Rt Retrieval Framework User's Guide Documentation

Release

JPL

January 12, 2016

TABLE OF CONTENTS

This software (NASA NTR-49044) retrieves a set of atmospheric/surface/instrument parameters from a simultaneous fit to spectra from multiple absorption bands. The software uses an iterative, non-linear retrieval technique (optimal estimation). After the retrieval process has converged, the software performs an error analysis. The products of the software include all quantities needed to understand the information content of the measurement, its uncertainty, and its dependence on interfering atmospheric properties.

The software provides a flexible, efficient, and accurate tool to retrieve the atmospheric composition from near-infrared spectra. Its unique features are:

- Spectra from ground-based or space-based measurement with arbitrary observation geometry can be analyzed.
- The retrieved parameters can be chosen from a large set of atmospheric (e.g., volume mixing ratio of gases or aerosol optical depth), surface (e.g., Lambertian reflection), and instrument (e.g., spectral shift or instrument line shape parameters) parameters.
- The software uses an accurate, state-of-the-art, multiple-scattering radiative transfer code combined with an efficient polarization approximation to simulate measured spectra.
- The software enables fast and highly accurate simulations of broad spectral ranges by an optional parallelization of the frequency processing in the radiative transfer model.

CHAPTER

ONE

SETUP

The following sections describe how to obtain and initialize the tools and source code needed to develop and utilize the resources available for running the RT Retrieval Framework code. These sections describe a recommended directory structure that can be changed by the user. Operators/developers should feel free to reorganize where they deploy the tools. The organization has been designed to not necessitate any one strict directory organization.

1.1 Obtaining the Code

The public copy of the source code can be obtained from Github.

```
$ git clone https://github.com/nasa/RtRetrievalFramework.git rtr_framework
```

This is a clone of the development repository used at JPL. On a regular basis we push our changes to the public repository from JPL.

1.2 Enviroment Setup

The downloaded directory contains a bash script that will include into your shell environment paths and variables needed by various tools contained therein. These scripts should be sourced in your bash startup script in order for these tools to be configured and ready to use each time you log in.

Add the following to your .bashrc or wherever you place these things for yourself:

```
$ source /path/to/rtr_framework/setup_env.sh
```

This script is used when working on code developing. On the Compilation page it will be discussed how to use the software as an installed program.

1.3 Development Environment Organization

The development environment checked out in the preceding section has the following important sub-directories:

lib/	Where most source code lives
input/	Input files and Lua configuration
support/	Supporting utilities
tests/	End-to-End tests
unit_test_data/	Inputs and expected results used by unit tests

CHAPTER

TWO

SYSTEM CONFIGURAITON

2.1 Memory Requirements

Make sure your system has at least 2.5G of memory (including swap) available or else compilation may fail unexpectedly.

2.2 Python Requirements

The Python bindings to the framework and support Python programs require the following Python packages. Most likely your Linux distribution already has packages available using it's respective package management tool. Otherwise, use the following links for details on installing each package:

- Numpy/Scipy
- Matplotlib
- h5py
- PLY (Python Lex-Yacc)
- nosetests

Optionally you may want to have the following Python packages installed to use some of the less commonly used utilities:

- pyproj
- pyephem

2.3 Fortran Compilers

One of the following Fortran compilers is required for compiling:

- GNU Fortran – 4.5 or greater
- Intel Fortran Compiler – 11.1 or newer

2.4 Linux System Setup

2.4.1 Gentoo Deployment Steps

Install gdb if you would like to use that, it is not installed by default:

```
$ emerge -av gdb
```

If you do not already have Subversion installed:

```
$ emerge -av subversion
```

Install scripting languages and required supporting packages needed by the build process:

```
$ emerge -av ruby rubygems
```

The Ruby narray package is also required, but may only be available in the Gentoo testing branch. If you have not enabled testing for your architecture you would have to do the follow (if your architecture is amd64):

```
$ echo 'ACCEPT_KEYWORDS="~amd64"' > /etc/make.conf
$ emerge -av narray
```

We also require the installation of Python but by default Gentoo satisfies that through the eselect-python package. Note that emerging the dev-lang/python will install Python 3.x which is not yet used by our software.

Install Python packages for support utils:

```
$ emerge -av numpy
$ emerge -av h5py
$ emerge -av ply
```

Follow Setup steps.

Compile according to Compilation instructions for a Non JPL Build.

2.4.2 Fedora/CentOS Deployment Steps

Under CentOS you may need to update to a newer version of gcc/gfortran. CentoOS is conservative with GCC updates and keep newer versions in separate packages:

```
$ yum install gcc44 gcc44-gfortran gcc44-c++
```

Fedora has the latest GCC in the default packages:

```
$ yum install gcc gcc-gfortran gcc-c++
```

A Fedora minimal system will also need the following:

```
$ yum install make automake patch zlib-devel bzip2-devel
```

If you do not already have Subversion installed:

```
$ yum install subversion
```

Install scripting languages needed by build process:

```
$ yum install python ruby
```

Install ruby development packages needed for using gem to install ruby packages:

```
$ yum install ruby-devel
```

For Fedora install the rubygems package:

```
$ yum install rubygems
```

For CentOS (also possibly Redhat) install rubygems from source. Get the latest package from the RubyGems.org website then install:

```
$ tar zfvx rubygems-X.X.X.tgz
$ cd rubygems-X.X.X
$ ruby setup.rb
```

Note that the latest version of Ruby Gems might not work with CentOS's Ruby version. You may need to try earlier version.

For CentOS also install:

```
$ yum install ruby-rdoc
```

Use rubygems to install narray:

```
$ gem install narray
```

Follow Setup steps.

Compile according to Compilation instructions for a Non JPL Build.

2.4.3 Ubuntu/Debian Deployment Steps

Install automake if you plan on developing the code and need to update the Makefile after adding files:

```
$ apt-get install automake
```

Most other distributions provide gfortran along with gcc, but Ubuntu keeps it in separate packages. Install gfortran if not already installed:

```
$ apt-get install gfortran
```

Install gdb if you would like to use that, it is not installed by default:

```
$ apt-get install gdb
```

If you do not already have Subversion installed:

```
$ apt-get install subversion
```

Install scripting languages and required supporting packages needed by build process:

```
$ apt-get install python ruby rubygems libnarray-ruby
```

Install system library headers that are not installed by default:

```
$ apt-get install libz-dev libbz2-dev
```

Install Python packages needed for support and operation tools:

```
$ apt-get install python-ply python-h5py
```

Follow Setup steps.

Compile according to Compilation instructions for a Non JPL Build.

2.4.4 OpenSuSE Deployment Steps

Install development packages using development patterns:

```
$ zypper install -t pattern devel_C_C++ devel_basis devel_ruby
```

Install gfortran if not already installed:

```
$ zypper install gcc-fortran
```

If you do not already have Subversion installed:

```
$ zypper install subversion
```

Install Ruby support packages:

```
$ gem install narray
```

Install system library headers that are not installed by default:

```
$ zypper install zlib-devel libbz2-devel
```

Follow Setup steps.

Compile according to Compilation instructions for a Non JPL Build.

2.5 OS X System Setup

The following instructions are intended for those attempting to deploy on a Mac OS X system.

2.5.1 Select MacPort or Fink

Additional tools are needed on the Mac to build. There are two different projects that supply these tools, Macport and Fink. The two projects are similar, it is a matter of preference which one you select. If you don't otherwise care, then use MacPort, it is what was used during development.

2.5.2 MacPort

If you select MacPort, then follow the directions in this section.

Install MacPort

Look at http://www.macports.org/install.php for directions on installing macport.

Install Packages

Install packages using port:

```
$ sudo port install gcc44 python27 py27-ipython hdf5-18 boost gsl
```

Configuration

The configuration is slightly tricky, if you want to have python working. The issue is that you need to use the same version of gcc and g+ used to create python, while at the same time using a different version of gfortran. The configuration used in development is:

```
$ ../Level2/configure \
   FCLIBS="/opt/local/lib/gcc44/gcc/x86_64-apple-darwin10/4.4.5/libgfortranbegin.a /opt/local/lib/gcc
   FC=/opt/local/bin/gfortran-mp-4.4 F77=/opt/local/bin/gfortran-mp-4.4 \
   --with-python-swig --with-lidort=build --with-cppad=build --with-blitz=build
```

Note the "FCLIBS" - this is required. This works around splitting between different versions for g+ and gfortran. Without this, the python library will use the libstdc+ coming from g+-4.4, not the system one. This results in fairly obscure memory errors when std::string from one version of g is passed to a library expecting a different version.

Note that the strict requirement on the compiler is only needed for building the python wrappers. If you don't care about the wrappers, you can leave off the FCLIBS and use the same version for gfortran and for g+ and gcc. For nonpython development, the configuration we used is:

```
$ ../Level2/configure THIRDPARTY=build FC=/opt/local/bin/gfortran-mp-4.4 CXX=/opt/local/bin/g++-mp-4.
    CC=/opt/local/bin/gcc-mp-4.4 F77=/opt/local/bin/gfortran-mp-4.4 --enable-debug --enable-maintaine
```

In this case, we build all the thirdparty libraries rather than using the system versions, this is needed so that a consistent compiler is used for things link HDF5 and BOOST.

2.5.3 Fink

If you select Fink, then follow the directions in this section.

Upgrade Xcode

Upgrade to Xcode 3.1.x for 10.5, 3.2.x for 10.6 from Apple Developers Connection. Go to the Developer Connection member site then select downloads to find the desired Xcode version.

Xcode needs to be upgraded so that Fink dependencies can be meet otherwise an error like this will show up: "xcode (>= 3.1.2)" for package "gcc44-4.4.1-1000"

Install Fink

Download Fink and follow instructions: http://www.finkproject.org/download/index.php?phpLang=en

Enable Fink Unstable

Enable fink unstable packages according to these instructions: http://www.finkproject.org/faq/usage-fink.php?phpLang=en#unstable

Install Fink Packages

Install free Fortran compilers:

```
$ fink install g77 g95
```

Install Python packages:

```
$ fink install python26 numpy-py27 h5py-py27 matplotlib-py27 scipy-py27 ipython-py27
```

Since we are using the Fink unstable repository everything comes from source and the compilation which includes many dependencies can takes a fair amount of time.

As a consequence of the dependencies of the above packages the latest GCC is installed and hence gfortran will also be available.

CHAPTER

THREE

COMPILATION

The RT Retrieval Framework uses the standard autotools used by many open source software. While you can build the software in the same directory that you have the source, it is suggested that you use a separate build directory. This allows you to do different builds from the same source, e.g. an optimized and debug version.

3.1 Building

There is some third party software included with the source code that need compiling along with our code. To build the framework's code along with the third party software, execute the following:

```
$ mkdir build_optimized
$ cd build_optimized
$ /your/path/to/rtr_framework/configure THIRDPARTY=build
$ make all
```

In the above replace `/your/path/to/rtr_framework` with the location where you downloaded the software during the Setup instructions.

