chimera

Release 0.1.1

February 14, 2016

Contents

CHAPTER 1

Chimera: Observatory Automation System

Chimera is a package for control of astronomical observatories, aiming at the creation of remote and even autonomous ("robotic") observatories in a simple manner. Using **Chimera** you can easily control telescopes, CCD cameras, focusers and domes in a very integrated way.

Chimera is:

Distributed Fully distributed system. You can use different machines to control each instrument and everything will looks like just one.

Powerful Very powerful autonomous mode. Give a target and exposure parameters and Chimera does the rest.

Hardware friendly Support for most common off-the-shelf components. See plugins page for supported devices.

Extensible Very easy to add support for new devices. See plugins page for more information.

Flexible Chimera can be used in a very integrated mode but also in standalone mode to control just one instrument.

Free It's free (as in *free beer*), licensed under the GNU license.

A Python Package All these qualities are the consequence of the chosen programming language: Python.

1.1 Getting Started

1.1.1 Prerequisites

Your platform of choice will need to have the following software installed:

- Python 2.7; **Chimera** has not been ported to Python3 yet.
- Git;

1.1.2 Installation

Current build status: **Chimera** currently lives in Github. To install it, go to your install directory, and run:

```
pip install git+https://github.com/astroufsc/chimera.git
```

This will clone the official repository and install into your system. Make sure that you have the right permissions to do this.

Distutils will install automatically the following Python dependencies:

- astropy
- PyYAML: 3.10
- Pyro: 3.16
- RO: 3.3.0
- SQLAlchemy: 0.9.1
- numpy: 1.8.0
- pyephem: 3.7.5.2
- python-dateutil: 2.2
- suds: 0.4

1.1.3 Alternative Methods

Alternatively, you can follow the how-tos below to install it on a virtual enviroment and on Windows.

Windows using Anaconda Python distribution

These steps were tested with Anaconda version 2.3.0.

- Download and install the latest Anaconda version for Windows.
- Download and install git for windows at https://msysgit.github.io/
- Install Visual C++ 9.0 for Python 2.7 (for pyephem package): https://www.microsoft.com/en-us/download/details.aspx?id=44266
- Open the Anaconda Command Prompt and install chimera using pip:

```
pip install git+https://github.com/astroufsc/chimera.git
```

- After install, you can run chimera and its scripts by executing

```
python C:\Anaconda\Scripts\chimera -vv
```

On the first run, chimera creates a sample configuration file with fake instruments on `%HOMEPATH%\chimera\chimera.config`

- **Optional:** For a convenient access create a VBS script named `chimera.vbs` on Desktop containing:

```
CreateObject("Wscript.Shell").Run("C:\Anaconda\python.exe C:\Anaconda\Scripts\chimera -vvvvv")
```

Python virtual environment

For those constrained by limited access to their platform, restrictions to the system provided python or any other reason, the python tool virtualenv provides an isolated environment in which to install **Chimera**.

- Install virtualenv;
- Go to your install dir, and run:

```
    virtualenv v_name
```

- This will generate a directory named *v_name*; go in and type

```
source bin/activate
```

(See the documentation for details).

- From tyhe same directory, you can now proceed to install as described above.

1.2 Using Chimera

Chimera provides a few features to make your life easier when getting started with the system:

- reasonable defaults;
- fake devices as default when no configuration is supplied;
- minimum boilerplate on command line once configured;
- an easy to read/write configuration format: YAML: YAML Ain't Markup Language.

Once installed, Chimera provides a few command line programs:

- **chimera**
- **chimera-tel**
- **chimera-cam**
- **chimera-dome**
- **chimera-focus**
- **chimera-sched**

1.2.1 Starting Chimera

To start the server component of the software, run:

```
chimera [-v|v]
```

This will start the server, with either the device set described in the configuration file or the set of default ones provided if no configuration is present.

1.2.2 Using the Chimera scripts

Every script has a *–help* option that displays usage information in great detail; here we will provide a few examples and/or use cases for your every day observing needs.

