User's Guide

James McDuffie

2013/01/16 09:12

Table of Contents

Account Access

SCF Account Access

Science Team members who need to use the Science Computing Facility must submit an account request form to gain access. You can download a copy of this from from this wiki: OCOAccountRequestFormNew-Electronic.pdf. The latest copy of this form is available on the JPL internal network through Docushare: OCO Account Request Form

Those who can not directly access these documents directly (for instance if you are an outside developer with no access to the internal JPL network) should contact the document maintainer Margareth Olm <Margareth.Olm@jpl.nasa.gov> for a copy. She can also advise on the process for external users which is detailed in this document: OCO.EUCA.Process2.pdf of which the latest version is stored in docushare: OCO Project Process for Access to Project Computers

Subversion Account Access

Team members who will be working on the Level 2 Full Physics code will also need an account to the Subversion repository. Subversion is a configuration control and management system that tracks and maintains current and historical versions of files such as source code, web pages, and documentation.

Subversion access is only available to those who have already been approved for to the Science Computing Facility. Requests for a Subversion account should be sent to Julie Foster <Julie.C.Foster@jpl.nasa.gov> and Charlie Avis <Charles.C.Avis@jpl.nasa.gov>. If a team member already has a Subversion account but forgot either their login or their password, they should contact Julie Webster for help.

SCF System Setup

SCF System Information

The ACOS SCF system has several servers available for use development and analysis purposes.

Fullerene (fullerene.jpl.nasa.gov) is the front end to the TORQUE cluster and a file server. CPU-intensive programs should not be run directly on the head node as these will affect the performance of the entire cluster. Instead run these types of jobs through TORQUE or one of the interactive machines as mentioned below.

The cluster nodes are managed by the TORQUE Resource Manage. The cluster has many nodes available for general use that are identical in software configuration but not necessarily in hardware configuration.

Various queues are available depending on the type of job you wish to run as well as to differentiate between different hardware architectures. The short, long and verylong queues share the same Intel-only nodes; jobs submitted to the short queue will take priority over jobs submitted to the long queue. These queues differ in the amount of time a job is permitted to run and the priority assigned to them. The amd queue consists purely of AMD nodes.

The following shows the difference in node hardware configuration:

Node sif-0-0 to sif-0-31 (short, long, verylong queues):

- 2 x Six-Core Intel 64bit Xeon X5670 Westmere Processors (2.93Ghz) (12 cores total)
- 16 GB Ram

Node sif-0-32 to sif-0-60 (short, long, verylong queues):

- 2 x Quad-Core Intel 64bit Xeon E5450 Processors (3Ghz) (8 cores total)
- 16 GB Ram

Node sif0-61 to sif-0-95 (amd queue):

- 2 x Dual-Core AMD 64bit Opteron 2212 Processors (2Ghz) (4 cores total)
- 8 GB Ram

The cluster resources can be monitored in real time by going to http://fullerene.jpl.nasa.gov/ which is accessible inside of JPL. If you need to view the cluster resources from the command line you can use the 'gstat -p8639' command on fullerene.

Two interactive machines are available for tasks not appropriate for submission to the TORQUE system:

- scf-srv (scf-srv.jpl.nasa.gov) is for interactive jobs such as running IDL, also this machine has a PAR and can be sshed into from offlab
- scf-srv2 (scf-srv2.jpl.nasa.gov) is for compiling and debugging interactively

scf-srv2 server has the following compilers available for use:

- IDL 7.1 (/opt/local/depot/rsi-64/idl/bin) (6 licenses total)
- Absoft 10.2 (/opt/absoft10.2/bin)
- NAGWare 5.1 (/opt/local/depot/NAGWare-64/5.1/bin)
- Intel IFORT 10.1.015 (/opt/local/depot/intel/10.1.015/bin)

Please use these servers for any compiler, or IDL related tasks that can not be run on the nodes. The fullerene nodes should be used for CPU intensive activities. The L2 Full Physics Users Guide contains details on running jobs interatively on the SCF nodes.

The systems are currently running CentOS 5.5.

If you need to access the servers we do have SSH running on both scf-srv and fullerene with a PAR available for external users. You should be able to connect directly to both systems from inside and outside of JPL. We do have a security measure in place which will lock you out of the system for a few minutes if you enter your password incorrectly after 3 tries. Please do let us know if you have troubles connecting.

We would also like to remind everyone to please remember that ONLY /home is being backed up. The /home directory is shared between scf-srv, fullerene, and nodes. We are also using NIS between the system so your password can be updated or changed on either scf-srv or fullerene.

Please let the ACOS system administrators know if you have any questions. Contact them at: ocohelp@list.jpl.nasa.gov.

Deployment

Environment Setup

The following sections describe how to retrieve and initialize the tools and source code needed to develop and utilize the resources available for running the L2 Full Physics 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.

These instructions are only specific to the ACOS/OCO Science Computing Facility where indicated.

Base Directory

It is recommended that your copy of the OCO L2 FP tools and source are located in your home directory under one common subdirectory. The recommended name for this base directory is: "*acos*". Perform these steps:

```
~]$ cd ~/
~]$ mkdir acos
~]$ cd acos
```

Subversion Setup

It is recommended that users set up an environmental variable that specifies the root URI (Uniform Resource Identifier) for the Subversion repository used by the ACOS/OCO Science Team.

In your /.bashrc file add the following line:

```
export SVNROOT=https://svn.jpl.nasa.gov/oco/alg/
```

You will need to logout and log back in for this change to take effect or enter the above line into the bash prompt to have it available in your current bash shell instance.

Development Environment Setup

The next step downloads a copy of the Level 2 development environment into your base directory. These steps assume that your current directory is: "/acos"

```
acos]$ svn co $SVNROOT/level_2/trunk level_2
```

If Subversion prompts you with this message:

```
ATTENTION!  Your password for authentication realm:
   <https://svn.jpl.nasa.gov:443> OCO Subversion repository
can only be stored to disk unencrypted!  You are advised to configure
your system so that Subversion can store passwords encrypted, if
possible.  See the documentation for details.
You can avoid future appearances of this warning by setting the value
of the 'store-plaintext-passwords' option to either 'yes' or 'no' in
'/home/yourusername/.subversion/servers'.
```

Then say no and alert your system administrator that Subversion is not set up in a secure manner. Otherwise you will need to enter your password every time subversion is used.