The `configure` command creates the Makefile to use. You only need to run this when you are creating a new build directory, after that you can just rerun `make all` to get any software updates. Configure will check a bunch of things on the system, and in the end print a report something like:

```
Level 2 Full Physics is now configured

 Installation directory:       /your/path/to/build_optimized/install
 Build debug version:          no
 Fortran compiler type:        ifort
 Fortran compiler:             /opt/local/depot/intel/11.1/064/bin/intel64/ifort -g -xSSE2 -O3 -Difor
 C compiler:                   gcc -g -O2
 LD Flags:                      -R /opt/local/depot/intel/11.1/064/lib/intel64:/opt/local/depot/intel

 HDF5 support:                 yes
 Build own HDF5 library:       yes
 Build own LIDORT library:     yes
 Install documentation:        no
```

If you are building from within your source checkout you can just use the following script included with the source:

```
$ ./nonjpl_build.script
```

In order to run all of the unit tests, you'll also need to have a copy of the ABSCO (Absoprtion Coefficient) tables. These files must be obtained separately from JPL.

For serious development it is recommended to compile the third party software once and then refer to it in subsequent builds:

```
$ mkdir build_third_party
$ cd build_third_party
$ /your/path/to/rtr_framework/configure THIRDPARTY=build --prefix=/install/path/for/third_party
$ make thirdparty
```

Now subsequent builds can use this directory instead of recompiling the third party packages:

```
$ /your/path/to/rtr_framework/configure THIRDPARTY=/install/path/for/third_party
$ make all
```

3.2 Build Results

The executable is `l2_fp`, and will be placed in the top of the build directory.

Another useful target is `install`, which will build `l2_fp` and copy it and all associated libraries to a separate install directory. The default directory used is `./install` unless you specify an alternative to `configure`. This is primarily of use for production to put the executable files in a separate location. For a developer, you generally don't need to do this step.

3.3 Using Other Compilers

The configure script will search for one of the supported compilers – ifort, f90, f95, gfortran, g95, pathf90, pgf90 - in that order.

Note that we only test with the Intel compiler "ifort" and "gfortran". We require a fairly new version of ifort (>= 11.1) and gfortran (>= 4.5).

If you want to use a different compiler than what configure automatically selects, you can specify it on the configure line as `FC=<compiler>`, e.g., `FC=f90`.

The thirdparty version of HDF-5 can only be built using ifort or gfortran. Other compilers will likely build if you don't try to build hdf5 (i.e., you don't have `--with-hdf5=build`).

3.4 Build Options

The third party libraries can be selectively compiled by specifying configure options such as `--with-hdf5=build`, or `--with-lidort=build`. This will build your own version of these libraries. This is particularly useful if you are actually modifying, for instance the LIDORT code (e.g., a bug fix), but still want to refer to a prebuilt directory of third party software. The full list of third party libraries can be seen by running configure with the `--help` option.

There are a few other options that can be passed to configure:

Option	Description
–help	Print out all the options for configure
–enable-debug	Build a version of l2_fp for debugging, rather than an optimized version
–enable-maintainer-mode	Automatically update configure, Makefile.in. Useful if you are editing the various ".am" files
–with-absco=<dir>	Specify location of ABSCO data used by unit tests
–prefix=<dir>	Specify directory to install to
FC=<compiler>	Specify a compiler to use
THIRDPARTY=build	Build a local copy of all the third party libraries
THIRDPARTY=<dir>	Search in <dir> in addition to the normal locations for third party libraries

3.5 Debugging

It has been discovered that gfortran works better with valgrind and debugging (although our "official" builds use ifort). The default gfortran on many systems is often too old, we use Fortran 2003 features that were not available until GCC version 4.5. To use a different gfortran version, you can use the following configure command:

```
$ /path/to/level_2/configure FC=gfortran-4.5 CC=gcc-4.5 CXX=g++-4.5 --enable-debug <other config opti
```

3.6 Developer Information

For developers who need to understand autotools in more detail, an nice introduction is found at Autotools: a practitioner's guide to Autoconf, Automake and Libtool, and detailed reference can be found at Autobook.

CHAPTER

FOUR

TESTING

4.1 Unit Testing

There are two kinds of tests available. The first is unit testing. These test individual pieces of the system. The focus of this testing is on the software itself: did it build correctly, are the individual classes calculating what we expect them to.

The unit tests are fully automated. You can run them with the following command from a build directory:

```
$ make check
```

This runs through all the unit tests that don't take a long time to run, and in the end says if the tests are successful or not (i.e., there is no analysis on your part, the test either succeeds or it doesn't).

When running the unit tests, it can be useful to work in a debug build. This means the configuration used the `--enable-debug` flag. This runs slower, but it adds additional checks such as range checking in both the C+ and Fortran arrays.

The target `fast_check` works just like `check` except it does not build any of the executables. You will not catch any problems introduced into the build of the top level executables. This test target is faster and used in instances where one is developing a class and only running unit tests. The target `check` should still be run after the major work on a class has been completed.

A longer, more complete set of tests can be run using the command:

```
$ make long_check
```

This runs all the tests that `make check` does, plus additional tests that take longer to run. For an optimized build (without the `--enable-debug` flag in the configuration), this takes a few minutes to run. A debug version takes longer to run, but still runs in under an hour.

Just like with `make check`, the long checks print out if the tests are successful or not.

Support exists for specifying specific unit tests to run. For instance so you can do:

```
$ make fast_check run_test=lidort_driver/*
```

The command above will run just the lsi_rt unit tests. This feature passes the string through to the `--run_test` argument of the Boost unit tests, so consult the Boost documentation to see what can be specified.

This method does not catch breakage to any other classes, running the full suite should still be done occasionally to make sure everything is working.

4.2 Unit Testing Variables

The ABSCO tables are needed for certain unit tests. Once you have a copy of the ABSCO table files you can specify the location of these files when doing the initial configuration by specifying `--with-absco=<dir>`. You can also override the default on the make command line used for testing, for example:

```
$ make fast_check run_test=lidort_driver/* abscodir=/path/to/my/copy/absco
```

4.3 End-to-End Test

A build target `run_tests` exists which runs l2_fp for several tests (unless it hasn't changed since the last run), and then compares the results against expected results, printing out the differences. Individual tests are run by specifying the build name with the suffix `_run`. The tests names can be seen by looking at the subdirectories under the `tests/` directory of the source code.

For example, to run all tests, in your build directory run:

```
$ make run_tests
```

To test only a OCO-2 test case use the following target:

```
$ make oco2_sounding_1_run
```

Or, to test only a GOSAT case use the following:

```
$ make tccon_sounding_1_run
```

CHAPTER

FIVE

LUA CONFIGURATION

Lua is a lightweight, embeddable scripting language. It was chosen to manage the configuration over a declarative approach due to its flexibility. The code itself was written in a manner such that the C++ portions are components that have minimal dependency on each other and adhere to well defined interfaces. The Lua configuration selects which components are used and makes them aware of each other. The configuration files themselves are written in layers where more specific configurations extend the general. At the bottom level are configurations that implement instrument specific arrangements of components. These files specify which files are involved in the processing as well as algorithmic choices. At this level the Lua code looks like declarative keyword/value configuration files. This approach combines flexibility with simplicity for end users.

The configuration loads input data, building a priori and initial guess vectors as it processes components. Components are connected to each other and passed their input in a hierarchal manner. Dependent components are loaded first so they are initialized and can be passed to components that need them. The top most level is the forward model component. It is created last and when ready called to start iterative execution as described in previous sections.

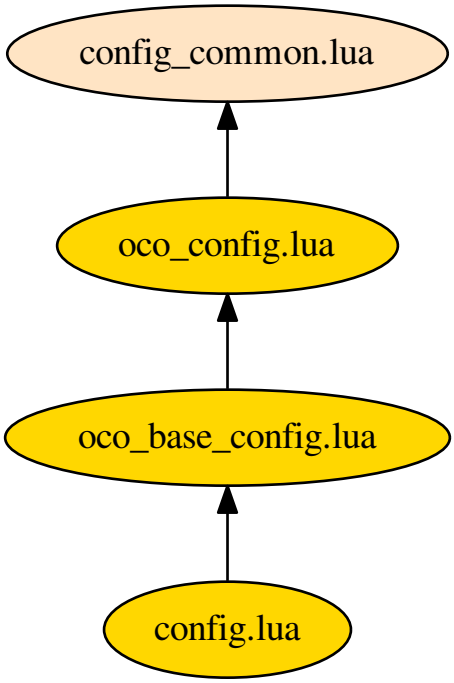
5.1 Indexing

It is worth noting that Lua uses 1-based indexing unlike the C++ code that is wrapping. Therefore care must be made when interacting with objects. If the object is pure Lua then use 1-based indexing. If you are dealing with, say a vector from the C++ world then it will be 0-based.

5.2 Organization

There are many separate Lua files involved in a single execution of L2 FP. These files have been organized such that the files with more general functionality are extended by files which use more specific implementation details. One of the configuration Lua files typically extends another by using the Lua `require` statement and Lua's table inheritance.

