CsPyController Programmers' Manual

Martin Tom Lichtman

2015 July 6

Contents

1 Introduction

The CsPyController software was written by MTL to run experiments and collect data for the AQuA(*Atomic Qubit Array*) project, however it was designed with extensibility in mind. This *Programmers' Manual* explains how to extend CsPyController by adding more **instruments** or **analyses**.

A separate *Users' Manual* explains the basic functionality of CsPyController. While successful use of CsPyController requires some knowledge of Python syntax, using the software as described in the *Users' Manual* requires little more than being able to write, for example, `a = 5`, or, `arange(10)`. This *Programmers' Manual* assumes familiarity with the use of CsPyController at least at the level of the *Users' Manual*. However, it also assumes at least a moderate degree of skill in object-oriented programming in Python.

Furthermore, while this manual explains the necessary features of a new **instrument** or **analysis**, it is left to the reader's creativity to invent new code that is powerful and useful.

2 GIT Version Control

As detailed in the *Users' Manual*, the CsPyController code is stored in, and can be cloned from, a GIT repository on the `hexagon`. You will need to be familiar at least with *branching, commiting*, *pushing* and *pulling* in GIT. A GIT primer is available on the Saffmanlab Wiki, and much more info is available on the web, particularly at `git-scm.com` and `stackexchange.com`. The main stable branch is called `master`. Always pull `master` before beginning your work to make sure your have the lastest version. You should never make edits in `master`. It is fast and resource-cheap to make new branches in GIT, so branch early and often. Make a new branch for every new idea that you try. Make frequent commits to your new branch, and push to the server to make sure your work is backed up.

The goal for all new branches should be to eventually merge them back into `master`, not to create a whole separate version of CsPyController for your project. New **instruments** and **analyses** that you program may eventually be useful to others, and they should be programmed with this in mind. Also consider that others may *not* want to use any particular **instrument** or **analysis**, and so they should always have `enable` flags that allow a particular piece of code to be ignored and consume no resources.

When your branch is mature enough to be both useful and stable, do not merge it into `master` yourself. Instead, make a *pull request* to whomever is in charge of CsPyController development (MTL until September 2015).

3 Tools

The author finds the *PyCharm* editor invaluable for programming in Python. A free *community edition* is available. It does a large amount of syntax and code flow checking for you on the fly, highlights potential errors, and it indexes the structure of your code so you can easily find the piece of code you are interested in.

4 Code Structure

4.1 Top-level Classes

The execution of CsPyController begins in `cs.py`, which does little more than create the GUI environment and assign an instance of `aqua.AQuA` to the GUI and vice-versa. `aqua.AQuA` is a subclass of `experiment.Experiment`. An instance of `experiment`, can be thought of as the master object that has complete knowledge of all the various components of the software apparatus. `experiment.Experiment` defines all the methods which control experiment flow and looping through experiments, iterations and measurements. `aqua.AQuA` catalogs the various instrument and analysis code that is available, and defines the evaluation and update order of those pieces. For example, `aqua.AQuA` has an instance of `andor.Andor` , an `Instrument` for the Andor camera. It also has an instance of `andor.AndorViewer`, which takes care of displaying new images from the Andor camera.

Once you program your **instrument** or **analysis** as a new class, you will usually need to add an instance of that class to `aqua.AQuA`. `Instruments` can be nested, and so in some cases you will not want to add your new **instrument** to `aqua.AQuA`, but instead to a lower class. For example, the `LabView.LabView` `TCPInstrument` handles communication for, and acts as a container for, several sub-instruments. So `LabView.LabView` has instances of `HSDIO.HSDIO`, `AnalogOutput.AnalogOutput`, and `AnalogInput.AnalogInput` (amongst others).

The appropriate places to add references to an `Instrument` are slightly different from an `Analysis`, and all the necessary references will be detailed in their respective sections of this document.

4.2 `Prop`

A `instrument_property.Prop` is the workhorse class that handles:

- evaluation with respect to the defined **constants**, **independent** and **dependent variables** namespace
- saving and loading settings

Most, if not all, classes that you define will inherit from `Prop`. For example, `Experiment`, `Instrument`, and `Analysis` are all subclasses of `Prop`. This allows their evaluation and save/load behavior to be standardized, called at the appropriate time. As long as you follow certain conventions, this allows you to program extensions to the code without worrying about these important processes.