Additionally, all **chimera** scripts have a common set of options:

--version	show program's version number and exit
-h, --help	show this help message and exit
-v, --verbose	Display information while working
-q, --quiet	Don't display information while working [default=True]

chimera-cam

Say you want to take two frames of 10 seconds each and save to file names like `fake-images-XXXX.fits`:

```
chimera-cam --frames 2 --exptime 10 --output fake-images
```

chimera-dome

In routine operations, the dome and the telescope devices are synchronized; it is however possible to move either independently:

```
chimera-dome --to=AZ
```

chimera-tel

Slew the scope:

```
chimera-tel --slew --object M5
```

As noted before, the dome will follow the telescope's position automatically. If the dome is still moving, **`chimera-cam`** will wait until the dome finishes:

```
chimera-cam --frames 1 --filters R,G,B --interval 2 --exptime 30
```

After about one and a half minute, you'll have three nice frames of M5 in R, G and B filters, ready to stack and make nice false color image.

chimera-filter

This script controls a configured (or fake) filter wheel:

```
chimera-filter [-F|--list-filters]

chimera-filter [-f |--set-filter=] FILTERNAME
```

The former command will list the filters configured in **`chimera`** (or the fakes), the latter moves the filter wheel to the position referred to by the filter's name.

chimera-sched

This scripts controls a configured scheduler controller:

```
chimera-sched --new -f my_objects.txt
```

For example, creates a new observation queue with the objects from `my_objects.txt` file. For more information about the scheduler types, please check `chimera-sched --help`.

1.3 Chimera Configuration

1.3.1 Introduction

For real world use, **`chimera`** needs to be configured for the subset of devices that comprise the *Observatory* you are driving. This encompasses:

- configuration of the server;
- description of the **controllers**;
- definition of the **instruments**;

1.3.2 The configuration file

All these components are configured in one file, located under a directory *.chimera* under your homedir; these are automatically generated for you the first time **`chimera`** is run, if they don't already exist.

The file syntax is very simple: it uses YAML, a very common format. Here is the default one:

```
chimera:
  host: 127.0.0.1
  port: 7666

site:
  name: CTIO
  latitude: "-70:48:20.48"
  longitude: "-30:10:04.31"
  altitude: 2187
  flat_alt: 80
  flat_az : 10

telescope:
  name: fake
  type: FakeTelescope

camera:
  name: fake
  type: FakeCamera
  use_dss: True
  filters: "U B V R I"

focuser:
  name: fake
  type: FakeFocuser

dome:
  name: fake
  type: FakeDome
  mode: track
  telescope: /FakeTelescope/fake

weatherstation:
  type: FakeWeatherStation
  name: fake

controller:
  - type: Autofocus
    name: fake
    camera: /FakeCamera/fake
    filterwheel: /FakeFilterWheel/fake

  - type: ImageServer
    name: fake
```

```
    httpd: True
    autoload: False
```

Configuration syntax

- Each section header goes in a line of its own, no spaces before nor after;
- Each subitem goes in a new line, indented; *no blank lines in between*;
- If a main item has more than one subitem, they are falgged by prepending a "- " to each.

With these rules in mind, lets examine the example above.

Server configuration

```
chimera:
    host: 127.0.0.1
    port: 7666
```

The server (the host where you ran the *chimera* script), is identified by the section header; it is followed by indented parameters *host* and *port*, indicating the network address:port of the server (remember chimera has distributed capabilities).

Site configuration

```
site:
    name: CTIO
    latitude: "-30:10:4.31"
    longitude: "-70:48:20.48"
    altitude: 2212
    flat_alt: 80
    flat_az : 10
```

This section describes your observatory's geolocation and the position for dome flats. Note the site coordinates are quoted.

Instruments configuration

Every defined instrument carries a number of configuration options; please refer to the *Advanced Chimera Configuration* section for details.

Controllers Configuration

The controller section is slightly different in the sense that it allows for subsections; the same syntax rules apply. Once again, for a detailed description of options, see the *Advanced Chimera Configuration* section.

1.4 Chimera Plugins

Plugins are the way to add support to chimera. Plugins are divided in two categories: instruments and controllers. Instruments are to add supported hardware by chimera and controllers are the high-level interfaces.