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 ~/acos/level_2/setup_env.sh
```

The path used above assumes you have checked out these packages in the recommend location.

The direct sub-directories of the level2 *directory each contain a setup*env.sh script that is sourced by the top level setup*env.sh. One could reorganize the sub-directories of the level*2 directory differently than they exist in the repository and would only need to source the individual setup*env.sh files in their bash start up script. The order that the packages are sourced into the environment does not matter. The setup scripts simply allow the various sub directories to find each other irrespective of their real location.*

Development Environment Organization

The L2 development environment checked out in the preceding section has the following sub-directories:

lib/full*physics*	L2 Full Physics software
input/	Input files for L2 FP and Absco
operations/	Operational utilities: Cluster Tools, Populator
support/	Supporting Python library, supporting utilities
tests/	End-to-End tests
unit_test_data	Inputs and expected results used by unit tests

Resources Available to L2 Full Physics

Because of the use of large temporarily allocated arrays in some L2 Full Physics routines an adequate stack size needs to be available or the program may mysteriously crash. If you are using bash then set the maximum stack size to something at least greater than or equal to 10240:

```
]$ ulimit -s 10240
```

Alternatively you can just set the stack size to unlimited if you are the trusting sort or if the above stack size is not enough:

```
]$ ulimit -s unlimited
```

Compilation

L2 Full Physics Code Compilation

The Level 2 Full Physics code uses the standard autotools used by most Linux software. This is the standard configure/make cycle you would have used if you have installed Linux 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 changes, e.g. an optimized and debug version.

Build at JPL

After following the Environment Setup instructions you can build on scf-srv by doing the following:

```
scf-srv% mkdir BuildOptimized
scf-srv% cd BuildOptimized
scf-srv% /your/path/to/level_2/configure
scf-srv% make -j 9 all
```

In the above replace /your/path/to/level2 *with the location where you downloaded the Level 2 software during the Environment Setup instructions.*

The "-j 9" uses 9 parallel jobs during the build. This is appropriate for scf-srv, which has 8 CPUs, but if you are building on your own system you'll want to pick an appropriate value. You can also leave this off if you don't want to do a parallel build.

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:       /home/smyth/Level2Temp/install
  Build debug version:
  Fortran compiler type:        ifort
  Fortran compiler:             /opt/local/depot/intel/11.1/064/bin/intel64/ifort -g -xSSE2 -O3 -Difort -heap-arrays 1024
  C compiler:                   gcc -g -O2
  LD Flags:                     -R /opt/local/depot/intel/11.1/064/lib/intel64:/opt/local/depot/intel/11.1/064/mkl/lib/em64t:/opt/local/
depot/intel/11.1/064/lib/intel64:/opt/local/depot/intel/11.1/064/mkl/lib/em64t
  HDF5 support:                 yes
  Build own HDF5 library:       yes
  Build own LIDORT library:     yes
  Install documentation:        no
```

If you get any kind of an error rather than this message, you can send the file "config.log" to Mike or James to have us determine what the problem is.

We've found that gfortran works better with valgrind and debugging (although our "official" builds use ifort). The default system gfortran is too old, we use Fortran 2003 features not available in version 4.1.2 installed on the system. There is a newer version installed in /opt/local/depot/gcc-4.5.1. To use gfortran, you can do the following configure command:

```
/your/path/to/level_2/configure FC="/opt/local/depot/gcc-4.5.1/bin/gfortran" CC="/opt/local/depot/gcc-4.5.1/bin/gcc"  CXX="/
opt/local/depot/gcc-4.5.1/bin/g++" <other config options>
```

Non-JPL Build

For the simplest way to build, we have a distribution tar file. This contains a simple script for building everything. You can get a copy from JPL at /groups/algorithm/l2*fp/distribution, the current latest is /groups/algorithm/l*2fp/distribution/ full*physics-2.08.00.tar.gz.*

```
mymachine% tar -xf full_physics-2.08.00.tar.gz
mymachine% cd full-physics-2.08.00
mymachine% ./nonjpl_build.script
```

You can also check a copy out of subversion and build the software directly. If you are building on your own non-JPL system, you'll need to build your own copy of the thirdparty libraries. You can do that like:

```
mymachine% mkdir BuildOptimized
mymachine% cd BuildOptimized
mymachine% /your/path/to/level_2/configure THIRDPARTY=build
mymachine% make all
```

In the above replace /your/path/to/level*2 with the location where you downloaded the Level 2 software during the* *Environment Setup* *instructions.*

In order to run all of the unit tests, you'll also need to have a copy of the ABSCO tables. The default location of the ABSCO is /groups/algorithm/l2*fp/absco/v3.1.0*alpha, but you can specify another location using the -with-absco option.

Build Results

Another useful target is "install", which will build the l2*fp and copy it and all associated libraries to a separate install directory. This will be "./install" if you don't otherwise specify it. 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.*

The executable is "l2*fp", and will be placed in the top of the build directory.*

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 (>= 4.3).

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".

Note that as of the current build (2.06.02), HDF-5 can only be built using ifort or gfortran. The other compilers will likely build if you don't try to build hdf5 (i.e., you don't have -with-hdf5=build).

gfortran seems to work better for debugging than ifort. It is a good idea to use the latest GCC/gfotran for debugging. For example to create a debug build with GCC/gfortran 4.5 do the following:

```
/path/to/level_2/configure --enable-debug FC=gfortran-4.5 CC=gcc-4.5 CXX=g++-4.5
```

Build Options

The software uses the third party libraries such as LIDORT and HDF. If you are building on the JPL machines, we have versions of these libraries already installed on the system, which the configuration scripts know where to find.

If you are working on an non JPL machine, or simply want your own copy of these libraries you can specify configure options such as "with-hdf5=build", or "with-lidort=build". This will build your own version of these libraries. This is particularly useful to do if you are actually modifying, for instance the LIDORT code (e.g., a bug fix). The full list of third party libraries can be seen by running configure with the -help option.

If you want to build a local copy of all the third party libraries, you can specify "THIRDPARTY=build". If you have a central location that you've set up third party libraries that isn't already searched, you can add "THIRDPARTY=<dir>".

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 editting 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

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.

Python Wrappers

We have wrappers that allow the C+ full physics library to be used in python.