Every `Prop` has at least the following instance variables:

- `name`: a name that gives it a unique path in the `property` tree (i.e. it does not have to be globally unique, just unique amongst its siblings)
- `description`: some helpful information about this particular `Prop` instance (such as why it was set to a particular value, or what units its value has)
- `experiment`: a reference to the top-level `Experiment` instance

Generally these three instance variables will be defined when the `Prop` is constructed, for example with a call to `super(myProp, Prop).__init__(name, experiment, description)`

Furthermore, every `Prop` also has at least the following instance variables:

- `properties`: a list of the instance variable names that should be evaluated (if they have such behavior) and saved. To add to the `properties` list, be sure to denote the variable names as strings, not as actual Python objects. Also, you will almost always want to append to the `properties`, instead of overwritting them, so that any `properties` from the parent class are preserved. For example:

```
self.properties += ['clockRate', 'units']
```

- `doNotSendToHardware`: a list of items that are also in `properties`, but that you do not wish to be processed by the `Prop.toHardware()` method, which creates XML code for transmitting to some TCP instrument server. Adding items to this list follows the same syntax as `properties.` `For example:`

  ```
  self.doNotSendToHardware = ['description']
  ```

When a `Prop` is evaluated, CsPyController iterates through the `properties` list and attempts to evaluate each item. The items in `properties` do not have to be instances of `Prop`. However, if you do include `Props` in `properties`, you can created nested trees of `Props`. Furthermore, when the settings are saved to HDF5 files, these nested trees are preserved in the HDF5 hierarchy. This is how many of the CsPyController `Instruments` organize their settings.

4.2.1 `EvalProp`

A `Prop` knows how to save/load its `properties`, and when a `Prop` is evaluated it knows how to go through its `properties` and try to evaluate them. However, it still does not know *how* to actually evaluate itself with respect to equations or functions and the like. For this, we have the `instrument_properties.EvalProp` class, and several of its subclasses `StrProp`, `IntProp`, `RangeProp`, `IntRangeProp`, `FloatRangeProp`, `FloatProp`, `BoolProp`, `EnumProp`, `Numpy1DProp` and `Numpy2DProp`.

In addition to all the workings of a `Prop`, each of these has a `function`, and a `value`. The `function` is a string which holds Python syntax code that will be evaluated in the namespace of the **constants**, **independent** and **dependent variables**. The evaluation is checked to make sure it results in the correct `type`, and in some cases within the correct range, and then is stored in `value`.

The different subclasses of `EvalProp` are generally what you will use for `Instrument` and `Analysis` settings when you want users to be able to use variables there. More often than not you might as well enable this behavior, as opposed to static settings, as some future user might want to scan a setting in a way you did not expect.

4.3 GUI

4.3.1 Enaml

The CsPyController uses the `enaml` package to create the GUI. For reference on *Enaml*, be sure to refer to the *Nucleic* documentation at `http://nucleic.github.io/enaml/docs/` or the source code at `https://github.com/nucleic/enaml`, and not the older documentation or code from *Enthought*. The GUI is defined using a heirarchical (i.e. nested) syntax in `cs_GUI.enaml`. The syntax for the *Enaml* file is mostly Python syntax, with several added operators (and some restrictions). In order to make the settings on your new **instrument** or **analysis** accessible to the user, you will have to first define the appropriate GUI widgets, and then link them to the backend instance variables that represent your **instrument** or **analysis**.

4.3.2 Make a new window

Usually you will create a new window to display your **instrument** settings or **analysis** results or graphs. To do this, first define the new `Window` widget.

4.3.2.1 Instrument window For example:

```
enamldef CameraWindow(Window):
    attr camera
    title = 'Groovy EMCCD Camera'
    Form:
```

```
            Label:
                text = 'enable'
            CheckBox:
                checked := camera.enable
            Label:
                text = 'scan mode'
            SpinBox:
                value := camera.scan_mode
                minimum = 1
                maximum = 3
        EvalProp:
            prop << camera.EM_gain
        EvalProp:
            prop << camera.cooling
        EvalProp:
            prop << camera.exposure_time
        PushButton:
            text = 'take a picture'
            clicked :: camera.take_one_picture()
```

In this example we see several new features. First the `enamldef` statement, which functions much like a `class` declaration, but signals the *Enaml* parser that this defines a new GUI object called `CameraWindow`. Here `CameraWindow` is defined as a subclass of `Window`. Merely defining `CameraWindow` does not actually create one, but we can create as many instances of it as we like, which will be explained below.