The graph below shows the organizational hierarchy for the OCO configuration:



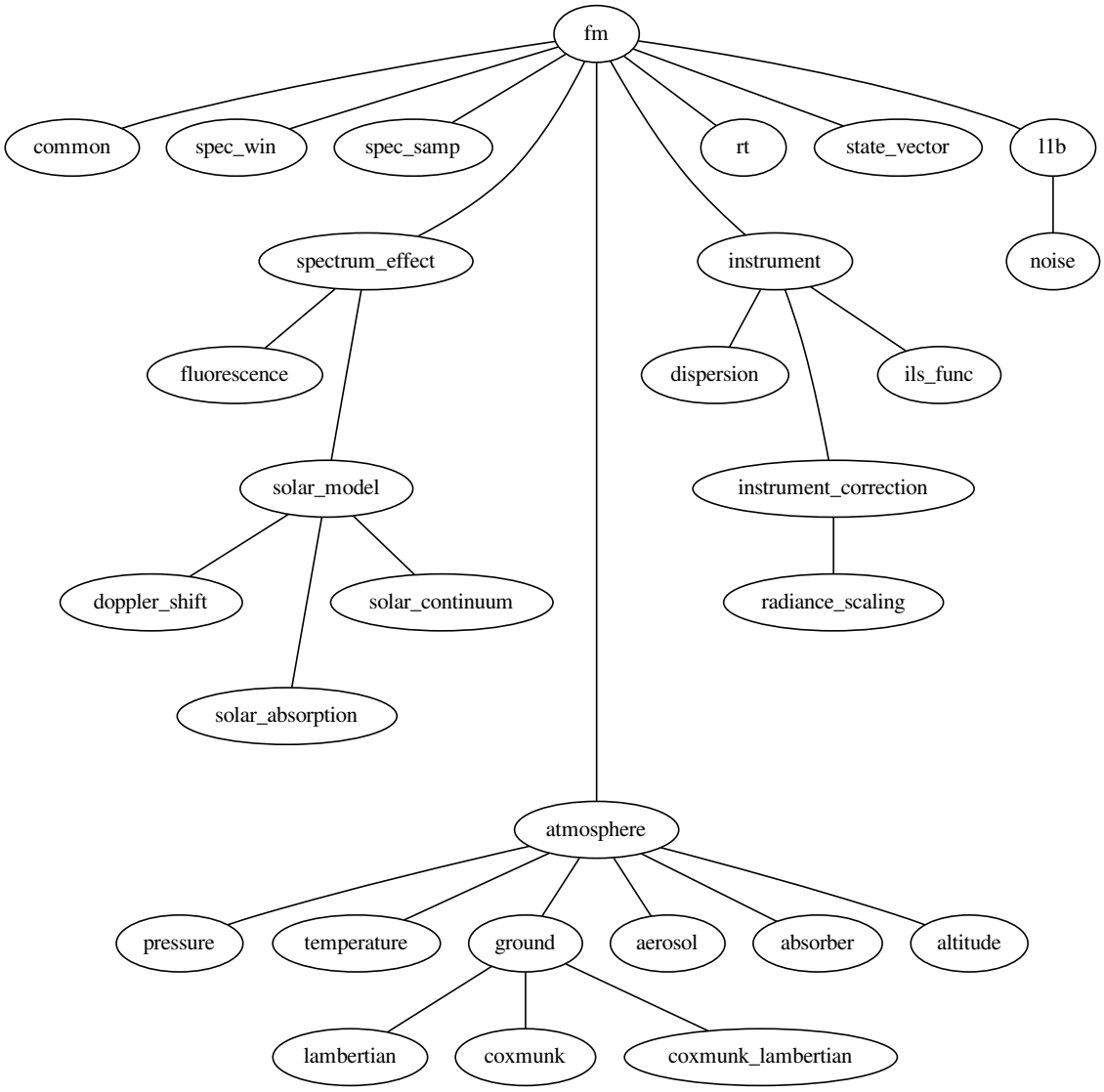
config_common.lua	Common routines for any instrument type
oco_config.lua	Common routines for OCO configuration
oco_base_config.lua	Base configuration of how common components are connected
config.lua	Empty config meant to be copied by users for modification

5.3 Config Common

As the table above states, `config_common.lua` is collection of common routines that are not instrument specific. It defines `ConfigCommon`, a table object which contains the configuration and also serves as a name space. The most important method is `do_config`. It is called from the lowest level configuration file to begin the process of instantiating objects.

L2 FP framework objects are created through Lua `Creator` classes. These classes define how a particular functionality is instantiated. They also provide an initial guess value, objects to be added to the state vector and objects to be registered into the output file. The `Creator` class also handles calling any other `Creator` objects that it depends on.

The hierarchy of `Creator` objects forms a tree which at the top is the one `Creator` that `do_config` knows about directly, the `fm` creator. All other `Creator` objects are dependencies of this one or other `Creator` objects. The graph below shows the organization of the `Creator` objects in the current OCO configuration.



5.4 Base Config

`oco_config.lua` extends the `ConfigCommon` object with `OcoConfig` by adding additional `Creator` classes and functions. `oco_base_config.lua` extends `OcoConfig` and declares which `Creator` classes are used. It defines the `fm` table with nested tables for dependent `Creator` classes. Each `Creator` block contains a required `creator` attributes defining the class to use. Additional attributes can be declared in the block for use by the `Creator` object.

An abbreviated portion of the table is duplicated below:

```
fm = {
   creator = ConfigCommon.oco_forward_model,
   instrument = {
```

```
      creator = ConfigCommon.ils_instrument,
      ils_half_width = { DoubleWithUnit(4.09e-04, "um"),
                         DoubleWithUnit(1.08e-03, "um"),
                         DoubleWithUnit(1.40e-03, "um") },
      dispersion = {
         creator = ConfigCommon.dispersion_polynomial,
         apriori = ConfigCommon.l1b_spectral_coefficient_i,
         covariance = OcoConfig.dispersion_covariance_i("Instrument/Dispersion"),
         number_pixel = ConfigCommon.hdf_read_int_1d("Instrument/Dispersion/number_pixel"),
         retrieved = true,
         is_one_based = true,
      },
      ils_func = {
         creator = OcoConfig.ils_table_l1b,
      },
   },
   atmosphere = {
      creator = ConfigCommon.atmosphere_oco,
      constants = {
         creator = ConfigCommon.default_constant,
      },
      pressure = {
         apriori = ConfigCommon.ecmwf_pressure,
         covariance = ConfigCommon.hdf_covariance("Surface_Pressure"),
         a = ConfigCommon.hdf_read_double_1d("Pressure/Pressure_sigma_a"),
         b = ConfigCommon.hdf_read_double_1d("Pressure/Pressure_sigma_b"),
         creator = ConfigCommon.pressure_sigma,
      },
      temperature = {
         apriori = ConfigCommon.hdf_apriori("Temperature/Offset"),
         covariance = ConfigCommon.hdf_covariance("Temperature/Offset"),
         creator = ConfigCommon.temperature_ecmwf,
      },
      absorber = {
         creator = ConfigCommon.absorber_creator,
         gases = {"CO2", "H2O", "O2"},
         CO2 = {
            apriori = ConfigCommon.tccon_co2_apriori_ecmwf,
            covariance = ConfigCommon.hdf_covariance("Gas/CO2"),
            absco = "v4.2.0_unscaled/co2_v4.2.0_with_ctm.hdf",
                table_scale = {1.0, 1.0038, 0.9946},
            creator = ConfigCommon.vmr_level,
         },
         H2O = {
            scale_apriori = 1.0,
            scale_cov = 0.25,
            absco = "v4.2.0_unscaled/h2o_v4.2.0.hdf",
            creator = ConfigCommon.vmr_ecmwf,
         },
         O2 = {
            apriori = ConfigCommon.hdf_read_double_1d("Gas/O2/average_mole_fraction"),
            absco = "v4.2.0_unscaled/o2_v4.2.0_drouin.hdf",
                table_scale = 1.0125,
            creator = ConfigCommon.vmr_level_constant_well_mixed,
         },
      },
   },
},
```

5.5 config.lua

The `config.lua` file for any given instrument is where the `do_config` method is called. It is intentionally kept short, with all default behavior kept in `oco_base_config.lua` and higher.

```
require "oco_base_config"

config = OcoBaseConfig:new()

config:do_config()
```

For configuration customization, it is standard practice to copy this file and place any changes after the `config` object is created and before the call to `do_config`. Most changes will take the form of modifying values from the *fm*' table from `oco_base_config.lua`.

5.6 Examples

The following snippets would all go in a modified `config.lua` before the `do_config` call. For instance:

```
require "oco_base_config"

config = OcoBaseConfig:new()

config.fm.atmosphere.absorber.H2O.retrieved = false

config:do_config()
```

For the remainder of the examples, we will just show the snippet of code that is added to the config file.

5.6.1 Toggle Retrieval

Most creators for items that are not band dependent have the `retrieved` attribute for controlling if the value appears in the state vector or not. Here are various items that can be controlled this way:

```
config.fm.atmosphere.absorber.H2O.retrieved = true
config.fm.atmosphere.absorber.CO2.retrieved =  true
config.fm.atmosphere.pressure.retrieved = true
config.fm.atmosphere.temperature.retrieved = true
config.fm.instrument.dispersion.retrieved = true
```

5.6.2 Rayleigh Aerosol Model

To ignore any aerosol particles in the retrieval and instead use only Rayleigh scattering, use the following::

```
config.fm.atmosphere.aerosol.creator = ConfigCommon.rayleigh_only
```

5.6.3 Toggle Retrieval Bands

By default the OCO retrieval uses all three spectrometer bands. One can pick and choose by using the strings: `ABO2`, `WCO2`, `SCO2` as in the following example of using the A-Band only::

```
require "single_band_support"
config.which_spectrometers = "ABO2"
init_single_band_support(config)
```

Or to use only the two CO2 bands you would use the following:

```
require "single_band_support"
config.which_spectrometers = "WCO2 SCO2"
init_single_band_support(config)
```

5.6.4 Ground Retrieval

Since the lambertian ground retrieval contains values for each band separately, the `retrieved` attribute will not work. Instead you can use the `retrieve_bands` attribute with a table containing a boolean for each band indicating whether or not it should be retrieved. For example to turn off the A-Band lambertian retrieval:

```
config.fm.atmosphere.ground.lambertian.retrieve_bands = { false, true, true }
```

Coxmunk retrievals do support the `retrieved` attribute and are controlled as follows:

```
config.fm.atmosphere.ground.coxmunk.retrieved = false
```

5.6.5 ABSCO Tables

The ABSCO tables on the OCO systems are installed in a central location. The `absco_path` value in the configuration specifies this location:

```
config.absco_path = "/groups/algorithm/l2_fp/absco"
```

There is generally no need to change this configuration value unless processing on a different machine. However even then it is best to use the `abscodir` environmental variable which if present overrides what is present in the config file.

A second ABSCO path, `absco_local_path` is a location where tables can be found on local disk from cluster machines. This is present to speed up processing, but the tables must be copied to each machine for this to be of any use:

```
config.absco_local_path = "/state/partition1/groups/algorithm/l2_fp/absco"
```

To change which tables are used for each gas the relative path and filename of each table under the `absco_dir` path is used::

```
config.fm.atmosphere.absorber.CO2.absco = "v4.2.0_unscaled/co2_v4.2.0_with_ctm.hdf"
config.fm.atmosphere.absorber.H2O.absco = "v4.2.0_unscaled/h2o_v4.2.0.hdf"
config.fm.atmosphere.absorber.O2.absco = "v4.2.0_unscaled/o2_v4.2.0_drouin.hdf"
```

The scaling applied to the tables for a gas can be changed with the `table_scale` attribute. If the value is an array then you can specify a different scaling per band. If a atomic value is used then the same value is used for all bands::

```
config.fm.atmosphere.absorber.O2.table_scale = 1.0125
config.fm.atmosphere.absorber.CO2.table_scale = { 1.0, 1.0038, 0.9946 }
```

Note in the above we just use a single value for O2 since this gas is only ever used in the A-Band. For CO2 we set the value in the A-Band to 1.0 as a placeholder since this gas is never used in that band.