The C+ interface is a bit awkward in some places. Until very recently, we needed to be able to call the old Fortran code which constrained various data structure such as the Atmosphere to match the older Fortran interface. That constraint is now gone, but we haven't yet reworked the interfaces for instantiating classes.

If you are creating objects to match an existing oco*l2.run file, the easiest thing to do is use our "ConfigurationHeritage" class. This will read the file and create objects for you. You can also directly create objects, you'll just need to look through the C+ documentation for how to do this.*

The exact same doxygen documentation is available in python using the standard python help system, in particular you can just type "help ConfigurationHeritage" or whatever at the python prompt.

We also have Python Documentation.

The wrappers are just straight python, but for interactive use I'd recommend using scipy's ipython, as well as the matplotlib and scipy libraries. If you aren't familiar when them, you can take a look at http://www.scipy.org/Getting_Started.

Building

*If you just want to try out the python wrappers, there is an already built system at "/groups/algorithm/python*tryout". You can skip building if you want to.

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:

```
../Level2/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 can build with:

```
make -j 8 all && make install
```

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 command installed in the install area:

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

If you are using the already built system, replace this with:

```
source /groups/algorithm/python_tryout/setup_fp_env.sh
```

Information Hiding

One minor issue - the ConfigurationHeritage purposely does information hiding. For example, it returns a general "RadiativeTransfer" class rather than the specific type LsiRt or LidortDriver. This makes a lot sense for the C+ code, but not necessarily for interactive use where you may want to use features that only apply to a LsiRt or LidortDriver. To address this, I've put in some python only conversions such as "LsiRt.ToLsiRt" which takes a generic RadiativeTransfer and if it is actually a LsiRt convert it to that type. Nothing bad happens if you call this with a RadiativeTransfer that is really a LRadDriver - all that happens is the Python "None" object is returned rather than a LsiRt. This is in the python documentation, but ask me if you have any issues. You can find out the underlying type of any object by doing a "print", so "print ConfigurationHeritage.radiative*transfer()" will tell you the specify type of object that was created.*

Examples

Tests all start with:

```
source ./install/setup_fp_env.sh # or "source /groups/algorithm/python_tryout/setup_fp_env.sh"
ipython -pylab
from scipy import *
from full_physics import *
conf = ConfigurationHeritage("/groups/algorithm/python_tryout/sample_run/oco_l2.run")
```

If you are working with this often, then you can place this into your /.ipython/ipythonrc file.

The sample run is just a particular canned test we use a lot during development. If you have another run you are interested in, you can just point to that configuration file instead.

Example 1 - Plotting Solar Spectrum

Get help on what conf is and can do:

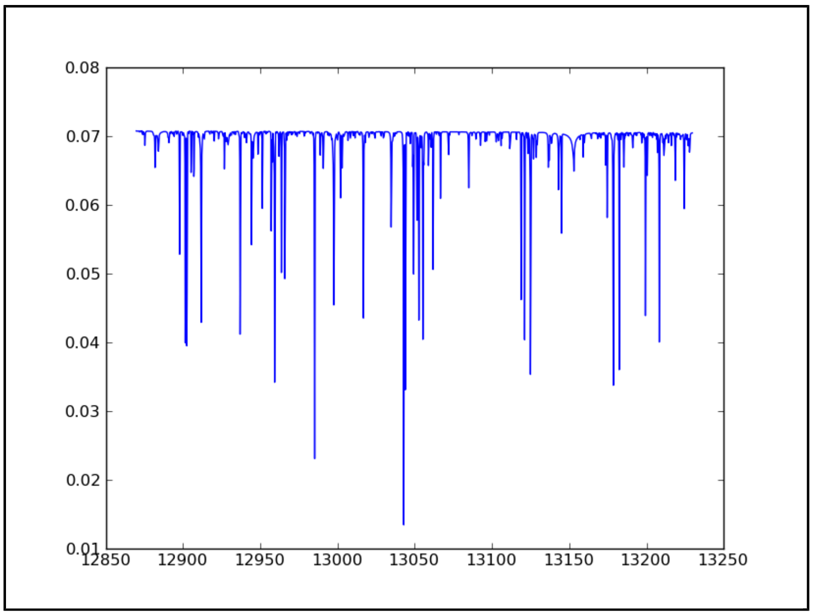
```
help conf
```

Get help on what solar model is and can do:

```
help conf.solar_model(0)
```

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.

```
# Pixel wavenumbers we have in band 0
band = 0
wn = conf.instrument().pixel_wavenumber(band)
plot(wn, conf.solar_model(band).solar_spectrum(wn))
```



Example 2 - Forward model without Jacobian

Run forward model and get radiance for all the bands

```
r = conf.forward_model().radiance()
```

Example 3 - Forward model with Jacobian

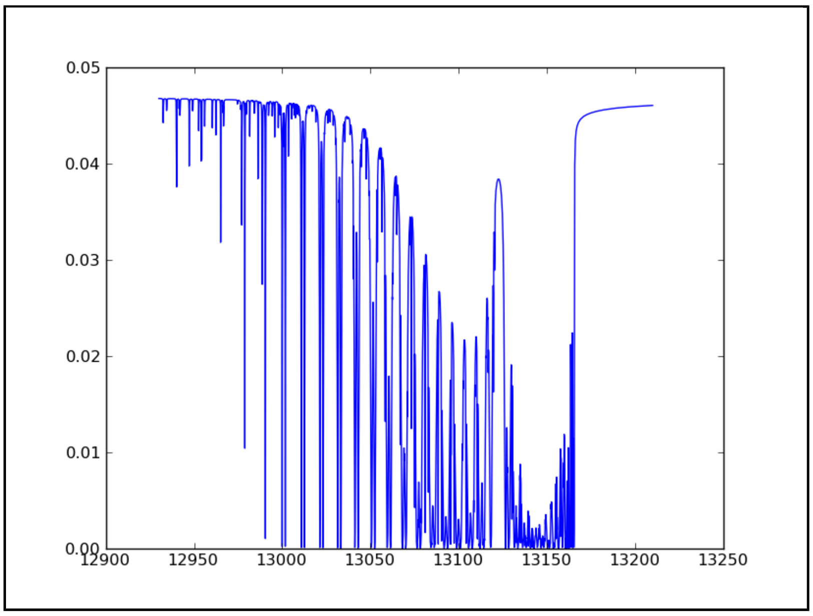
Run forward model and get radiance and Jacobian (takes longer to run)