Next `attr camera` is how the instance variable `camera` must be declared. Here `camera` will store a reference to an instance of an `Instrument` which contains all the information and functions for controlling a camera.

The layout of the `Window` and its sub-widgets is controlled using a nested syntax. For example the `Window` contains a `Form` (an invisible container with two column layout), which in turn contains `Label` and `CheckBox`. These are base widgets from the `enaml` package. The default layout for *Enaml* widgets is usually adequate, but you may fine tune all the layout and behavior as described in the *Enaml* docs.

There are several assignment operators that are unique to *Enaml*. First we see `text = 'enable'` which is a simple one-time assignment of the string `'enable'` to the `text` field of the `Label`. Next we see `checked := camera.enable`, where the `:=` operator denotes a two-way synchronizaton. Any changes to `camera.enable` (a True/False boolean variable) will update the checked/unchecked state of the `CheckBox`. At the same time any time the user checks/unchecks the `CheckBox` causing its `checked` state to change, the value of `camera.enable` will change on the backend. This is one of the best features of *Enaml* that makes it easy to link up variables on the GUI and backend. `camera.enable` is an example of a setting that does not respond to equations (it is a `Bool`, not an `EvalProp`).

The `Form` also contains another `Label` and a `SpinBox`. The `SpinBox` has its `value` synced to the variable `camera.scan_mode`, which is another example of a setting that does not take equations (it is an `Int`, not an `IntProp`). The `minimimum` and `maximum` arguments define the available range of the `SpinBox`.

Then we see the `EvalProp` widget, which is a useful custom widget defined in `cs_GUI.enaml`, that links to an `instrument_property.EvalProp`, displays the `EvalProp.name`, gives a place to enter the `EvalProp.description` and `EvalProp.function`, and displays the evaluated `EvalProp.value`. This works with any kind of `EvalProp`, be it an `IntProp`, `StrProp`, or `FloatProp`, etc. Here we see how the CsPyController backend has made it easy for you to handle `EM_gain` (an `IntProp`), `cooling` (a `BoolProp`), and `exposure_time` (a `FloatProp`) all using the same code. The `function` field will highlight in red if it does not evaluate to the correct type, making it easy to find user errors. A `placeholder` in the `function` field shows the expected type or range if the field is left blank.

You may of course define your own custom widgets to handle your data structures in new ways.

Within the `EvalProp` widgets, we see the use of the subscription operator `<<`. This operator is a one-way subscription, so that whenever, for example, the `camera.EMGain` object changes identity (such as during a settings load), the GUI will update (but not vice-versa). The operator `>>` is also available which is a one-way broadcasting that will update the backend variable when the GUI updates, but not vice-versa.

Finally, we have clickable button defined using `PushButton.` We see the `::` operator, which does not pass a value, but instead defines an action to be taken. In this case, when the `clicked` state of the `PushButton` changes, the method `take_one_picture()` is called.

4.3.2.2 Analysis Window A `Window` for an `Analysis` is created in much the same way as for an `Instrument`. For the `Analysis` you will usually want to have more ways to display data and statistics. For example:

```
enamldef PictureViewer(Window):
    attr viewer
    title = 'Picture Viewer'
    MPLCanvas:
        figure << viewer.fig
    Label:
        text << viewer.text
```

In this example the attribute `viewer` would be linked to an instance of, for example an `analysis.AnalysisWithFigure`. The `MPLCanvas` is a widget which allows the display of any `matplotlib` figure. The GUI display is only updated whenever the identity of `viewer.fig` is changed as specified by the `<<` subscription operator. (A simple redraw is not enough, but these mechanics are handled for you if you use the `AnalysisWithFigure` class.) Finally the `Label` widget is used to display dynamic from `viewer.text` by using the `<<` subscription operator, unlike in the `CameraWindow` example above where the `Label text` is static.