5.6.6 ILS Half Width

The half width used in ILS convolution can be controlled band by band. The ILS expects a `DoubleWithUnit` object which is simply a class that wraps a value with its unit. The values are set in the `ils_half_width` array as shown in this example:

```
config.fm.instrument.ils_half_width[1] = DoubleWithUnit(4.09e-04, "um")
config.fm.instrument.ils_half_width[2] = DoubleWithUnit(1.08e-03, "um")
config.fm.instrument.ils_half_width[3] = DoubleWithUnit(1.40e-03, "um")
```

5.6.7 High Resolution Spectra

The following option will enable output of high resolution spectra. These values are the raw output of the radiative transfer before the instrument model and solar model are applied. Furthermore, they will be on the high resolution grid used in the radiative transfer. When enabled a new group `HighResSpectra` will appear in the output files.

```
config.write_high_res_spectra = true
```

CHAPTER

SIX

PYTHON WRAPPERS

We have wrappers that allow the C+ Full Physics library to be used in Python.

The exact same doxygen documentation is available in Python using the standard Python help system, in particular you can issue a command such as `help(AtmosphereOco)`.

Check the Python documentation generated using doxygen for information what classes are available through Python.

The wrappers are straight Python, but for interactive purposes use IPython, as well as the matplotlib and scipy libraries. If you aren't familiar when them, you can take a look at their getting started guide.

6.1 Building

The Python bindings are not built by default, you need to explicitly request that the bindings be compiled. This is done by adding an option to the configuration line:

```
$ /path/to/rtr_framework/configure --with-python-swig
```

You need to do a full install, not just a `make all`. This is because Python needs to see the libraries in a particular directory structure that gets created with the install. So you would build with:

```
$ make all && make install
```

6.2 Running

After building, you need to make sure that the Python path is set to point to the installed library. You can do this using the setup script installed in the install area:

```
$ source ./install/setup_fp_env.sh
```

6.3 Examples

First you will need to load the correct environmental variables then load a Python shell:

```
$ source ./install/setup_fp_env.sh
$ ipython --matplotlib auto
```

The easiest way to get started is by using a pre-existing Lua configuration:

```
In [1]: from full_physics import *
In [2]: from matplotlib.pylab import *
In [3]: conf = L2FpConfigurationLua('/path/to/preexisting_run/config.lua')
```

Add some information about getting the right enviromental variables made available.

6.3.1 Example 1 - Plotting Solar Spectrum

Get help on what conf is and can do:

```
In [3]: help(conf)
```

If you are using IPython, you can just use a ? after the object to get help:

```
In [3]: conf?
```

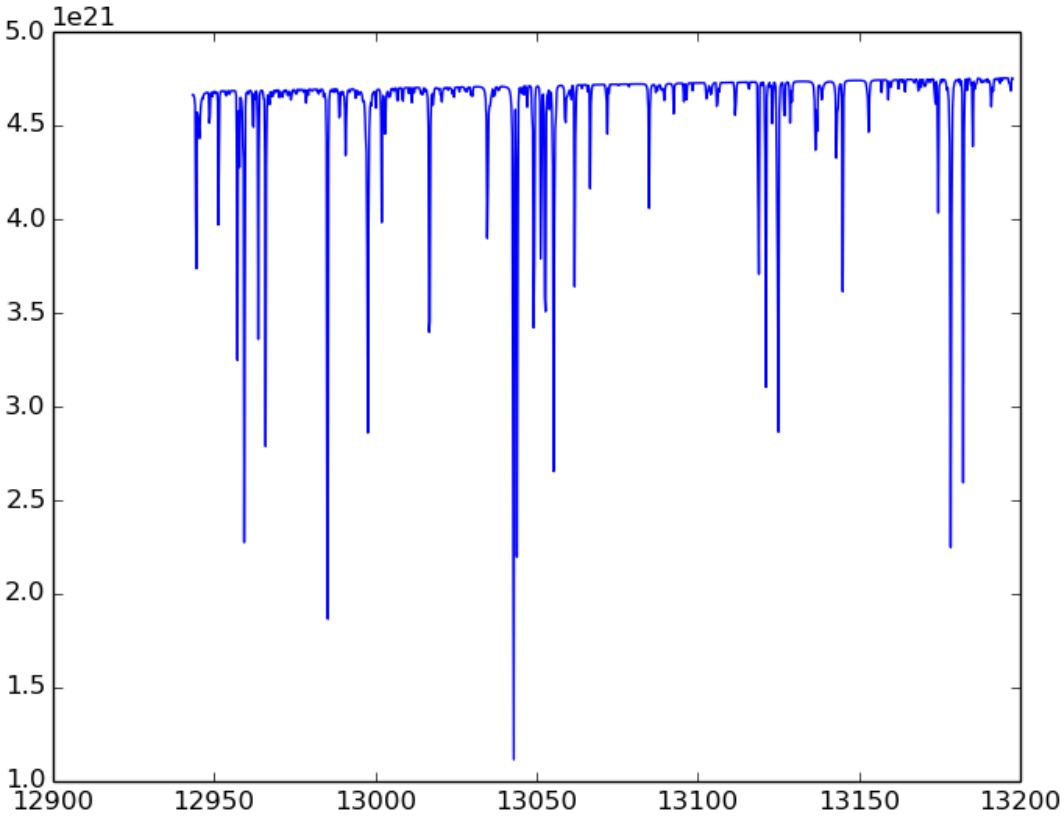
The solar model object is one of several spectrum effects, arbitrary operators on spectrum before instrument effects have been processed. It will probably be at index 0 of the spectrum effects list. Get help on what solar model is and can do:

```
In [4]: help(conf.forward_model.speceff_val(0, 0))
```

The second index above is the band index. Note that this is pretty much the same documentation as found in the Doxygen Documentation, or the Python Documentation but it can be convenient to do this in IPython without going to a browser.

We can plot the solar model with a couple of short commands as follows:

```
In [5]: band = 0
In [6]: spec_dom = conf.forward_model.instrument.pixel_spectral_domain(band)
In [7]: solar_spec = conf.forward_model.speceff_val(0, band).solar_spectrum(spec_dom)
In [8]: plot(solar_spec.wavenumber, solar_spec.value)
```



6.3.2 Example 2 - Forward model without Jacobian

Run forward model and get radiances only for all the bands:

```
In [9]: r = conf.forward_model.radiance_all(True)
```

The boolean argument to `radiance_all` tells the code to skip Jacobian calculations.

6.3.3 Example 3 - Forward model with Jacobian

Run forward model and get radiance and Jacobian (takes longer to run):

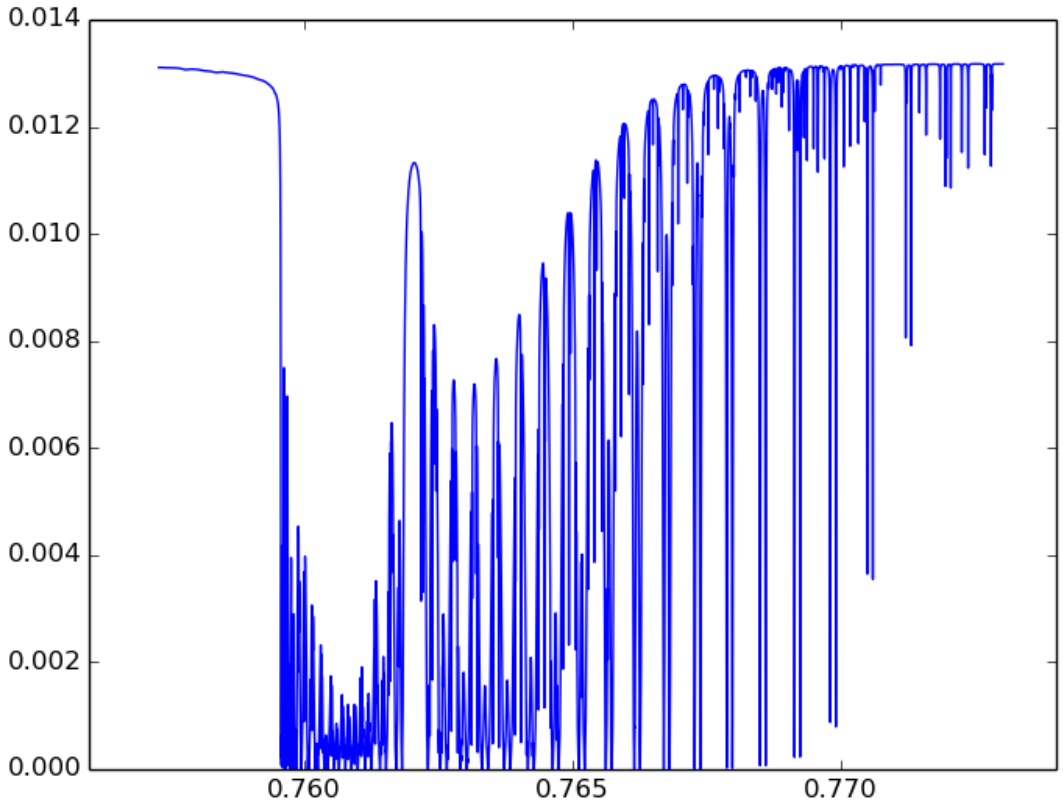
```
In [10]: r = conf.forward_model.radiance_all()
```

6.3.4 Example 4 - Radiative Transfer

Run just the radiative transfer w/o applying solar or instrument model:

```
In [11]: band = 0
In [12]: ils_hw = conf.forward_model.instrument.ils_half_width(band)
In [13]: spec_pix = conf.forward_model.instrument.pixel_spectral_domain(band)
In [14]: spec_samp = conf.forward_model.spectrum_sampling.spectral_domain(band, spec_pix, ils_hw)
```