If you are interested on developing a new plugin for chimera, please take look at our chimera for devs page.

Note: If you need support for any device which Chimera's doesn't support, call us and we can try to develop it or help you to do it.

1.4.1 Instruments

For details on installation, configuration and an updated list of tested devices, please follow the link to the plugin page.

- chimera-apogee: For Apogee Imaging Systems cameras and filter wheels.
- chimera-ascom: For ASCOM standard compatible devices.
- chimera-astelco: For ASTELCO TPL2 communication standard telescopes.
- chimera-avt: For Allied Vision video cameras.
- chimera-bisque: For Software Bisque's TheSky versions 5 and 6 telescope control software.
- chimera-fli: For Finger Lakes Instrumets cameras and filter wheels.
- chimera-jmismart232: For JMI Smart 232 focusers.
- chimera-meade: For MEADE GOTO telescopes.
- chimera-optec: For OPTEC focusers.
- chimera-sbig: For Santa Barbara Instruments Group cameras and filter wheels.

1.4.2 Controllers

- chimera-autofocus: Focus your telescope automatically every time you need. *On the main chimera package*
- chimera-autoguider: Easy automatic guiding using chimera.
- chimera-gui: A simple Graphical User Interface to chimera. *On the main chimera package*
- chimera-headers: Template plugin to modify FITS header keywords. *Advanced use*
- chimera-pverify: Verify the telescope pointing accuracy easily. *On the main chimera package*
- chimera-skyflat: Wakes the telescope at the right time and make the exposure time calculations to make automatic sky-flatting.
- chimera-stellarium: Integrates Stellarium ephemeris software with chimera.
- chimera-webadmin: Start/Stop/Resume your robotic observatory from a web page.
- chimera-xephem: Integrates XEphem ephemeris software with chimera.

1.5 Chimera Advanced Usage

1.5.1 Chimera Concepts

These are terms commonly found within the software; they represent concepts that are important to understand in order to fully exploit **Chimera**'s capabilities.

Manager: This is the python class that provides every other instance with the tools to be able to function within :program:chimera: initialization, life cycle management, distributed networking capabilities.

ChimeraObject: In order to facilitate the administration of objects, the **Manager** functionality among other utilities is encapsulated in a **ChimeraObject** class. *Every object in chimera should subclass this one.* More details are available in *Chimera objects*.

Location: Every chimera object running somewhere is accessible via a URI style identifier that uniquely *locates* it in the distributed environment; it spells like: [host:port]/ClassName/instance_name[?param1=value1,...].

The host:port may be left out if the referred object is running in the *localhost*, and/or have been defined in the configuration file.

1.5.2 Advanced Chimera Configuration

Every **ChimeraObject** has a *class attribute*, a python dictionary that defines possible configuration options for the object, along with sensible defaults for each. This attribute, named `__config__`, can be referred to when looking for options to include in the *configuration file*. For example, the telescope interface default `__config__`:

```
    __config__ = {"device": "/dev/ttyS0",
                  "model": "Fake Telescopes Inc.",
                  "optics": ["Newtonian", "SCT", "RCT"],
                  "mount": "Mount type Inc.",
                  "aperture": 100.0,  # mm
                  "focal_length": 1000.0,  # mm unit (ex., 0.5 for a half length focal reducer)
                  "focal_reduction": 1.0,
                  }
```

can have attribute members overwritten and/or added from the plugin and from the configuration file and the others will keep their default values.

For example, on the meade plugin, besides the default options listed above, we add the configuration option on the instrument class:

```
__config__ = {'azimuth180Correct': True}
```

and, on the configuration, we can change the defaults to a different value:

```
# Meade telescope on serial port
telescope:
    driver: Meade
    device:/dev/ttyS1       # Overwritten from the interface
    my_custom_option: 3.0   # Added on configuration file
```

1.5.3 Default configuration parameters by interface type

- **Site**

```
    __config__ = dict(name="UFSC",
                      latitude=Coord.fromDMS("-23 00 00"),
                      longitude=Coord.fromDMS(-48.5),
                      altitude=20,
                      flat_alt=Coord.fromDMS(80),
                      flat_az=Coord.fromDMS(0))
```