4.3.3 Add the Window to the list

In order so that the user can call up your new `Window` by selecting it on the combo box in the `MainWindow` (the one that opens when you launch `cs.py`), it must be added to the `window_dictionary` used in `Main`. Toward the bottom of `cs_GUI.enaml` you will find the definition of `window_dictionary` as a Python `dict`. Add your `Window` to the list with the following syntax:

```
'Groovy Camera Setup': 'CameraWindow(camera = main.experiment.camera)',
```

or

```
'Groovy Camera Display': 'PictureViewer(analysis = main.experiment.picture_viewer)',
```

The first element is a string key that is used as the display text in the combo box. The second element is a string, which when evaluated is a call to the constructor for your new `Window` subclass. The constructor is passed values for all the `attr` attributes defined in the `Window`. In `main.experiment.camera`, first `main` refers to the `MainWindow`, which knows about `experiment` which is your instance of `experiments.Experiment`, which finally has an instance of an `Instrument` named `camera`. (We will cover creating this backend instance below.) Similarly, the `PictureViewer` instance is passed a reference to an instance of an `Analysis` named `picture_viewer` on the backend. This list is automatically sorted alphabetically, so position is not important. However, be sure that each line except the last ends with a comma.

5 atom

Use of `enaml` for the GUI requires that we use the `atom` package to support the variable-to-GUI synchronization and event observation. This offers both advantages and additional headaches. Any class whose variables we would like to sync with the GUI, must descend from the class `atom.api.Atom`. To achieve this, we make `Prop` a subclass of `Atom` so that every `EvalProp`, `Instrument` and `Analysis` already has this inheritance.

One disadvantage (although it provides a performance boost) of using an `atom.api.Atom` is that you cannot declare instance-wide variables on the fly, they must be declared at the top of the class definition. What this means is that you cannot state:

```
from atom.api import Atom
class MyClass(Atom):
    def myMethod(self):
        self.x = 5
```

Instead you must declare `x` using, for example:

```
from atom.api import Atom, Int
class MyClass(Atom):
    x = Int()
    def myMethod(self):
        self.x = 5
```

Here the type used is `Int`, which is not the basic Python `int` but instead is an `atom` class which implements error checking to make sure that only integers are assigned to `x`. `atom` also has classes for `Bool`, `Float`, `String` and many other types as well as customizable wrappers. It is required to use these `atom` types instead of basic Python types. If you would like to synchronize one of these backend variables with the GUI, then the declared type of the variable must match the type expected by the GUI widget.

There are often variables that you would prefer not to have to both to declare, perhaps because they will never be used in the GUI, or they may have some unique type that is not supported easily by `atom`. For these, use the `Member` type which is the most general that `atom` allows:

```
from atom.api import Atom, Member
class MyClass(Atom):
    x = Member()
    def myMethod(self):
        self.x = some_weird_type()
```

The synchronization tools of `atom` are activated automatically by using the `:=`, `<<`, or `>>` operators in the `.enaml` file. There are further ways to leverage atom to perform actions on variable changes, such as the `@observe` decorator. Used here in an example from `AnalogInput.py`:

```
from atom.api import observe, Str
class MyClass(Atom):
    list_of_what_to_plot = Str()
    @observe('list_of_what_to_plot')
    def reload(self, change):
        self.updateFigure()
```

In this example, whenever the string `list_of_what_to_plot` is changed, then the `updateFigure()` method is called.

Info on the `atom` package is available at `https://github.com/nucleic/atom`, however the most complete information on `atom` is actually available in the `enaml` examples.

6 Instrument

6.1 Create a new Instrument

The class `cs_instrument.Instrument` is the base class to use to describe a new **instrument**. First, create a new `.py` file to hold your class. At the top of the class, you will almost always want some fashion of the following import lines:

- Usually it is a good idea to set default divison to be floating point, not integer math (so that $1/2 = 0.5$ instead of $1/2 = 0$).

  ```
  from __future__ import division
  ```

- Don't use print statements, instead use `logger.info()`, `logger.debug()`, `logger.warning()`, `logger.error()`. These will handle time stamping and saving to the log.txt file.

  ```
  import logging logger = logging.getLogger(__name__)
  ```

- Your code should implement try/except error catching blocks, and give description errors sent to the `logger` commands. When the error is bad enough that the experiment execution should be paused, use `raise PauseError`.

  ```
  from cs_errors import PauseError
  ```

- You will probably need some `atom` types:

  ```
  from atom.api import Member, Int, Bool, Str, Float
  ```

- Numerical functions are very often useful:

  ```
  import numpy as np
  ```

- You will probably want some `EvalProp` types:

  ```
  from instrument_property import BoolProp, IntProp, FloatProp, StrProp
  ```