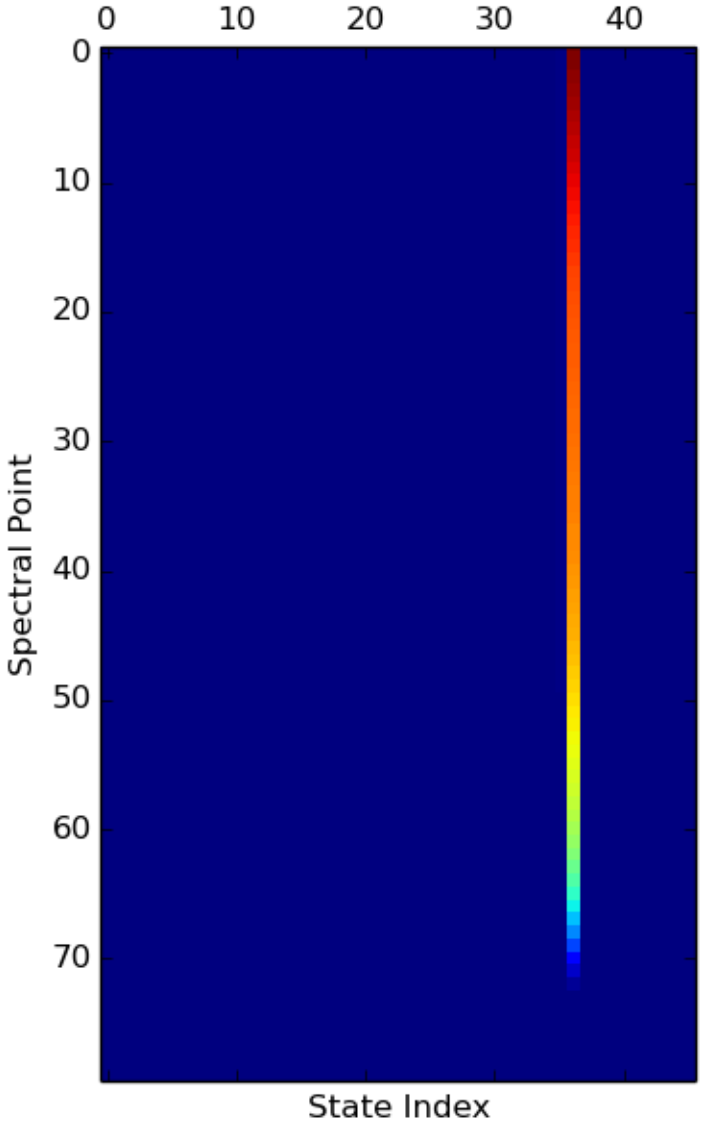
```
In [15]: rrt = conf.forward_model.radiative_transfer.reflectance(spec_samp, band, True)
In [16]: plot(rrt.spectral_domain.data, rrt.spectral_range.data)
```



We can also run the RT with Jacobian not disabled and then plot up a portion of the spectral data (first dim) for all state vector types:

```
In [17]: rrt = conf.forward_model.radiative_transfer.reflectance(spec_samp, band)
In [18]: jac = rrt.spectral_range.data_ad.jacobian
In [19]: matshow(jac[0:80, :], cmap=cm.jet)
In [20]: xlabel("State Index")
In [21]: ylabel("Spectral Point")
```

This shows that the Jacobian is dominated by one value:



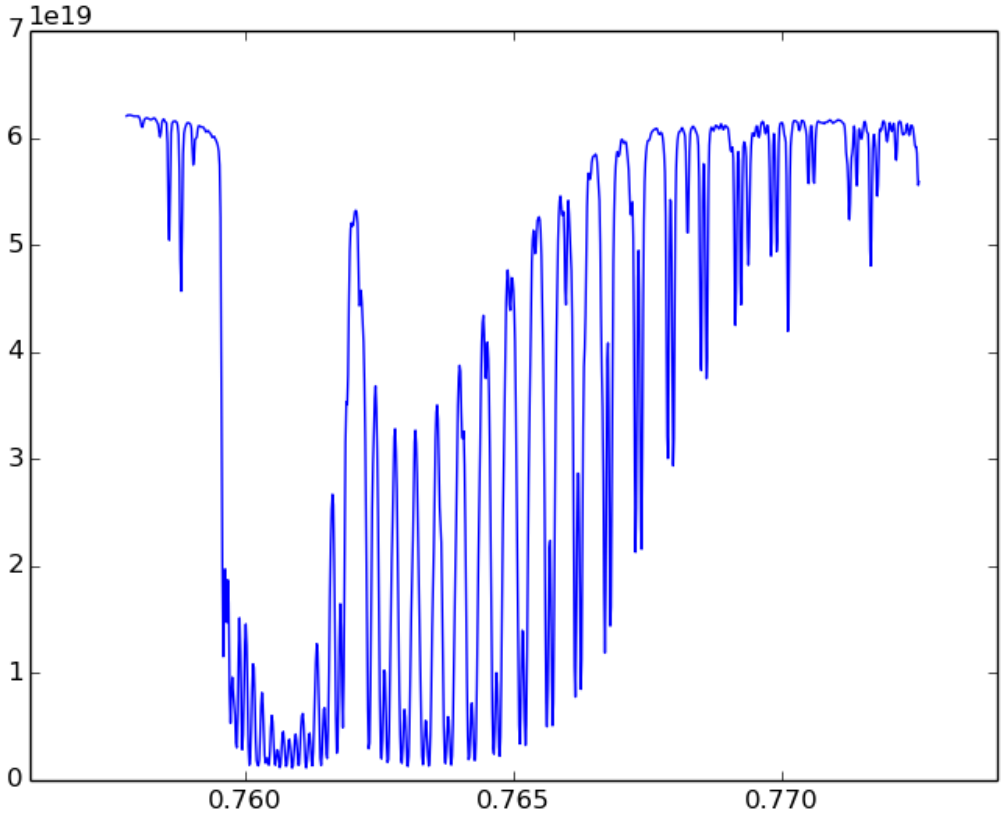
Look at vector and find largest one, and name of it:

```
In [22]: conf.forward_model.state_vector.state_vector_name[argmax(jac[0,:])]
Out[22]: 'Ground Lambertian A-Band Albedo Parm 2'
```

6.3.5 Example 5 - Apply Solar and Instrument Models

To apply the solar model and the instrument model along with other effects we would use the RT spectrum from the last example:

```
In [26]: rinst = conf.forward_model.apply_spectrum_corrections(rrt, band)
In [27]: plot(rinst.spectral_domain.data, rinst.spectral_range.data)
```



6.3.6 Example 6 - Use Generic Solver

DEPRECATED

Python SciPy comes with some generic solvers, including one based on the standard Fortran minpack routines. We have a version of the cost function that uses the more standard format of embedding the a priori values, a priori covariance matrix, and radiance uncertainty:

As an example of using this:

```
import scipy.optimize

cost_func = FmStandardFormatCostFunction(conf.forward_model(), \
        conf.spectral_window_apply().radiance(), \
        conf.spectral_window_apply().radiance_uncertainty(), \
        conf.initial_guess().apriori(), \
        conf.initial_guess().apriori_covariance())

x = conf.initial_guess().initial_guess()
cf = lambda x: cost_func.residual(x)
jf = lambda x: cost_func.jacobian(x)
yinitial = cf(conf.initial_guess().initial_guess())
```

```
print "Initial chisq: %f" % (sum(yinitial * yinitial)/ yinitial.size)
xsol, ier = scipy.optimize.leastsq(cf, x, Dfun=jf, maxfev=10)

if(ier < 0):
    print "An error occured"

ysol = cf(xsol)
print "Final chisq: %f" % (sum(ysol * ysol)/ ysol.size)
```

This print out:

```
Initial chisq: 131.571840
Final chisq: 18.775889
```

NOTE: This example only allows 10 evaluations of the cost function, and uses the default stopping criteria. This is really meant as a quick example, rather than saying this is a particularly good solver. Also, for C+ we were thinking of investigating the GSL. The GSL has python wrappers (PyGSL), and can be used as an alternative to the scipy solver. The standard Level 2 Full Physics solver gets a chisq of 0.614842 in 5 evaluations, so this example obviously needs some tuning to work for real.

6.3.7 More Advanced Example

DEPRECATED

The C+ interface exposed to Python has an interface centered around running the full physics retrieval. As we get feedback, we can extend this interface to be more useful in an investigative ipython environment. But you can also add your own layer of functionality on top of the lower level C+ interface, either as a quick prototype of a C+ interface change or instead of changing the C++.

As a concrete example, the interface to the radiative transfer holds things like the solar zenith angle, number of streams, etc. fixed since these don't vary in the Level 2 Retrieval. But a very useful investigation would be to vary these parameters and see how they affect the Radiative Transfer results.

The ConfigurationHeritage interface in the earlier examples are tied to a particular run configuration file. But there is no reason that you need to create objects strictly from the run file, you can also use the more generic constructors of various classes, or modify objects after they have been created.

Continuing our example, here a wrapper class that sets up Lidort, the LRad polarization correction, and a model atmosphere based on the configuration file. You can then vary parameters such as the surface pressure, solar zenith angle, and number of streams. This example does not include the LSI speed up (since this example looks at a single wavelength), but you could include that is you wanted to for some reason.

We create a new class "RtExtraKnobs". Save this in the file "rt_extra_knobs.py". Note while you are developing code like this, you can repeatedly load updated versions by using the ipython "%run" command (once it is complete, you can just import it like any other module).

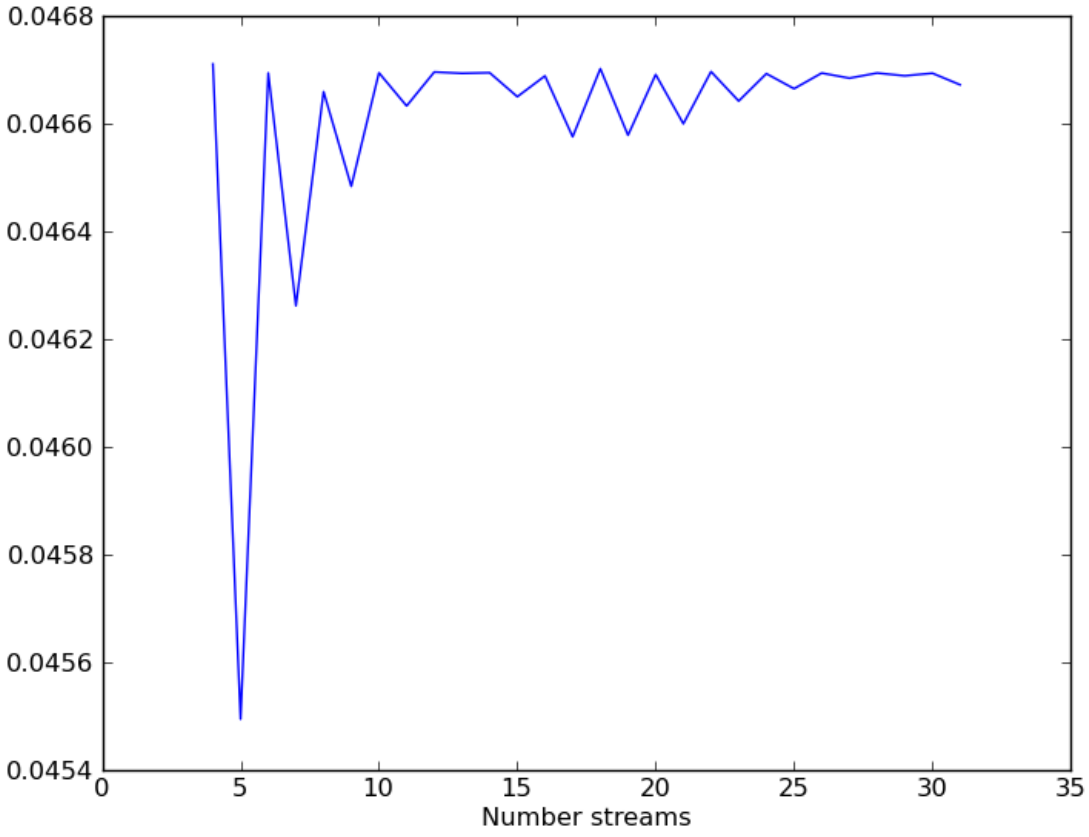
```
import full_physics as fp

class RtExtraKnobs:
    def __init__(self, fname="/groups/algorithm/python_tryout/sample_run/oco_l2.run"):
        conf = fp.ConfigurationHeritage(fname)
        self.conf = conf
        self.atm = conf.atmosphere()
        self.state_vector = conf.state_vector()
        self.level_1b = conf.level_1b()
        self.band = 0

    # Return radiance for single point for given solar zenith, pressure,
```

