header_file_parser

- **Purpose**: this test reads an ASTROLABE data header file, the ones that describe how data is stored (see section 3.7.1 of [1]).
- **How to use it:**
  - `test_astrolabe_header_file_parser astrolabe_header_file [schema_file]`
- **Inputs:**
  - An ASTROLABE header file – Use file `data_header_example.obs` in the `test` folder of the ASTROLABE distribution.

    - The schema defining the syntax of ASTROLABE header files. This is file `astrolabe-header_file.xsd` located in the `data` folder of the ASTROLABE distribution. Provide the full path. Note that this file is **optional**.
- **Outputs:**
    - the test dumps on the screen the information found in the input ASTROLABE header file.

2.5.2 test_metadata_parser

- **Purpose**: this test reads an ASTROLABE navigation metadata file (see section 3.6.1 of [1]). All applications reading ASTROLABE data should read metadata first, as done in this example.
- **How to use it:**
    - `test_metadata_parser metadata_file UDUNITS2_units_database_file [schema_file]`
- **Inputs:**
    - An ASTROLABE navigation metadata file – Use file `navigation_metadata_example.nmd` in the `test` folder of the ASTROLABE distribution.
    - Full path to file `udunits2.xml` in the udunits2 distribution. See section 2.4 for details.
    - The schema defining the syntax of ASTROLABE navigation metadata files. This is file `nav_metadata.xsd` located in the `data` folder of the ASTROLABE distribution. Provide the full path. Note that his file is optional.
- **Outputs:**
    - the test dumps on the screen the information found in the input ASTROLABE navigation metadata file.

2.5.3 test_navdir_parser

- **Purpose**: this test reads an ASTROLABE navigation directory (see section 3.6.2 of [1]). Moreover, it writes an output file containing the data in the input one (read / write example).
- **How to use it:**
    - `test_navdir_parser input_navdir_file navdir_schema_file output_navdir_file`
- **Inputs:**
    - An ASTROLABE navigation directory file – Use any of the navigation directory sample files in the file `test` folder of the ASTROLABE distribution (`nav_directory_bw.ndf`, `nav_directory_fw.ndf` or `nav_directory_cb.ndf`).
    - The schema defining the syntax of ASTROLABE navigation directory files. This is file `nav-directory_file.xsd` located in the `data` folder of the ASTROLABE distribution. Provide the full path.
- **Outputs:**
    - the test dumps on the screen the information found in the input ASTROLABE navigation directory file.
    - The input file is copied, using ASTROLABE classes, to the output file.

2.6 Test dealing with data – high level

2.6.1 About the high level data tests

As stated in section 2.1, two kinds of tests handling data have been included in the ASTROLABE distribution. This is the first one, dealing with the high level API, **the one recommended to developers** using this library.

Although there are eight tests (four writers, four readers), all of them work in the same way, so a common description will be provided for the sake of simplicity. Note that, additionally, these tests work in couples; that is, the output of the writers is the input of the readers. The output of one kind of writer (for example, **test_observation_writer**) may not be used by a reader of another kind (that is, cannot be used by, for instance, **test_rmatrix_reader**) since these represent different data entities.

All the tests (but those related to instruments, see later) may write / read data using three different channels: binary or text files and sockets. If a writer creates data of some kind of format (for instance, a binary file) the reader must consume the very same kind of data (that is, select the binary file option too). Instrument readers and writers do not deal with binary file formats, since these are not part of the ASTROLABE specification: only text files and sockets are taken into account.

When working with sockets both readers and writers may work as servers or clients. This is an option that may be selected when running the test. Note that the server (no matter whether it is the reader or the writer) must be started first, then the client.

Two command line windows will be necessary when working with sockets. One for the server, one for the client. Use `localhost` as the server when requested for this information. Use a high port number as 2000 to avoid port clashes with predefined services. Note that, in Windows environment, an operating system warning may show up when running servers (that is, software accepting connections). Such a warning will ask for the user permission to allow the tool to run, avoiding any firewall limitation that might exist.