- Auto-focus

```
__config__ = {"camera": "/Camera/0",
              "filterwheel": "/FilterWheel/0",
              "focuser": "/Focuser/0",
              "max_tries": 3}
```

- Autoguider

```
__config__ = {"site": '/Site/0',             # Telescope Site.
              "telescope": "/Telescope/0",   # Telescope instrument that will be guided by the au
              "camera": "/Camera/0",         # Guider camera instrument.
              "filterwheel": None,           # Filter wheel instrument, if there is one.
              "focuser": None,               # Guider camera focuser, if there is one.
              "autofocus": None,             # Autofocus controller, if there is one.
              "scheduler": None,             # Scheduler controller, if there is one.
              "max_acquire_tries": 3,        # Number of tries to find a guiding star.
              "max_fit_tries": 3}            # Number of tries to acquire the guide star offset be
```

- Camera

```
__config__ = {"device": "Unknown",             # Bus address identifier for this camera. E.g. USB,
              "ccd": CCD.IMAGING,              # CCD to be used when multiple ccd camera. IMAGING
              "camera_model": "Unknown",       # Camera model string. To be used by metadata purpo
              "ccd_model": "Unknown",          # CCD model string. To be used by metadata purposes
              "ccd_saturation_level": None,    # CCD level at which arises saturation (in ADUs).
                                               # Needed by SExtractor when doing auto-focus, autog

              # WCS configuration parameters   #
              "telescope_focal_length": None,  # Telescope focal length (in millimeters)
              "rotation": 0.                   # Angle between the North and the second axis of th
                                               # positive to the East (in degrees)

              }
```

- Dome

```
__config__ = {"device": "/dev/ttyS1",
              "telescope": "/Telescope/0",
              "mode": Mode.Stand,

              "model": "Fake Domes Inc.",
              "style": Style.Classic,

              'park_position': Coord.fromD(155),
              'park_on_shutdown': False,
              'close_on_shutdown': False,

              "az_resolution": 2,  # dome position resolution in degrees
              "slew_timeout": 120,
              "abort_timeout": 60,
              "init_timeout": 5,
              "open_timeout": 20,
              "close_timeout": 20}
```

- Filter wheel

```
__config__ = {"device": "/dev/ttyS0",
              "filter_wheel_model": "Fake Filters Inc.",
              "filters": "R G B LUNAR CLEAR"  # space separated filter names (in position order)
              }
```

- Focuser

```
                    FocuserAxis.V: FocuserFeature.CONTROLLABLE_V,
                    FocuserAxis.W: FocuserFeature.CONTROLLABLE_W,
                    }

class InvalidFocusPositionException(ChimeraException):
```

- Point Verify

```
    __config__ = {"camera": "/Camera/0",            # Camera attached to the telescope.
                  "filterwheel": "/FilterWheel/0",  # Filterwheel, if exists.
                  "telescope": "/Telescope/0",      # Telescope to verify pointing.

                  "exptime":  10.0,                 # Exposure time.
                  "filter":  "R",                   # Filter to expose.
                  "max_fields": 100,                # Maximum number of Landlodt fields to use.
                  "max_tries": 5,                   # Maximum number of tries to point the telescope
                  "dec_tolerance": 0.0167,          # Maximum declination error tolerance (degrees).
```

- Telescope

```
    __config__ = {"device": "/dev/ttyS0",
                  "model": "Fake Telescopes Inc.",
                  "optics": ["Newtonian", "SCT", "RCT"],
                  "mount": "Mount type Inc.",
                  "aperture": 100.0,  # mm
                  "focal_length": 1000.0,  # mm unit (ex., 0.5 for a half length focal reducer)
                  "focal_reduction": 1.0,
                  }
    __config__ = {"timeout": 30,  # s
                  "slew_rate": SlewRate.MAX,
                  "auto_align": True,
                  "align_mode": AlignMode.POLAR,
                  "slew_idle_time": 0.1,  # s
                  "max_slew_time": 90.0,  # s
                  "stabilization_time": 2.0,  # s
                  "position_sigma_delta": 60.0,  # arcseconds
                  "skip_init": False,
                  "min_altitude": 20}
    __config__ = {"default_park_position": Position.fromAltAz(90, 180)}
```

- Weather Station

```
    __config__ = {"device": None,           # weather station device
                  "model": "unknown",     # weather station model
                  }
```

1.5.4 Fake Instruments default configuration parameters

- Camera

```
    __config__ = {"use_dss": True,
                  "ccd_width": 512,
                  "ccd_height": 512}
```

1.6 Developing for Chimera

1.6.1 Introduction

Chimera uses a client/server model coupled with remote procedure call (RPC) system. Chimera defines all its entities in terms of objects.