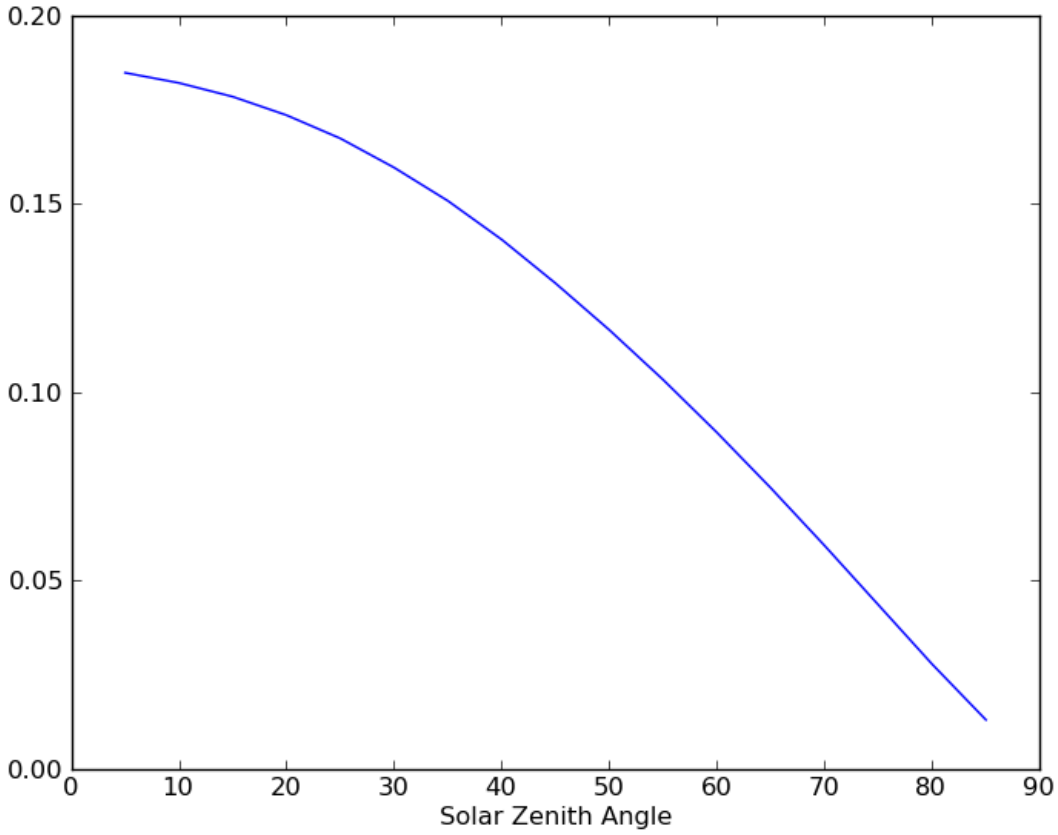
```
    # and number of streams
    def radiance(self, wn, sza, surface_press, nstream):
         try:
             self.atm.pressure().surface_pressure(surface_press)
             rt = self.__rt(sza, nstream)
             # Don't need log message for processing one point
             fp.FpLogger.turn_off_logger()
             return rt.radiance([wn], self.band)[0]
         finally:
             # Turn back on
             fp.FpLogger.turn_on_logger()

    # The relative azimuth needs to be modified because the convention used
    # in the OCO L1B1 file to to take both the solar and observation angles
    # as viewed from an observer standing in the FOV. LIDORT on the other
    # hand has the "follow the photons" convention. This results in a 180
    # degree change
    def __rel_azm(self, band):
         r = (180 + self.level_1b.sounding_azimuth(band)) - \
             self.level_1b.solar_azimuth(band)
         if(r >= 360): r = r - 360
         if(r < 0): r = r + 360
         return r

    # Get RT for a particular solar zenith angle and number of streams
    def __rt(self, solar_zenith, number_stream):
         band = self.band
         # Hardcode these for this example
         nbrdf_quadratures = 50
         nstoke = 3
         ss_corr = True
         delta_m_scaling = True
         uplooking = False
         # Value needed to be true for LRad
         get_rad_dif = True
         nmom = number_stream if(number_stream >= 4) else 4
         rt_lidort = fp.LidortDriver(self.atm, self.state_vector,
                                     [self.level_1b.stokes_coefficient(band)],
                                     [solar_zenith],
                                     [self.level_1b.sounding_zenith(band)],
                                     [self.__rel_azm(band)],
                                     number_stream, nmom, nbrdf_quadratures,
                                     nstoke, get_rad_dif, ss_corr,
                                     delta_m_scaling, uplooking)
         rt_lrad = fp.LRadDriver(rt_lidort, [solar_zenith],
                                 [self.level_1b.sounding_zenith(band)],
                                 [self.__rel_azm(band)])
         return rt_lrad
```

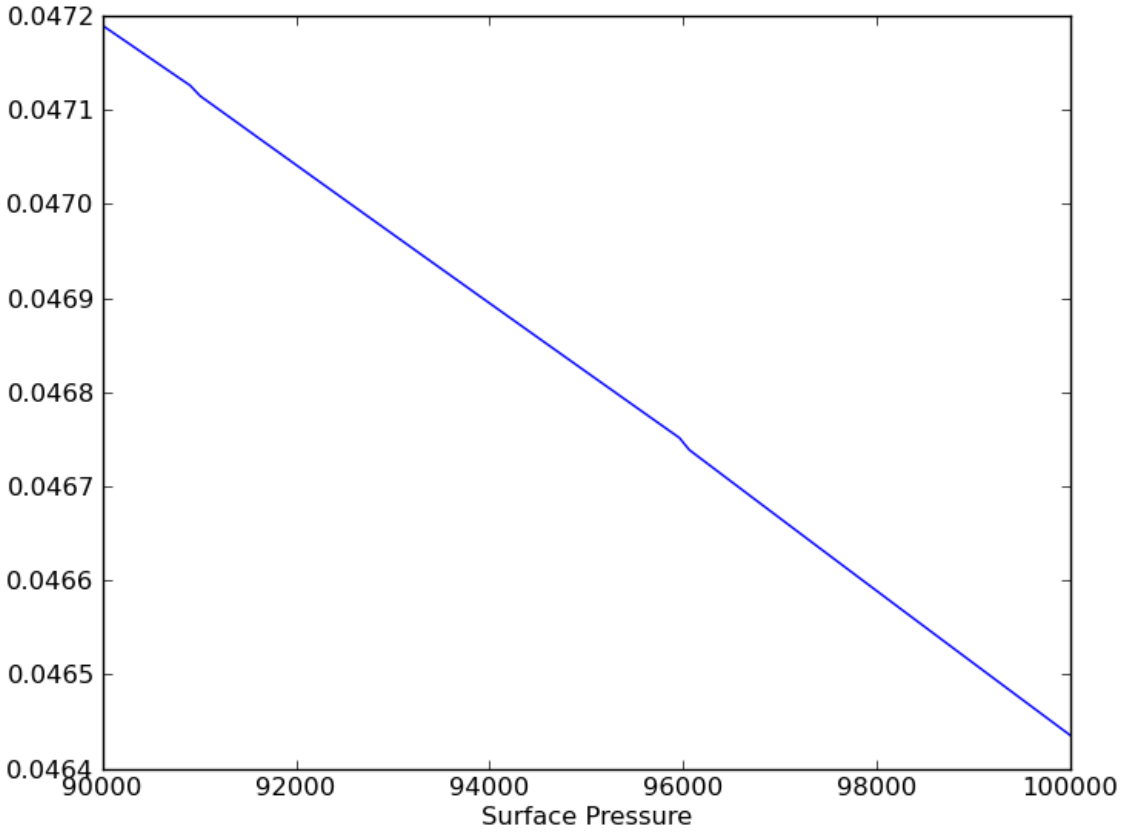
Once we have this module, we can use this in ipython to generate simple plots:

```
from rt_extra_knobs import *
rt = RtExtraKnobs()
wn = 13005.0
sza = 74.0
psurf = 96716.0
nstream = 16
```

```
rad_by_sza = lambda x: rt.radiance(wn,x,psurf, nstream)
rad_by_psurf = lambda x: rt.radiance(wn,sza,x, nstream)
rad_by_nstream = lambda n: rt.radiance(wn,sza,psurf, int(n))

nstream_arr = r_[4:32]
plot(nstream_arr, map(rad_by_nstream, nstream_arr))
pylab.xlabel("Number streams")
sza_arr = r_[0:90:5]
plot(sza_arr, map(rad_by_sza, sza_arr))
pylab.xlabel("Solar Zenith Angle")
psurf_arr = r_[90000:100000:100j]
plot(psurf_arr, map(rad_by_psurf, psurf_arr))
pylab.xlabel("Surface Pressure")
```




0.20
0.15
0.10
0.05
0.00
0
10
20
30
40
50
60
70
80
90
Solar Zenith Angle



6.4 Python Callback

To be documented - providing Python classes to be used in place of C+ (e.g., prototype a new LSI or solar model and test in retrieval).

CHAPTER

SEVEN

DEVELOPER'S INFORMATION

7.1 Introduction

The rest of the user's guide is concerned with Level 2 from a user's point of view, dealing with things like "How do a run the software" and "How do a I build it". This document is concerned with Level 2 from a Developer's point of view. It describes the overall software architecture, how to make changes to the system, etc.

7.2 Build System

We use the same build system used by many open source projects, the standard "configure/make" cycle used to build many linux packages. The details of actually building the software is covered in the compilation section.

We'll give a little more detail in this section on using the build system from a developer's point of view, e.g, "how do I add a file?".

7.2.1 Autotools

The system is built using the standard GNU autotools chain. If you are familiar with this, then this system is pretty standard. Things are set up as with most projects. Different projects seem to put local autoconf macros in different places, on this system they are in config/m4.

If you are only vaguely familiar with autotools, it can be a bit confusing about what files we generate, and which are derived files.

The file ./booststrap can be used to initial generate the derived files.