- Finally, you will need the CsPyController base class for an **instrument**:

  ```
  from cs_instruments import Instrument
  ```

Now define your new class. In this simple example we will create a new camera class that uses a `DLL` (dynamic link library) driver to send commands to the hardware. This is very hardware specific, and you might use some other means to communicate with the hardware. Use of the `TCP_Instrument` to communicate with a separate **instrument server** is shown later. The main points to absorb here are how the variables are set up to coordinate with the GUI frontend described above.

```
from ctypes import CDLL
import class GroovyCamera(Instrument):
    EM_gain = Member()
    cooling = Member()
    exposure_time = Member()
    scan_mode = Int(1)

    dll = Member()
```

```
current_picture = Member()

def __init___(self, name, experiment, description='A great new camera'):
    # call Instrument.__init__ to setup the more general features, such as enable
    super(self, GroovyCamera).__init__(name, experiment, description)

    # create instances for the Prop properties
    self.EM_gain = IntProp('EM_gain', experiment, 'the electron multiplier gain (0-255)', '0')
    self.cooling = BoolProp('cooling', experiment, 'whether or not to turn on the TEC', 'True')
    self.exposure_time = FloatProp('exposure_time', experiment, 'how long to open the shutter [

    # list all the properties that will be evaluated and saved
    properties += ['EM_gain', 'cooling', 'exposure_time', 'scan_mode']

def initialize(self):
    """initialize the DLL"""
    self.dll = CDLL("camera_driver.dll")
    super(self, GroovyCamera).initialize()

def take_one_picture(self):
    """Send a single shot command to the camera.
    Use a hardware command, which might be call to a DLL, for example
    This ficticious example returns the picture as an array, which is assigned to self.current_
    """
    if not self.isInitialized:
        self.initialize()
    if self.enable:
        self.current_picture = self.dll.take_picture_now()

def update(self):
    """Send the current settings to hardware."""
    self.dll.set_EM_gain(self.EM_gain.value)
    self.dll.set_cooling(self.cooling.value)
    self.dll.set_exposure_time(self.exposure_time.value)
    self.dll.scan_mode(self.scan_mode)

def start(self):
    """Tell the camera to wait for a trigger and then capture an image to its buffer."""
    self.dll.wait_for_trigger()
    self.isDone = True

def acquire_data(self):
    """Get the latest image from the buffer."""
    self.current_picture = self.dll.get_picture_from_buffer()

def writeResults(self, hdf5):
    """Write the previously obtained results to the experiment hdf5 file."""
    try:
        hdf5['groovy_camera/data'] = self.current_picture()
    except Exception as e:
        logger.error('in GroovyCamera.writeResults() while attempting to save camera data to hd
        raise PauseError
```

Let us go through the parts of this `Instrument`. First, we needed to declare all the instance variables, because `Instrument` is a `Prop` which is an `Atom`. The `EM_gain`, `cooling`, `exposure_time` and `scan_mode` variables all require this because they are synchronized with the GUI. The `dll` and `current_picture` variables don't have a purpose in being declared, but it is required to declare all instance variables within an `Atom`.

The `__init__` method is called when an instance of `GroovyCamera` is constructed, and takes in the `name`, a reference to the `Experiment`, and an optional `description` with a default description of `'A great new camera'`. We then immediately pass on most of this information to the parent class using a `super` command, so that `Instrument` can handle this info with the default behavior. Next we create instances for each `Prop` that this class contains. `IntProp`, `BoolProp` and `FloatProp` each take the same arguments of `(name, experiment, description, initial_function_string)` with the only difference being in the type that the function string will resolve to. It is necessary that the `name` parameter be a string that matches the actual variable name, and so `self.EM_gain` is given a `name` of `'EM_gain'`. The reference to `experiment` will be passed in when we instantiate this class, which will be shown later. The `description` should contain useful text specific to this variable, such as the units, or why a particular value was chosen. Finally, the `initial_function_string` can contain variables and equations, but must evaluate to the correct type. Note that this is a string, and so we write `'50.0'` and not `50.0`. If a `StrProp` were used, the string must evaluate to a string, so you could write '''hi''' or `'str(5)'` for example. Finally we must indicate that all these variables should be evaluated and saved, by adding them to the `properties` string. Note that once again this is a list of strings, and so we write `['EM_gain', 'cooling', 'exposure_time', 'scan_mode']`, and not `[EM_gain, cooling, exposure_time, scan_mode]`. Also note that we say `properties +=` and not `=`, because we do not want to lose any properties from the list that were assigned in `Instruments.__init__()`, which in this case includes the `enable` variable.