To be a valid Chimera object a class must extend `ChimeraObject` class. `ChimeraObject` class implements `ILifeCycle` interface which defines basic methods every object must have to be started and stopped. `ChimeraObject` provides a basic implementation of a control loop, a common structure in control programs.

By itself, a Chimera object is just a static bunch of code that inherits from a specific class. To be useful, every `ChimeraObject` must have a `Manager` to manage its life cycle.

Note: We call it instrument, controller or driver purely from a semantic point of view, as they are equal from a code point of view, there is no different base class for different kinds of objects. Every instrument, controller or driver extends `ChimeraObject`. More specifically, it must implement `ILifeCycle`, and `ChimeraObject` provides us with a basic implementation.

The `Manager` class is responsible for object initialization, life cycle (start, stop) and proxy creation. `Manager` is also our server in the client/server model. It's a server of objects. We can think of `Manager` as a pool of objects available to be used.

As we need RPC support for every object and want to make the system easy to use and write, we use a Proxy class that handles all the networking for us. This way, you don't have to write networking code on your objects, just the real action, Proxy and friends add networking for you.

Before describing `Manager`'s responsibilities in more detail, let's describe all features we can have in a Chimera object.

1.6.2 Chimera objects

A Chimera object, as already said, is a normal Python class which extends from a specific base class, `ChimeraObject`, the simplest `ChimeraObject` is the following.

```
from chimera.core.chimeraobject import ChimeraObject

class Simplest (ChimeraObject):

    def __init__ (self):
        ChimeraObject.__init__(self)
```

this object has no methods, configuration or events, so it's the simplest and dumbest possible object.

As Python doesn't call base constructors per se, you need to call the base constructor from Simplest constructor (`__init__()` method).

Chimera uses the concept of `Location` all over the code. A Location is much like an URL, but without a scheme. Locations identify specific class instances running somewhere. The basic format is the following:

```
[host:port]/Classname/instance_name[?param1=value1,...]
```

host and *port*, optional fields, tell Chimera where to look for this particular object. *Classname* is the class name of the object and *instance_name* the name given to a specific instance running on host:port. When you add objects to `Manager`, you must specify a name. Also, you can pass configuration parameters as comma separated param=value pairs.

Let's write down a class that uses this and see how to actually use this from Chimera.

```
from chimera.core.chimeraobject import ChimeraObject

class Example1 (ChimeraObject):

    __config__ = {"param1": "a string parameter"}

    def __init__ (self):
        ChimeraObject.__init__(self)

    def __start__ (self):
        self.doSomething("test argument")

    def doSomething (self, arg):
        self.log.warning("Hi, I'm doing something.")
        self.log.warning("My arg=%s" % arg)
        self.log.warning("My param1=%s" % self["param1"])
```

This example requires some explanations, but before, let's run it using **chimera** script. **chimera** script is a script to initialize `Manager` and add objects either from Locations given on command line or from a configuration file.

To follow Chimera conventions, a file with a class named `Example1` must be saved to a file name `example1.py` to allow Chimera ClassLoader to find it. We may simplify this in the future.

You can save this file anywhere on your system, let's suppose you saved it on your $HOME directory.

To run it, call **chimera** this way:

```
$ chimera -I $HOME -i /Example1/example
```

You'll see something like this:

```
[date] WARNING chimera.example1 (example1) example1.py:15 Hi, I'm doing something.
[date] WARNING chimera.example1 (example1) example1.py:16 My arg=test argument
[date] WARNING chimera.example1 (example1) example1.py:17 My param1=a string parameter
```

You should use Ctrl+C (SIGINT) to stop **chimera**.