Although obvious, the four couples of tests are:

- `test_instrument_writer / test_instrument_reader`
- `test_observation_writer / test_observation_reader`
- `test_parameter_writer / test_parameter_reader`
- `test_rmatrix_writer / test_rmatrix_reader`

For the sake of simplicity, these couples will be referred to as "test_*whatever*_writer" and "test_*whatever*_writer" in the descriptions provided below, where *whatever* stands for instrument, observation, parameter or rmatrix.

2.6.2 test_*whatever*_writer

Please, see the end of the section 2.6.1 for a description of the meaning of "whatever" in this context.

- **Purpose**: this test writes an ASTROLABE *whatever* file (see section 3.7.2 of [1] for a description of the instrument, observation, parameter, and correlation matrix data files).
- **How to use it:**
  - `test_`*`whatever`*`_writer`
- **Inputs:**
  - The user selects what kind of output device (format) will be used: ext file / binary file / socket. Instrument writers, however, do not produce binary files – these are an exception.
  - Depending on the kind of output selected, specific data is requested. For files, header and raw data file names. For sockets, whether to work as a client or server, the port number to

use and the name of the host to connect to in case of a client connection.

- **Outputs:**
  - Either a file or a stream of data transmitted through a socket connection.

2.6.3 *test_whatever*_reader

Please, see the end of the section 2.6.1 for a description of the meaning of “whatever” in this context.

- **Purpose**: this test reads an ASTROLABE *whatever* file (see section 3.7.2 of [1] for a description of the instrument, observation, parameter, and correlation matrix data files).
- **How to use it:**
  - `test_whatever_reader`
- **Inputs:**
  - An ASTROLABE header file.
    - When working with sockets, use file `data_header_example.obs` in the `test` folder of the ASTROLABE distribution; change the port number to match the one used by the `test_whatever_writer` counterpart if necessary (port 2000 is recommended and set in the example header file).
    - When working with files, either binary or text, use the one created by the `test_whatever_writer` counterpart.
  - The schema defining the syntax of ASTROLABE header files. This is file `astrolabe-header_file.xsd` located in the `data` folder of the ASTROLABE distribution. Provide the full path.
  - Only when working with sockets, the user will have to decide whether the tool will work in client or server mode. Please use the mode complementary to that used by the `test_whatever_writer` counterpart (that is, if the counterpart works as a client, this test must work as a server, and conversely).
  - Finally, the names of the header and raw data output files will be requested.
- **Outputs:**
  - An ASTROLABE whatever file, which in fact consists of a data header and raw data files.

2.7 Test dealing with data – low level

2.7.1 About the high level data tests

As stated in section 2.1, two kinds of tests handling data have been included in the ASTROLABE distribution. This is the second one, dealing with the low level API. **This low-level API should never be used;** the examples provided here are targeted at developers of the ASTROLABE library only and should never be used by their regular users.

The tests in this group do not rely on actual data samples provided in the distribution of the library. On the contrary, several of the tests create or transmit synthetic data files (either text or binary) or streams (socket connection) by themselves. These files or streams are generated using some auxiliary classes - that are not part of the ASTROLABE library – as described in section 2.2. Therefore, all the tools that fall in the “readers” category may be fed by the output of these synthetic writers.

Furthermore, the tests in this group do not handle ASTROLABE header files but raw data files only (this is so because they use the low-level API classes, which completely ignore header files). Therefore, XML schemas will never be necessary (so the handling of the tools is easier for the user).

These tools were originally used to check the correctness of the several low level classes included in the library. They may be chained in such a way that the output of the last one may be a file whose contents must be exactly the same than the original input of the first tool in the chain. This would prove the correctness of all the tools in the chain.

For instance, this is a possible chain:

1. `test_synthtxt` (creates a synthetic raw text data file).
2. `test_txt_to_bin` (converts a raw text data file to raw binary data file).
3. `test_bin_to_txt` (from binary to text).