The input files are configure.in and Makefile.am. From these files the ./configure and Makefile.in are generated. In addition to these file, there are local autoconf macros in config/m4, and derived files used by configure in the directory config. You can generally ignore these - you really need to know how to use autotools before you want to mess around with the autoconf macros (it isn't hard, just a bit obscure).

To make it manageable, Makefile.am is broken into a number of pieces that are included into the top level Makefile.am file. Each directory with code has its own .am file that describes the code in that directory, for instance exe/full_physics/full_physics.am.

Once these files are generated, the actual build is done by ./configure and make cycle that should be familiar if you've installed any of the GNU software before. This generates an actual Makefile.

The specific tools used for each piece:

- automake - most commonly used piece. This takes the various ".am" files and produces a Makefile.in file. This needs to be run whenever you change one of the automake files.

- autoconf - produces the "configure" file. This takes the macros and configure.in file and produces the configure file. Need to run this when configure.in has been changed.
- aclocal - collects all the local macros and prepares them for use with autoconf. Need to run this if you change any of the local ".m4" files.
- libtoolize - prepares libtool for use with automake. Only need to run this to update the version of libtool used.

If can be easier to just run automake and the other tools by hand when needed, particularly if you only change something once in a while.

However if you are going to regularly be modifying things (e.g., you are a developer), it can be convenient to enable "maintainer mode" in the Makefiles. This adds rules in the Makefile to rebuild the ./configure and Makefile.in files as needed. To turn this on, you add the option "–enable-maintainer-mode" when you do configure, e.g, "./configure –enable-maintainer-mode".

There is nothing magic about maintainer mode, it just runs automake and autoconf when input files have changed.

As you use autotools more, you may want to consult the Automake manual. A very nice introduction to autotools is Autotools: a practitioner's guide to Autoconf, Automake and Libtool.

7.2.2 Autotools Problems

Autotools has been showing its age and haphazard design for some time (often referred to as "auto-hell" for example see: "Why the KDE project switched to CMake"), however right now there isn't an obvious replacement. Most FOSS project still use these tools, so for now we'll continue to do the same. KDE recently moved to CMake, but right now they are the only large project to use CMake.

For a user, autotools stuff work fine, all of the complication is in writing the input files during development. Fortunately, Level 2 Physics isn't particularly complicated compared to other projects (like KDE), so for now this complication is manageable.

In the future, if there is a clear replacement for autotools, we may want to move this system to that.

7.2.3 Adding New Code

There are two steps to adding new code:

1. Update the corresponding ".am" file
2. Run automake (either by hand or automatically if you did "–enable-maintainer-mode"

The easiest way to add code is to open one of the existing ".am" files such as "implementation.am" and copy what is done there. Code is added by adding to make variables. There are typically 4 files associated with a new piece of code (one or more may be missing, in which case you just leave them off)

- The header file (".h") gets added to "fullphysicsinc_HEADERS".
- The source file (".cc", ".F90") get added to "libfp_la_SOURCES".
- The SWIG interface file (".i") gets added to "SWIG_SOURCES".
- The unit test file ("_test.cc") get added to "lib_test_all_SOURCES".

After added the code, run automake to regenerate the Makefile.in

7.3 Platform Information

7.3.1 Mac

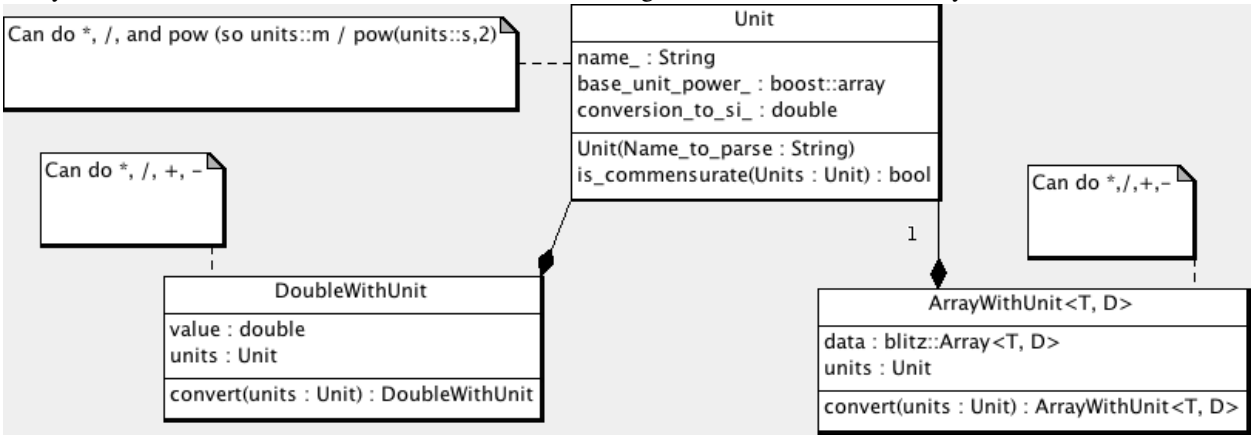
The Mac does not put debugging information into an executable when you build it. Instead, there is a separate step using the "dsymutil" program. So if you want to debug l2_fp or test_all you need to first run dsymutil to get debugging information into it.

7.4 Design Information

7.4.1 Unit classes

We track physical units in our code (e.g., a velocity may be marked as "m/s"). This handles conversions when needed (e.g., add 10 m/s to 15 km/hr), and detects error with units that don't match (e.g., add 5 m to 3 hr).

The unit tracking is handled by the class "Unit". Two closely related classes are DoubleWithUnit and the template ArrayWithUnit<T, D>. These two classes contain units along with a double or blitz::Array<T, D>.



The class Unit does dimensional analysis. We track the power of each of the SI base units. In order, these are meter, kilogram, second, Kelvin, Ampere, mole, candela, steradian, radian, photon

Note that steradian and radian are actually dimensionless, but it is useful to track them. Also photon is a photon count, which doesn't really have units either. But it is useful to track because we can determine the photon count at a particular wavelength to convert to cm^-1.

The units have a "name" associated with them. This is a free form string. The intent is this represents the units is a human readable form. We need to have this attached rather than automatically generated because depending on the context we may represent the units in different manners. For example, the radiance units are usually given as "W / m^2 / sr / cm^-1". This is exactly the same as "0.01 kg * m / s^3 / sr", but the latter would be a fairly confusing way to label the radiance. There is no real way to have a program know how we want the unit represented as a string, so the easiest thing to do is to just attach it.

Note there is no problem going the other way. Given a string, we can uniquely parse it to give a Unit object. This is supported by the unit constructor, which can take strings such as "m/s" or "W / m^s / sr / cm^-1".

We can combine units using the operation "*", "/" and "pow'. These handle the dimensional analysis, and creates a basic name. The name is generated by simple combination rules, but you may want to override the generated name with a preferred string. So for example kg * m / s * s can be called "N" for Newton.

The DoubleWithUnit and ArrayWithUnit can be combined with the normal math operations of *, /, +, and -. You can also convert to a new set of units using "convert".

7.4.2 Compile vs. Runtime classes

The Unit class described in the previous section has the feature that the units are determined at run time. There is another class of unit libraries such as boost::units where the determination is at compile time. This is one of those classic design trade offs.

The compile time unit classes have the significant advantage that the units are determined at compile time. There is **no** runtime cost, everything is determined at compile time. All unit errors are also determined at compile time. However, it has the disadvantage that you need to know what the units are at compile time.

In our case, the overhead of the units class is minimal. We are pretty much insensitive to time overhead except in the innermost wavenumber/wavelength loop in the forward model. All of our unit calculations are outside of this loop, so the minor overhead in run time is acceptable. We want to be able to determine the units at run time, so for example OCO has wavelength given in microns while GOSAT has wavenumber given in cm^-1. Which kind of processing we are doing is determined at run time by the contents of the Lua configuration file.

7.4.3 Instrument

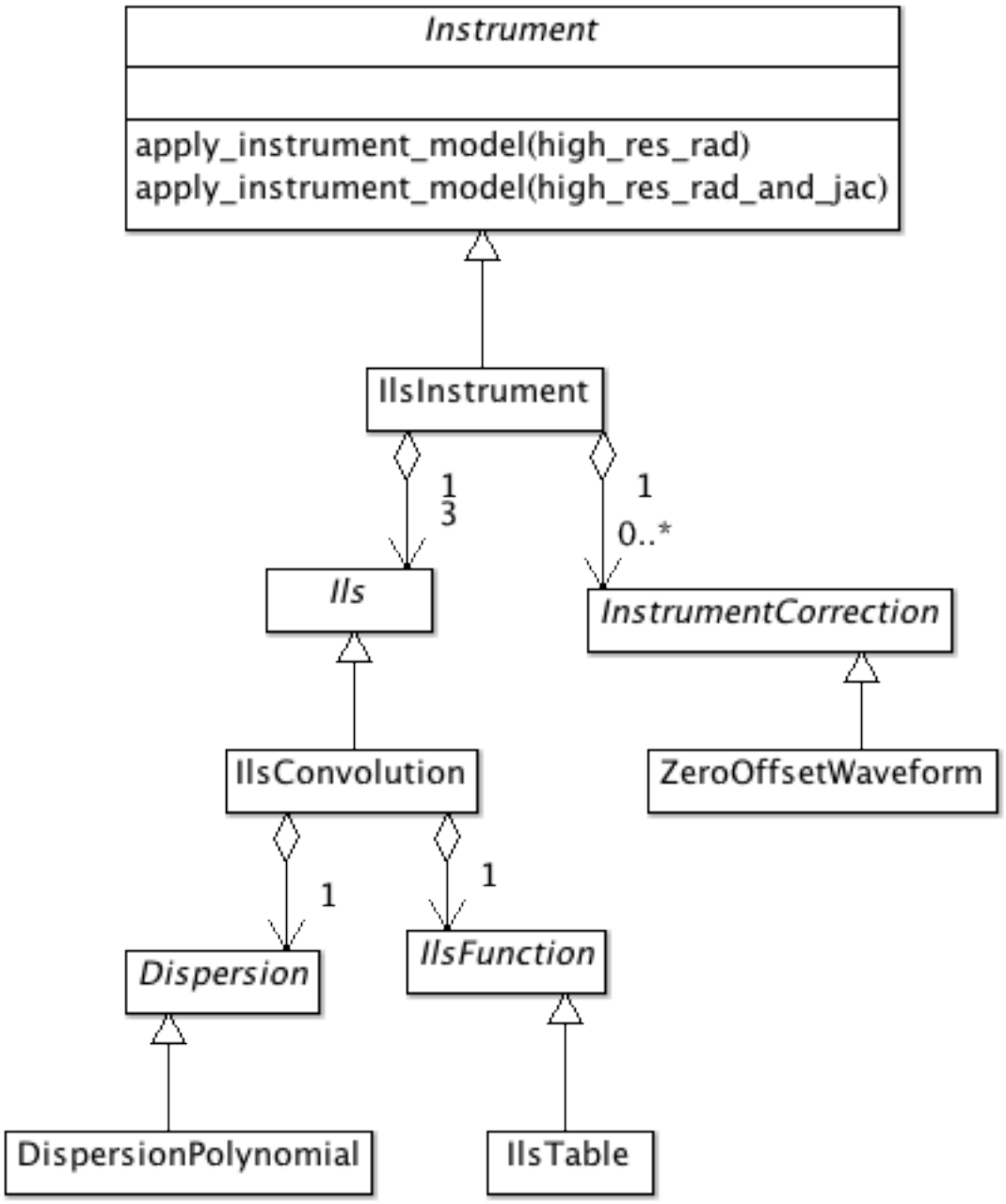
The Instrument class is used by the ForwardModel to model the measurement instrument. This class takes the radiance values calculated on the high resolution spectral grid (using the RadiativeTransfer and SolarModel classes), and produces a low resolution spectral grid of radiance values. This data is what our instrument model would have seen if it was to observe the high resolution spectral grid.