```
r, jac = conf.forward_model().radiance_and_jacobian()
```

Example 4 - Radiative Transfer

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

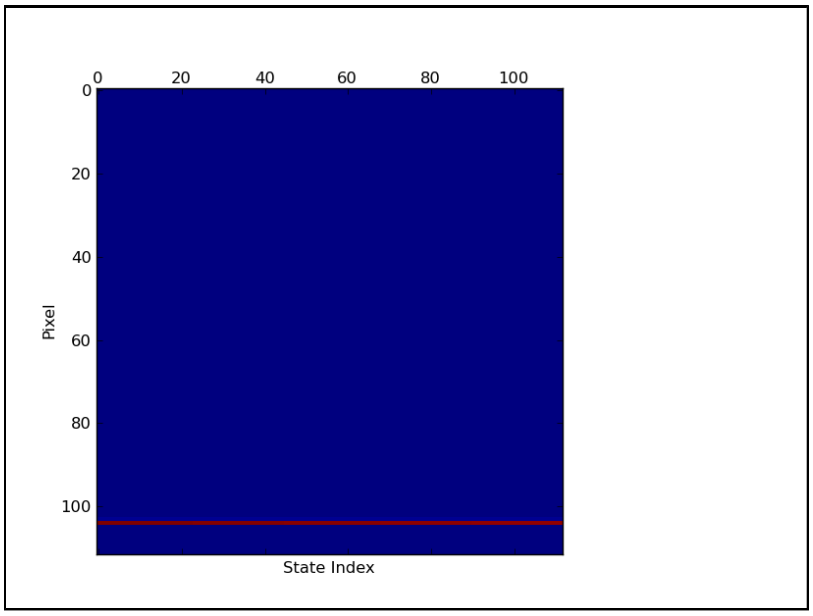
```
# Wavelengths we use for high resolution band 0. Can also supply own list
band = 0
wn = conf.spectrum_sampling().wave_numbers(band)
rrt = conf.radiative_transfer().radiance(wn, band)
plot(wn, rrt)
```



with Jacobian (takes long):

```
rrt, jacrt = conf.radiative_transfer().radiance_and_jacobian(wn, band)
matshow(abs(transpose(jacrt[0:112, :])), cmap=cm.jet)
pylab.xlabel("State Index")
pylab.ylabel("Pixel")
```

Shows not surprising result that Jacobian is dominated by one value:



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

```
print conf.state_vector().state_vector_name()[argmax(jacrt[0,:])]
```

Returns "Ground Lambertian Spectrally Dependent Spectral Channel 1 Parameter 1, 1"

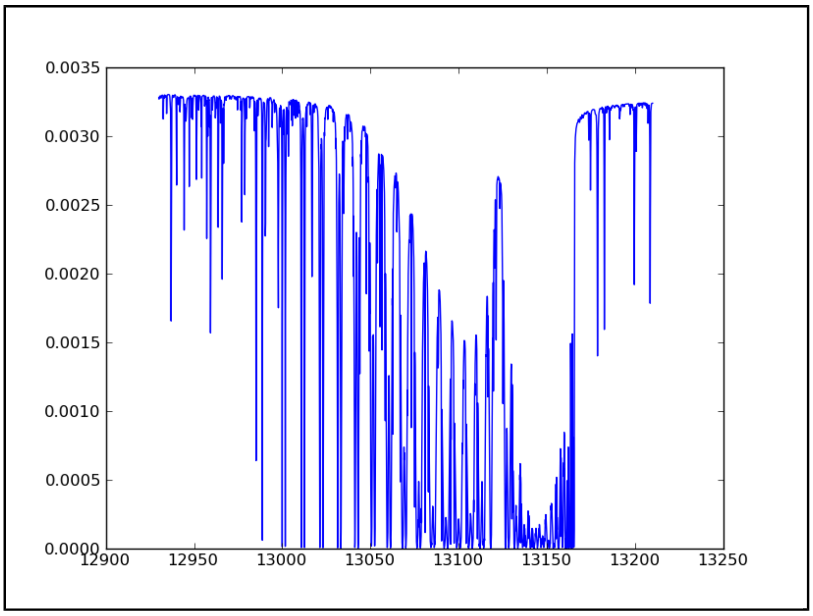
Example 5 - Apply Solar Model

Take results from last example and apply solar model. This is what goes into instrument

```
rsolar = conf.solar_model(band).apply_solar_model(wn, rrt)
plot(wn, rsolar)
```

with Jacobian:

```
rsolar, jacsolar = conf.solar_model(band).apply_solar_model(wn, rrt, jacrt)
```



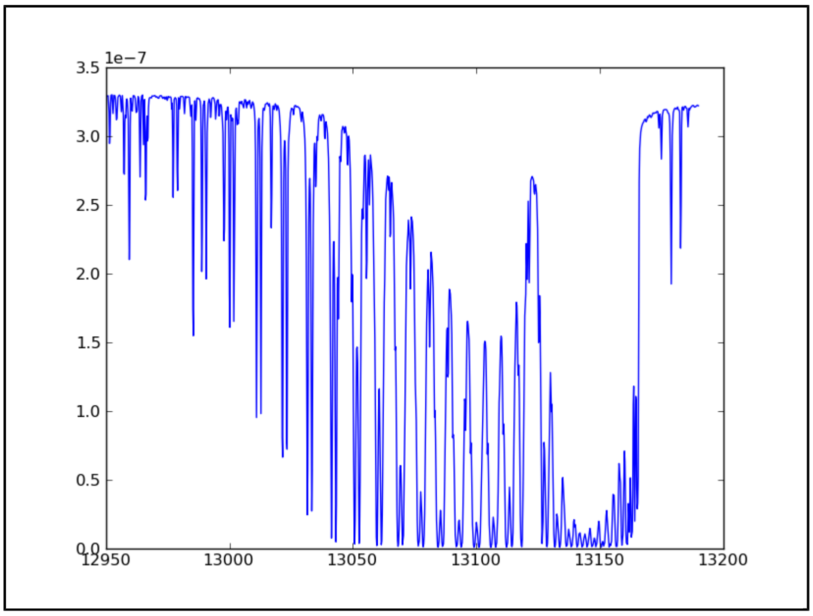
Example 6 - Apply Instrument Model

Take those results, and apply instrument model