And this is another one:

1. `test_synthtxt` (creates a synthetic raw text data file).
2. `test_txt_to_reverse_txt` (reverses the ordering of data within epochs).
3. `test_txt_to_bin` (from text to binary).
4. `test_bin_to_txt` (from binary to text).
5. `test_txt_to_reverse_txt` (reverses data in epochs once more).
6. `test_socket_to_txt` (reads a raw data text file through a socket connection, writing a raw data text file containing such data as output).
7. `test_txt_to_socket` (sends a raw data text file through a socket connection).

Note the apparent incorrect sequence regarding `test_socket_to_txt` and `test_txt_to_socket`. Logic says that these two tools should be run in just the opposite order. However, this is so because `test_socket_to_txt` must be started first, since it works as a server; then, when `test_txt_to_socket` (working as a client) is started it finds someone wishing to hear him. If the sequence would be reversed, `test_txt_to_socket` would complain about not being able to connect.

The sole exception to the rule that everything may be combined with everything else is `test_synthbinsplit`. This test creates a split binary file and there is no other tool in the low level API set able to read these.

Any chain must be started by `test_synthtxt`. Binary files may be thus obtained using `test_txt_to_bin`. These are the two sources of input data for any of the other tools (but `test_synthbinsplit`).

When working with socket-related tools, use `localhost` as the host to connect to and 2000 as the port to use to transmit data. Other ports would equally work, but 2000 is safe in the sense that it is – normally – not used by other services like ftp. When running `test_socket_to_txt` in Windows computers, a warning about permissions (firewall) might show up; if so, please accept the set of permissions requested, since otherwise the tool would not work properly. Note that two command line windows will be necessary to run the two complementary socket tests (one for the client, another for the server test).

2.7.2 test_bin_to_reverse_bin

- **Purpose**: Reverse the order of data within each epoch in a raw binary file.
- **How to use it:**
  - `test_bin_to_reverse_bin input_bin_file_name output_bin_file_name`
- **Inputs:**

    - A raw binary data file.
- **Outputs:**
    - Another raw binary data file, where the order of records within epochs has been reversed.

2.7.3 test_bin_to_txt

- **Purpose**: Convert a raw data file from text to binary format.
- **How to use it:**
    - `test_bin_to_txt input_bin_file_name output_txt_file_name`
- **Inputs:**
    - A raw binary data file.
- **Outputs:**
    - A raw text data file, whose contents is equivalent to the input text one.

2.7.4 test_socket_to_txt

- **Purpose**: Read an ASTROLABE raw data socket stream and save it as a raw data text file.
- **How to use it:**
    - `test_socket_to_txt port_number output_txt_file_name`
    - This test must be run in parallel with test_txt_to_socket (each in a command line window). This tool must be started IN THE FIRST PLACE, since it works as a server.
- **Inputs:**
    - The port number to receive data through (must be in concordance with the one used by this tool's counterpart, test_txt_to_socket). Port 2000 is recommended.
- **Outputs:**
    - A raw text data file, containing the data received through the socket connection.

2.7.5 test_synthbinsplit

- **Purpose**: Create a synthetic binary raw file, split into several fragments (to test the splitting capabilities of the library).
- **How to use it:**
    - `test_synthbinsplit`
- **Inputs:**
    - The user must type the base name of the output set of binary raw data files. Such base name will be modified according to ASTROLABE's naming conventions to accommodate the creation of three files (the fragments).
    - The user must also type the name of a raw text data file where the same information will be written but in text form – so it is possible to check what's been written to the output files visually.
- **Outputs:**

  - A set of three binary files, making together a single one (split files).

2.7.6 test_synthtxt

- **Purpose**: Create a synthetic text raw file. This is the way to create data to play with.
- **How to use it:**
  - `test_synthtxt output_txt_file_name`
- **Inputs:**
  - None
- **Outputs:**
  - A raw text data file.

2.7.7 test_txt_to_bin

- **Purpose**: Converts a raw text data file into its binary equivalent.
- **How to use it:**
  - `test_txt_to_bin input_txt_file_name output_bin_file_name`
- **Inputs:**
  - A raw text data file.
- **Outputs:**
  - A raw binary file data file, whose contents is equivalent to that stored in the input file.