Any object of the Instrument class can be used, this just needs to supply one major function "apply_instrument_model". There are two variations of this function, one that only processes the radiance data and one that also processes the Jacobian of the radiance data with respect to the StateVector.

If you are implementing a new instrument, you just need to match the interface specified by Instrument. However, in practice your we have a instrument model as described in the OCO ATB. To implement this, we divide the different pieces of the instrument calculation into different pieces, as described in the next section.

IlsInstrument

The particular implementation we use in our code is described in the following diagram:



We divide the calculation into two major pieces. The modeling of the Instrument Line Shape (ILS) is done by the Ils class. There is one object for each spectral band. In principle we could use different models of of the ILS, and indeed a different model for each band. But in practice, we always do a convolution, using the IlsConvolution class. We then apply zero or more instrument corrections. This is where we do things like a zero offset correction or a continuum correction.

Ils

The Ils class is responsible for doing modeling the Instrument Line Shape. There is one main function that needs to be implements "apply_ils". This has two variations, with and without also calculation the Jacobian.

You can use any object of the Ils class, but in practice we have one implementation we use IlsConvolution. This divides the calculation into three pieces. The dispersion calculation is done by a Dispersion object, this determines the

wavenumbers for each of the instrument pixels we will be calculating. The IlsFunction determines the values we will be convolving with for a particular pixel And the IlsConvolution class actually convolves the high resolution spectra with the IlsFunction to get the low resolution instrument spectra.

InstrumentCorrection

After the ILS is done, zero or more instrument corrections can be applied. There is just one major function that needs to be supplied, "apply_correction". This has two flavors, with and without Jacobian calculations.

Dispersion

The Dispersion object determines the wavenumber of each instrument pixel. It supplies one function, "pixel_wavenumber".

IlsFunction

The IlsFunction object supplies one function, "ils".

7.4.4 Lua Config

Introduction

We generate our configuration files in Lua. Lua is a simple language that integrates well with C/C++, and is ideal for things like configuration files.We use the package Luabind for wrapping our C++ code for use in the configuration files.

Documenation:

- Lua documentation
- Lua Wiki
- Luabind
- We have stashed copies of the Lua and Luabind manuals in the source tree under doc. If you are off the web, or there is a problem with one of these web sites you can consult the documentation there.

Lua For the Impatient

You should consult the documentation for details, but Lua is a very simple language (which is why we selected it). If you have used any procedural/OO language you should be able to pick up the syntax quickly. A few things to note:

1. Comments start with "–" and go to the end of the line (like C++ "//").
2. Classes are slightly different in Lua. It doesn't directly support them, but supports "Tables" where are fairly similar. Like python, all function that you would think of as object oriented take "self" as the first argument. You can call a function on a class using the standard ".", but if you do that you need to explicitly pass the object is, so "foo.func(foo, arg1, arg2)". As a special notation, you can instead use ":" which automatically adds the object as the first argument, so "foo.func(foo,arg1, arg2)" is exactly the same as "foo:func(arg1, arg2)". By convention, you should use the second form.

Wrapping C++ code

To create classes in Lua, we need to be able to call C++ code. The connection between Lua and C++ is handled by the Luabind library. This is a template based library that automatically generates the glue code between the two languages.

The registration of the C++ code is handled by the RegisterLua class.

One approach to this is to have a central function that registers everything, and as we add classes update that central function. An alternative is the one selected here, were we have a more decentralized registration. Classes set up the registration in their own area, and then simple get listed and needing registration in the file "register_lua.cc". It would be nice to decentralize this completely, but I could never figure out a way to actually do this.

So registration involves 2 steps:

1. Add the registration code to the class code (e.g., for Foo, this is the file "foo.cc").
2. Add the class to the list of classes in the function RegisterLua::register_lua found at "lib/Lua/register_lua.cc"

The registration code is cookie cutter, so we have macros to help do this. The registration is different depending on if we have a derived class with a base class, or a class that doesn't derive from another (or at least one that we want to tell Lua about).

An example of this in level_1b.cc:

```
#ifdef HAVE_LUA
   #include "register_lua.h"
   REGISTER_LUA_CLASS(Level1b)
   REGISTER_LUA_END()
#endif
```

In level_1b_hdf.cc:

```
#ifdef HAVE_LUA
   #include "register_lua.h"
   REGISTER_LUA_DERIVED_CLASS(Level1bAcos, Level1b)
     .def(luabind::constructor<std::string, std::string>())
   REGISTER_LUA_END()
#endif
```

Then in register_lua.cc, we add:

```
REGISTER_LUA_LIST(Level1b);
REGISTER_LUA_LIST(Level1Hdf);
```

Note that you don't need to put all the member functions into the Lua registration, just the ones you want to call in Lua. For many classes, this will just be the constructors. We use Lua configuration files for creating the objects needed in Level 2 Full physics, not to do major computation with it. That is more what we do with the Python wrappers. Lua is a small language that is ideal for integration in the C++ code, but it is no replacement for Python, nor is it meant to be.

Pretty much all our classes are Printable. We've put the magic incantation in place for classes in the macros (this ties the Lua function __tostring__ to the C++ code print_to_string). If you have a class that is not printable, we'll need to add a macro to support that.

We normally use Lua through our C++ code. It can be useful, particularly when testing, to go the other way. We define the function "luaopen_fullphysics" to go the other way, call in Lua like:

```
require("libfull_physics")
```

Note that you should use the installed library, like we do with python (i.e., do a "make install").

You will need to make sure that the library is on the PATH. Lua uses an odd syntax for its path, an example of using it would be:

```
LUA_CPATH=install/lib/?.so lua
require "libfull_physics"
l1b = Level1bAcos("filename","soundingid")
```

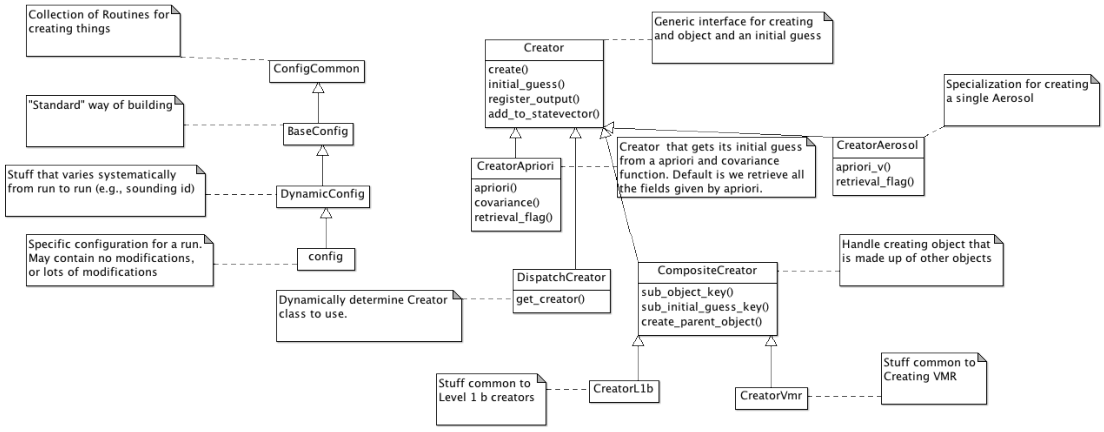
Writing Configuration Files

The configuration file purposely have very minimum requirements. To use in Level 2 full physics, you just need to create a handful of global variables however you would like. These variables are listed in the config.lua file found in input/gosat/config/config.lua. This includes things like "forward_model" and "solver".

However, most of the time you will want to use a "standard" run with possibly some local modifications. The "standard" run uses a nested set of files:

- config_common.lua - General purpose routines for creating Level 2 objects
- base_config.lua - The "standard" way of building things
- dynamic_config.lua - Things that change from one run to the next, but in a systematic way (e.g, sounding_id, surface_type)
- config.lua - The local config file. In the simplest case, just include dynamic_config.lua without change. But can contain local modifications.

It turns out that constructing objects tends to follow the same pattern for the many kinds of Level 2 objects we create. We have introduce a "Creator" Lua class (*not* Level 2 C++ class), along with a few derived classes to handle common scenarios. The UML diagram of this code:



Debugging Configuration Files

Because the configuration files are actual Lua code, you can get errors in the file.

If a Lua error occurs, you can optionally turn on diagnostic messages by setting the diagnostic flag to true in the Lua file. This will print some tracing messages, which will help you locate the portion of the Lua where an error occurs.

CHAPTER

EIGHT

OTHER DOCUMENTATION

8.1 ATBD

The OCO-2 algorithm theoretical document describes the algorithms in the RT Retrieval Framework used to retrieve the column-averaged CO2 dry air mole fraction XCO2 and other quantities included in the Level 2 (L2) Product from the spectra collected by the Orbiting Carbon Observatory-2 (OCO-2). It identifies sources of input data, describes the physical theory and mathematical background underlying the use of this information in the retrievals; includes implementation details; and summarizes the assumptions and limitations of the adopted approach.

8.2 Doxygen

Doxygen documentation can be created from the build directory:

```
$ make doxygen-run
```

The resulting documentation will be available in the `doc/html` subdirectory.

8.3 Python

Sphinx Python documentation can be built along with the Python bindings:

```
$ /path/to/rtr_framework/configure --with-python-swig --with-documentation
$ make install
```

The `--with-documentation` creates the Sphinx files along with the code compilation. The resulting files will be in the `install/share/doc/full_physics/html` subdirectories.

CHAPTER

NINE

ACRONYMS

ACOS	Atmospheric Carbon dioxide Observations from Space
FP	Full Physics
FTS	Fourier Transform Spectrometer
GOSAT	Greenhouse gases Observing SATellite “IBUKI”
L2	Level 2
OCO	Orbiting Carbon Observatory