```
# Get list of pixels that fall in spectral window
pix_list = conf.spectral_window_apply().pixel_list(band)
rinst = conf.instrument().apply_instrument_model(wn, rsolar, pix_list, band)
# Wavenumbers corresponding to pix_list. This includes dispersion
wninst = conf.instrument().pixel_wavenumber(band)[array(pix_list)]
plot(wninst, rinst)
```

with Jacobian:

```
rinst, jacinst = conf.instrument().apply_instrument_model(wn, rsolar, jacsolar, pix_list, band)
```



Example 7 - Use Generic Solver

Pyton scipy comes with some generic solvers, including one based on the standard Fortran minpack routines. We have a version the the cost function that uses the more standard format of embedding the a priori values, a priori covariance matrix, and radiance uncertainty (see Edwin's "THE LEVENBERG-MARQUARDT ALGORITHM FOR SOLVING THE NONLINEAR LEAST SQUARES PROBLEM" slides).

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.

More Advanced Example

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.

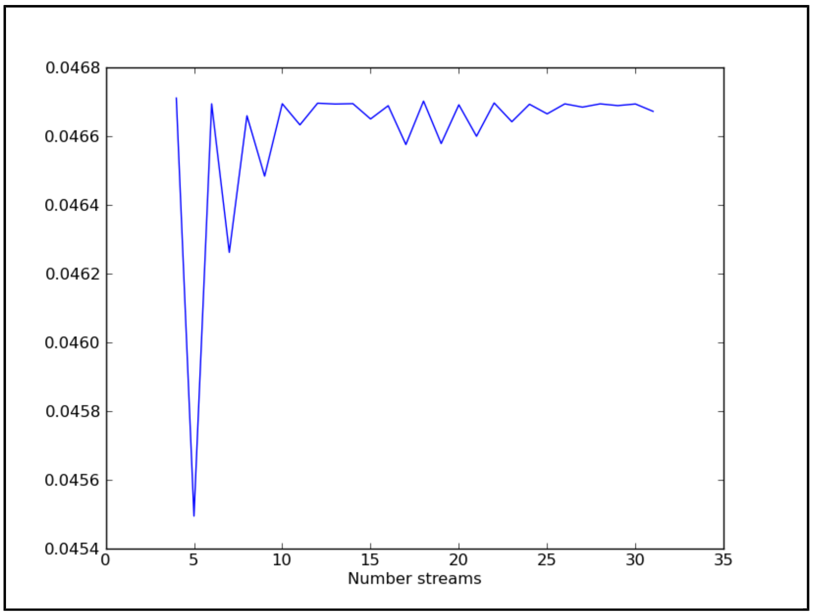
We create a new class "RtExtraKnobs". Save this in the file "rt*extra*knobs.py". Note while you are developing code like this, you can repeatedly load updated versions by using the ipython "%run" command (once it is complete, you can just import it like any other module).

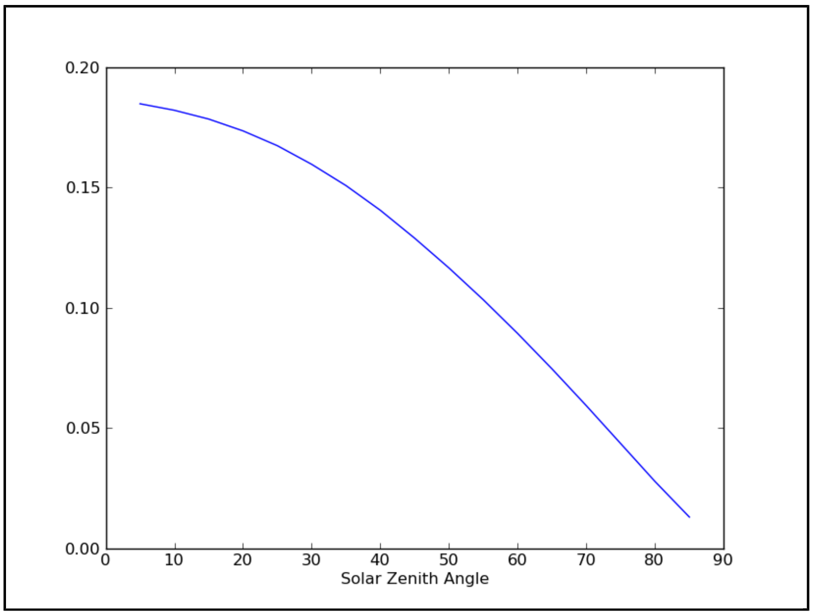
```
import full_physics as fp
class RtExtraKnobs:
  def __init__(self, fname="/groups/algorithm/python_tryout/sample_run/oco_l2.run"):
      conf = fp.ConfigurationHeritage(fname)
      self.conf = conf
      self.atm = conf.atmosphere()
      self.state_vector = conf.state_vector()
      self.level_1b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

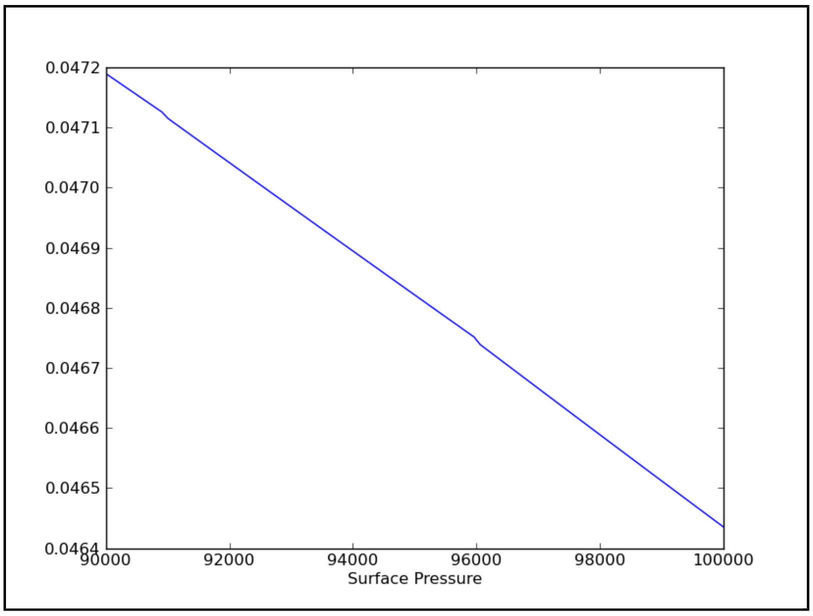
```
        # Hardcode these for this example
         nbrdf_quadratures = 50
         nstoke = 3
         ss_corr = True
         delta_m_scaling = True
         uplooking = False
        # Value needed to be true for LRad
         get_rad_dif = True
         nmom = number_stream if(number_stream >= 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()
# Value when we hold something constant
wn = 13005.0
sza = 74.0
psurf = 96716.0
nstream = 16
# Make versions of functions that only have one thing vary at a time
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")
```


0.0468
0.0466
0.0464
0.0462
0.0460
0.0458
0.0456
0.0454
0
5
10
15
20
25
30
35
Number streams


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



Python Callback

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

Testing

Unit Testing

There are two kinds of Level 2 Full Physics testing. The first is unit testing. This tests 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, etc.