The -I on chimera tells Chimera where to look for instruments, this case to look in your $HOME directory (you can use any directory there, '.' for example). Then to -i we pass a valid Chimera Location. From the Location, Chimera knows that you want to create an instance of Example1 class and call this instance 'example'.

You can also pass configuration parameters right on the Location given in the command line. Use:

```
$ chimera -I $HOME -i /Example1/example?param1="Now for something different"
```

A few points need explanation on `Example1`:

1. `__config__` is a class attribute (class field in some circles) where you should pass a Python dictionary with any parameter you like to add to your object. Chimera uses the value you pass in as default value and also uses the type of it to do some type checking for you. Look at src/chimera/core/config.py for valid types.

2. `__start__()` method. This method is from `ILifeCycle` interface, `ChimeraObject` implementation just does nothing, here we use it to call a specific method on object initialization. `Manager` first call `__init__()` to create an instance, configure this instance passing any parameter you gave on command line, then call `__start__()` and when system is shutting down call `__stop__()`.

3. `log`. `ChimeraObject` implementation give a *log* attribute (instance field, in other circles) to every class, you can use this to log messages to default Chimera log system. It's a normal Python's `logging`

logger, so consult logging for more information.

4. `self["param1"]`. You define your object parameters using `__config__` dict, but to access the actual value, you use the current object (self) as dictionary to access values from it. Thus, `self["param1"]` treat *self* as a dict and get key *param1* from it. For most purposes, self is a dict and normal dict. You can also set things, with normal `self["param1"] = "value1"`.

When you use **chimera** script, a `Manager` is created for you, but you can do it by yourself to learn how things work in Chimera. The following example is based on `server.py`.

```
from chimera.core.manager import Manager

manager = Manager(host='localhost', port=8000)
manager.addLocation("/Example1/example", start=True)

manager.wait()
```

Suppose you save it to `server.py` in the same directory where you put `example1.py` (this is a not a restriction, just to make things easier).

```
$ python server.py
```

You'll see exactly the same as running **chimera**.

But, as said in the first paragraph of this document Chimera is client/server, `server.py` shows how to create a server, let's see how to use it in a client.

```
from chimera.core.manager import Manager

manager = Manager()

example = manager.getProxy("localhost:8000/Example1/example")
example.doSomething("client argument")
```

Save it to `client.py`. First run, `server.py` as explained above and then run `client.py`.

```
$ python client.py
```

You will see something like this:

```
[date] WARNING chimera.example1 (example1) example1.py:15 Hi, I'm doing something.
[date] WARNING chimera.example1 (example1) example1.py:16 My arg=test argument
[date] WARNING chimera.example1 (example1) example1.py:17 My param1=a string parameter
[date] WARNING chimera.example1 (example1) example1.py:15 Hi, I'm doing something.
[date] WARNING chimera.example1 (example1) example1.py:16 My arg=client argument
[date] WARNING chimera.example1 (example1) example1.py:17 My param1=a string parameter
```

The first three lines are from `__start__()` calling `doSomething()`, and later three from our client calling it again.

In `client.py`, you see we create a normal `Manager`, just like in server.py, but we only use this `Manager` to get access to `Example1` running on other `Manager` (localhost:8000).

`Manager.getProxy()` returns a `Proxy` object for the specifies Location. For all purposes this `Proxy` class acts like the original object, so you can call any method just like you would with the original object.

1.6.3 Plugin development

After trying to find inside the chimera core package, chimera tries to find controllers and instruments on packages with names starting with *chimera_*. This opens chimera to be customizable with third-party plugins of all kinds.

To facilitate the development of those plugins, we created a plugin template which can be forked and changed to born a new plugin. Take a look on our chimera-template plugin and, if there is any doubt, don't hesitate to contact us by opening an issue on github or sending an e-mail to our mailing list

1.7 Contact us

If you need help on setting chimera on your observatory, please contact us over our mailing list.

Bugs and feature requests can be sent over our GitHub page.

1.8 License

Chimera is Free/Open Software licensed by GPL v2 or later (at your choice).