2.7.8 test_txt_to_reverse_txt

- **Purpose**: Reverse the order of data within each epoch in a raw text file.
- **How to use it:**
  - `test_txt_to_reverse_txt input_txt_file_name output_txt_file_name`
- **Inputs:**
  - A raw text data file.
- **Outputs:**
  - Another raw text data file, where the order of records within epochs has been reversed.

2.7.9 test_txt_to_socket

- **Purpose**: Send the contents of an ASTROLABE raw text data file through a socket stream.
- **How to use it:**
  - `test_txt_to_socket input_txt_file_name server_name_or_IP_address port_number`
  - This test must be run in parallel with test_socket_to_txt (each in a command line window).

This tool must be started IN SECOND PLACE, since it works as a client.

- **Inputs:**
  - The name of the raw text data file whose data must be sent through the socket connection.
  - The name of the server where test_socket_to_txt (this tool's counterpart) is running. It is recommended to run both tools in the same computer. Therefore, this input may be "localhost" (without quotes).
  - The port number to send data through (must be in concordance with the one used by this tool's counterpart, test_socket_to_txt). Port 2000 is recommended.
- **Outputs:**
  - A raw text data file, containing the data received through the socket connection.

2.8 Other tests

2.8.1 About the remaining tests

This section describes those tests that do not fit in the former categories (metadata, high and low level data tests).

2.8.2 test_navfile_manager

- **Purpose**: To show how to create (compress) and restore (decompress) navigation files, using up to three navigation directory files, for forward, backward, combined – block and smoothing – operation modes. To do it, some fake files (those referenced in the navigation directory files used) will be created, compressed (this is the navigation file) and decompressed in a temporal folder chosen by the user.
- **How to use it:**
  - `test_navfile_manager`
  - Create an empty temporal folder. Name it as you wish.
  - Open a command line prompt. Change directory to some folder of your choice.
  - Copy in this directory the tree navigation directory files included as samples (`nav_directory_bw.ndf,` `nav_directory_fw.ndf` and `nav_directory_cb.ndf`). in the `test` folder of the ASTROLABE distribution. **Do not change** the names of these files!
  - Provide the several inputs required interactively by the tool. When asked for the path to the folder to decompress the navigation file, type the full path to the temporal folder created above.
- **Inputs:**
  - Up to three navigation directory files, that must reside in the current default directory (see "How to use it" above.
  - The schema defining the syntax of ASTROLABE navigation directory files. This is file `nav-directory_file.xsd` located in the `data` folder of the ASTROLABE distribution. Provide the full path.
  - The operation mode. Depending on the user selection (forward, backward, combined block, combined smoothing) more or less navigation directory files (the ones copied in the default directory) will be needed.

  - Name of the output navigation file (this is a zipped file, in spite of the recommended `.nf` extension).
  - Path to the folder where the former navigation file will be decompressed.
- **Outputs:**
  - A navigation file.
  - The navigation file, decompressed, in the folder selected by the user (the files there are fake ones, created by the test program; they have no actual contents and their names are obtained from the sample navigation directory files).

3 BIBLIOGRAPHY

[1] Parés, M. E., Navarro, J. A., Colomina, I. 2017. “ASTROLABE version 1.0 Interface Control Document”.

[2] Navarro, J. A.; Parés, M.E.; Colomina, I.; Blázquez, M. "A generic, extensible data model for trajectory determination systems", in Proceedings of the 3rd International Conference on Geographical Information Systems Theory, Applications and Management (GISTAM 2017), pp. 17-25, 27-28 April 2017, Porto (Portugal). doi:10.5220/0006258400170025

[3] Navarro, J. A.; Parés, M.E.; Colomina, I.; Blázquez, M. “ASTROLABE: A Rigorous, Geodetic-Oriented Data Model for Trajectory Determination Systems”. ISPRS Int. J. Geo-Inf. 2017, 6, 98.

[4] Navarro, J. A. “Building the ASTROLABE library”. 2018. (Official ASTROLABE documentation)