The unit tests are fully automated. You can run them with the command:

```
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 "make check" should still be run aftet=r 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 any breakage to any other classes, running the full suite should still be done occasionally to make sure everything is working.

Unit Testing Variables

Should you be testing on an on SCF machine then you will need the ABSCO tables for certain unit tests.&nbsp; Once you have downloaded the ABSCO table files from a SCF machine you can specify the location of these files. The default absco directory is /groups/algorithm/l2_fp/absco/v3.1.0_alpha, but you can specify a different directory 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/v3.1.0_alpha
```

End-to-End Test

A build target "run_tests" exists such that when you use this, it 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 L2 FP checkout.

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

```
make run_tests
```

To test only the GOSAT data full run use the l2_fts_run target:

```
make l2_full_run
```

Or, to test only the FTS full run use the l2_fts_run target:

```
make fts_run
```

Bug Reporting

Web Interface

L2 Full Physics uses Trac for bug reporting and tracking. Our installation of Bugzilla is available at the following URL: https://svn.jpl.nasa.gov/trac/

Access to this system requires access to the JPL network and a Subversion user/password. Login into the program is done using the same username/password used to access the Subversion revision control system.

Runs Setup

Populator Usage

After following the details on the environment setup page you should be ready to run the L2 FP support software needed to set up retrieval inputs.

Inside the level_2/support package there is a tool known as "The Populator", which can generate the inputs needed by the L2 FP retrieval code. This software is located in the "level_2/support/populator" directory. The setup scripts mentioned in previous sections will export this path to your environment so that you can run this program from anywhere on the system.

Shared Run Directory

On the SCF computer system a directory shared by members of the "algorithm" group is the intended location for all runs. A directory with a descriptive name of the test or processing being performed should be created under this directory and used as the destination where runs are created and processed:

```
/groups/algorithm/l2_fp/runs/
```

It is a **bad** idea to save runs into your home directory because they are backed up automatically. Typically the output of L2 runs result in a large number of files that take up a large amount of disk space. The volume of files generated can severely cripple the backup software as it tries to do automatic backups.

Populator Data Sources

This tool can produce run directories for any of the following data sources:

- gosat - GOSAT flight data
- fts - TCCON FTS data
- oco - OCO downlooking data
- oco_uplooking - OCO uplooking data

At the minimum, for "gosat" or "oco" data sources the following files are needed:

- L1B HDF file (.hdf)
- Resampled ECMWF HDF file (.hdf)

Additionally, if available, cloud screening data can be setup to be supplied to the L2 FP code for use in assessing data retrieval quality.

The "fts" data source requires the following files

- Run log file (.grl)
- A and B spectra files (.XXX where XXX is a 3 digit number)
- ASCII atmosphere file, which can be converted from GFIT's .mav file (.dat)

Location of GOSAT Data

GOSAT data is available from the SCF file system at the following paths:

- /acos/product/Production/*/L1bXXXX/ -- L1B
- /acos/product/Production/*/EcmXXXX/ -- Resampled ECMWF
- /acos/product/Production/*/CldXXXX/ -- Cloud Screening

In the above XXXX represents a version number using only numerals. For instance version 2.8.00 data would be represented with the string 2800.

The * in the path above is a placeholder for the GOSAT calibration version number. NIES does not reprocess when changing calibration input data sets so historical observation will exist in different GOSAT version number directories.

Location of FTS Data

FTS data should be obtained from the TCCON group at Caltech.

Populator Config File Creation

If you have been delivered an aggregate directory from the production pipeline then you should already have a .config file and you may skip this section. Otherwise, you will need to create a .config file that allows the Populator to find the various files it needs as well as to identify the data source.

At a minimum the config creation tool needs the data source type name and the path to the files mentioned in the last section for each data source type. In the following examples a bogus path is used for illustration purposes only. The path can be relative or absolute.

Before creating a config file you must create a directory somewhere on the file system that will be the base directory for this run. It is suggested that you create subdirectory somewhere under the shared runs path:

/groups/algorithm/l2_fp/runs/

Once you have created your directory chdir to it. Any of the follow examples are performed from the directory that serves as the base of your runs. The config file will live in the base directory, sub-directories for log and output files are created later by the populator tool.

For example, to create a config file for a "gosat" data source you would run the following command from the "my_runs" base directory:_

```
my_runs]$ create_config.py -t gosat /path/to/acos_L1b_data.h5 /path/to/acos_Ecm_data.h5 /path/to/acos_Cld_data.h5
```

Once the command has finished executing you will have a new .config file with a name based upon the data source type and the current directory.

Optionally, you can specify a different config file name for output as well as filter the sounding ids used based upon and user supplied list of sounding ids.

For GOSAT data, the two polarization channels "S" and "P" are indicated by the appending of these letters to a sounding id. A pure numeral sounding id without either of these characters indicates that the S and P channels should be averaged.

To create a FTS config file you would use the "fts" data source and point to all the required files. For example:

```
my_runs]$ create_config.py -t fts /path/to/runlog.grl /path/to/spectra-a.001 /path/to/spectra-b.001 ... /path/to/
atmosphere_from_gfit.dat
```

Note in the above the ellipsis indicate that you would specify ALL the spectra files as inputs on the command line. Using shell globbing is acceptable as all the filenames in a directory will be expanded before the shell passes control to the program. Additionally, create_config.py will ignore any files it does not know anything about._

Help text describing available options are displayed by running the program with the following option:

```
my_runs]$ create_config.py --help
```

Run Directory Population

Once you have a config file created either manually or delivered from the production pipeline you are ready to use the Populator program.