Note how we used the `enable` variable in the GUI example, and yet it is not shown in the code above (except in `take_one_picture()`). That is because `enable` is set up in the parent class and inherited without modification here. It is necessary to specifically check the `enable` variable in `take_one_picture()` because that is a custom method for this example. However, we do not check `enable` in `initialize()`, `update()`, `start()`, `acquire_data()` or `writeResults()` because CsPyController takes care of checking that for all `Instruments` before these methods are called in `Experiment`.

The `initialize()` method is called for all `Instruments` before they `update`, but only once (or as long as `self.isInitialized == False`). This is a good place to do initial one-time setup of the instrument. In this example we use this method to setup the DLL. We end with a call to `Instrument.initialize()` via `super`, which in this case will just set `self.isInitialized = True` for us, so that `initialize` will not be called again.

The `take_one_picture()` method is something that we set up in this example to be called by a `PushButton` on the GUI. This method is therefore executed in the GUI thread. If the DLL call is very slow, it would be necessary to have this method spawn a different thread which would then make the DLL call, so as to not cause the GUI to hang. We check `isInitialized` before proceeding with this method because `initialize()` may not have been called yet, if this button is pressed before the first experiment is run.

The `update()` method is called at the beginning of every **iteration**, after everything has been evaluated with the newly iterated variables. The job of `update` is to send the updated settings to the hardware. This is very hardware specific, and in this example we do so with a series of calls to the DLL. Note how we pass `EM_gain.value`, `cooling.value`, and `exposure_time.value`, and not `EM_gain`, `cooling` and `exposure_time`, because we do not want to pass the whole `EvalProp` instance, just the relevant evaluated value. For `scan_mode` we can just pass `scan_mode` because it is a primitive type, not an `EvalProp`.

The `start()` method is called at the beginning of every **measurement**. If this instrument's timing is to be triggered by some other piece of hardware, like for example an HSDIO digital output channel, then `start()` should just set the instrument up to wait for the trigger, which is what we have done here. If this instrument will be internally timed, then just go ahead and tell it to proceed with a **measurement**. The **measurement** will not end until all the instruments have `self.isDone == True` (or if the experiment timeout is reached). You can delay setting `isDone` to `True` until you have confirmation that the instrument has fired, for example by creating a watchdog thread which keeps checking the camera status or buffer and updates `isDone` only once that status changes. Or you can take the easier route that we use here, and simply trust that the camera will do its job if it gets the trigger, and that the HSDIO will not report `isDone` until it finishes its sequence and all the output triggers. So here we just set `self.isDone = True` right away.

Next, `acquire_data()` is called after all the `Instruments` reach the `isDone` state. This may not be necessary for all instruments, if for example the data was returned right away in `start()`. This method should store the data in an instance variable, where it can be used directly or accessed later for saving to the results file.

Finally, we have the `writeResults()` method. This method should save any data to the HDF5 file. A reference to the HDF5 node within this particular measurement (i.e. `f[/iterations/#/measurements/#/data]`) is passed in as `hdf5`. The reason that this method exists separately from `acquire_data` is that in some cases you may need to get back data from other instruments before deciding exactly how to process and save the data from this instrument. This separation is not always necessary, and so you could do the work of both `acquire_data` and `writeResults` in just `writeResults` and avoid having to create the `current_picture` temporary storage variable.

6.2 Create a new `TCP_Instrument`

6.3 Instantiate your `Instrument`

7 Analysis

7.1 Create a new `Analysis`

7.2 Instantiate your `Analysis`

8 Afterword

This guide is intended to explain the minimum necessary structure for adding an **instrument** or **analysis**. CsPyController is however a complicated package, with at the time of this writing 35519 lines of Python code, a great deal of auxiliary code in LabView, C, C++ and C#, totaling 821 MB for the repository, and 764 GIT commits on `master`. The ultimate way to understand the details of implementation, and to get ideas for how complicated structures have been implemented, is to look at the source code. A great deal of effort, to the best of the author's ability and time, has been put into making the source code well commented. Your contributions to making this code even better will be greatly appreciated.