At a minimum this program needs the name of a config file and optionally the path to a L2 FP binary. If a path to a L2 FP binary is not specified then helper scripts for running the created inputs will not be created.

For example to populate the run directories for a config file created for an "gosat" data source you would run the following command:

```
my_runs]$ populate.py -b /path/to/my/l2_fp gosat_my_runs.config
```

Once the program has stopped executing you will have the following directories and files:

- log -- Directory where log files for per sounding runs are written
- output -- Directory where HDF5 output files are written
- sdos_input_list.dat -- Log file used by SDOS aggregator containing versioning information as well as details on source files and directories used by the Populator
- sounding_id.list -- List of sounding ids to be processed. _
- l2_fp_job.sh, launch_jobs.sh -- When the "-b" option is given to populate.py this file these files are created to aid in running the software as explained in a later section.

Running Jobs

Running L2 FP Through Populator Produced Scripts

When the Populator is run with the -b option specifying a path to a L2 FP binary the following two scripts will be created:

- l2_fp_job.sh -- Runs an individual sounding
- launch_jobs.sh -- Launches all configured soundings by setting up a Torque job array running l2_fp_torque.sh for each sounding

You normally would not run l2_fp_job.sh directly, although it can be useful when debugging a single sounding. This script will take an argument specifying which sounding to process, however the argument is an index into the list of soundings as present in the input configuration file. Instead of using this script directly for debugging it is better to use it as example as to how the L2 code is run per sounding.

launch_jobs.sh is a simple wrapper around the Torque qsub command. It only makes sense to run this script on the head node machine, fullerene. Any arguments specified to launch_jobs.sh will be passed directly to qsub. On the SCF system the one required argument missing from the qsub call done in launch_jobs.sh is the argument specifying which queue to use. This argument is specified by the "-q" option and hence the normal use case for running launch_jobs.sh is as follows:

```
launch_jobs.sh -q long
```

The above command will launch all soundings into the long queue. Should you wish to use a different queue just swap out "long" for the desired queue name. Any additional options for qsub can also be specified in the call to the launch_jobs.sh script.

Additional Torque Documentation

For full documentation on TORQUE job submission, please consult this document:http://www.clusterresources.com/torquedocs21/2.1jobsubmission.shtml

For additional documentation on the TORQUE job submission command (qsub), please consult this document:http://www.clusterresources.com/torquedocs/commands/qsub.shtml

Interactive Jobs

Should you wish to run a program interactively you can use the following command:

```
~]$ qsub -I -q long -d ./
```

In the above command the "-I" option instructs qsub to launch and interactive job. The "-d" option specifies which directory we will be placed in once our interactive shell has been launched. And "-q" specifies which of the fullerene queues to use.

Checking Cluster Job Status

After jobs are running they will be executed as soon as resources are available. You can check the status of these jobs with various level of verbosity using the "qstat" command:

```
~]$ qstat
```

All running jobs will be listed using this command with a "R" in the next to last column indicating that job is currently running and a "Q" in that column meaning it is queued.

More information on using this command can be found from its manual page:

```
~]$ man qstat
```

Stopping Jobs

Jobs can be stopped before finishing normally by use of the 'qdel' command. First obtain the job id by querying the cluster through qstat:

```
~]$ qstat
Job id              Name          User          Time Use S Queue
------------------------ ---------------- -------------- -------- - -----
7960-1.fullerene      test.sh-1      youruser           0 R short
7960-2.fullerene      test.sh-2      youruser           0 Q short
```

Then specify the job id to qdel to stop the job:

```
~]$ qdel 7960
```

Ids reported by qstat with a hyphen after the job id are job array runs where multiple similar jobs are grouped together. To kill all of these jobs as in the example above you the common job id. To kill a specific job array index you would use the job id with the hypen:

```
~]$ qdel 7960-1
```

Post Run Analysis

After a set of L2 runs are finished processing scripts are available to aid in evaluating the results of the processing.

Quick Status Summary

The run_results.py and stats.py programs provide a way to quickly categorize the results of processing aggregate. The stats.py is a wrapper around run_results.py that has the additional functionality of outputting the sounding ids for the tracked categories into separate files. The run_results.py program only by default provides an overall summary printed to standard output. Note that stats.py will always generate a set of files and it is recommended that the o option be used to specify the location for these files to avoid cluttering your base aggregate directory. More information about the arguments available to both of these programs can be viewed by using the -help argument when running them.

The recommended practice is to run stats.py and specify a output directory named "results" for the generated files. The following example illustrates how to run the program:

```
#$ cd /path/to/my_runs
#$ stats.py -o results
```

When the program is finished counting it will display something similar to the following to screen as well as write it to the file output/run_results.txt:

```
 156 converged
  52 exceeded maximum iterations
     100 with good master quality
       8 with caution master quality
     100 with caution master quality
----
       2 completed coxmunk type
     206 completed lambertian type
----
 146 handled errors
====
 354 total runs
```

In the above output the left most column of numbers all sum to the total number of runs. The indented column of numbers are additional categorization counts of the numbers they are indented beneath. This second indented column of numbers includes the same runs counted in different ways and will not sum to the total number of runs.

SQL Summary

For more detailed analysis, you can produce a sqlite database file with detailed information about each run. This can be done using the tool "runs_database". You can then create reports by accessing the sqlite database. Use the "--help" with "runs_database" to see all the options.

Note that sqlite has the nice property of allowing different database files to look like different tables in which you can then do normal joins etc. with. This means we can generate a separate file for each version, but still be able to do cross comparisons between the different versions. So for example with B3.01 we could match up soundings with B2.09 to generate comparison statistics.

Note that the runs_database just puts the data in place, it does not currently create any views or index. For performance reasons, you may want to create index for some columns (e.g., sounding_id). Likewise, it can be useful to create views. For example:

```
create index index1 on runs (sounding);
create index index2 on runs (sounding, handled_error, execution_error);
create view good_runs as select * from runs where handled_error='f' and execution_error='f';
```

Aggregating Output Files

When the L2 FP soundings have completed processing there will be one HDF5 file per sounding. Soundings that complete without an error will be named l2_<sounding_id>.h5, while soundings exiting due to an error will be named l2_<sounding_id>.h5.error.

Included in the level_2/support directory is a program that can be used to aggregate the individual output files into single HDF5 file, making it easier to investigate the retrieval results.

The simplest usage of this program is reflected in the following example run from the base directory of the L2 FP runs:

```
#$ splice_acos_hdf_files.py --aggregate -o output/l2_aggregated.h5 output/*.h5
```

The --aggregate options instructs the tool to splice together all soundings available in the supplied input files. To supply the list of sounding ids to splice together, say for instance to aggregate only converged soundings, you would run the following example using one of the output files of the stats.py program:

```
#$ splice_acos_hdf_files.py -s results/converged.list -o output/l2_converged.h5 output/*.h5
```

Analysis Environment

The analyze_l2.py program allows a user to load one or more spliced L2 single sounding output files (created with splice_acos_hdf_files.py) or aggregated L2f files into an interactive environment which provides predefined plot and informational routines. The motivation for the creation of this program was the provide a tool that makes it easier to define routines for data analysis where the user focuses on what data they want and how to manipulate it. This tool leaves the order of analysis open ended and does no prescribe any set order of investigation by providing an interactive environment. When a user does want some level of automation they can take their experimentation in the interactive environment and turn it into a script to be run from the command line.

The main features of the tool are:

- Loads a customized IPython shell which includes many predefined analysis routines
- Correlates soundings present in all files passed to tool so that analysis routines have consistent inputs
- Handles extraction of data for correlated soundings and delivers to analysis routines
- Allows the definition of analysis routines by simply defining a function where the arguments define the datasets to be extracted files loaded into the environment

Requirements

The tool requires the following Python packages.

- Those mentioned in the Python Requirements section of the Linux System Setup section.
- IPython

Launching

At a minimum the program requires at least one or more spliced L2 output files to be passed as a command line argument. After calling the program the user is presented with a list of available routines grouped by common behavior as well as an IPython prompt.

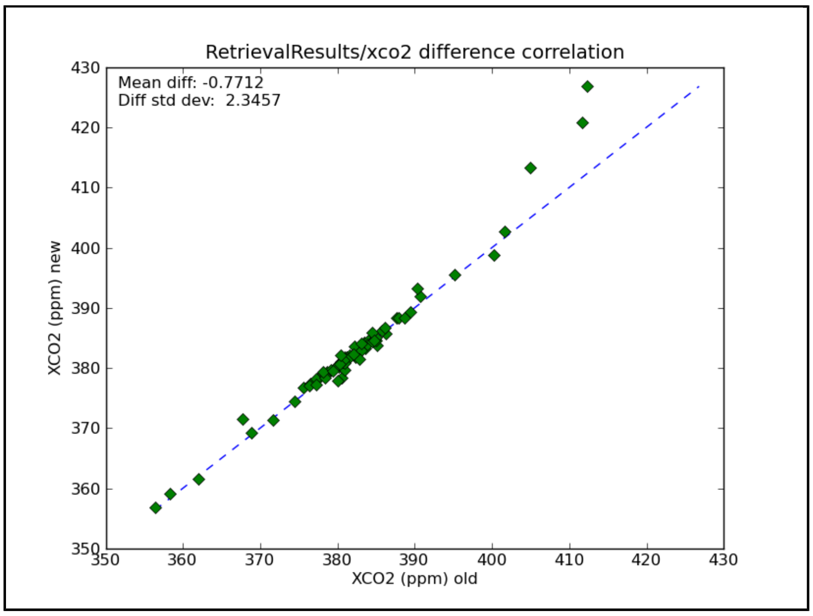
```
#$ analyze_l2.py old.h5 new.h5
*** Launcing L2 Analysis Shell ***
Available analysis routines:
filter_outcome
filter_surface_type
master_quality
outcome
plot_abo2_chi2_diff_hist, plot_abo2_rms_diff_time, plot_abo2_chi2_diff_corr,
  plot_abo2_rms_time, plot_abo2_rad_mean_diff_hist, plot_abo2_rad_mean_uncert_diff_time,
  plot_abo2_rad_mean_diff_time, plot_abo2_rad_mean_uncert_diff_corr,
  plot_abo2_chi2_diff_time, plot_abo2_rms_diff_corr, plot_abo2_rad_mean_time,
  plot_abo2_albedo_diff_time, plot_abo2_rms_diff_hist, plot_abo2_albedo_time,
  plot_abo2_chi2_time, plot_abo2_albedo_diff_hist, plot_abo2_albedo_diff_corr,
  plot_abo2_rad_mean_uncert_time, plot_abo2_rad_mean_uncert_diff_hist,
  plot_abo2_rad_mean_diff_corr
plot_div_histogram
plot_iter_histogram
plot_psurf_diff_time, plot_psurf_time, plot_psurf_diff_corr, plot_psurf_diff_hist
plot_sco2_albedo_diff_hist, plot_sco2_albedo_time, plot_sco2_chi2_diff_hist,
  plot_sco2_rms_diff_time, plot_sco2_chi2_time, plot_sco2_chi2_diff_corr,
  plot_sco2_rad_mean_diff_hist, plot_sco2_chi2_diff_time, plot_sco2_rms_diff_corr,
  plot_sco2_rad_mean_uncert_diff_hist, plot_sco2_rad_mean_diff_corr,
  plot_sco2_rad_mean_uncert_time, plot_sco2_rms_time,
  plot_sco2_rad_mean_uncert_diff_corr, plot_sco2_albedo_diff_corr,
  plot_sco2_rad_mean_diff_time, plot_sco2_albedo_diff_time,
  plot_sco2_rad_mean_uncert_diff_time, plot_sco2_rad_mean_time, plot_sco2_rms_diff_hist
plot_statevector_diff_trend
plot_wco2_chi2_time, plot_wco2_rad_mean_time, plot_wco2_rad_mean_diff_corr,
  plot_wco2_rad_mean_uncert_diff_corr, plot_wco2_rad_mean_uncert_time,
  plot_wco2_rms_time, plot_wco2_chi2_diff_time, plot_wco2_rms_diff_hist,
  plot_wco2_albedo_diff_time, plot_wco2_chi2_diff_corr,
  plot_wco2_rad_mean_uncert_diff_hist, plot_wco2_rms_diff_time,
  plot_wco2_albedo_diff_hist, plot_wco2_chi2_diff_hist, plot_wco2_albedo_diff_corr,
  plot_wco2_rad_mean_uncert_diff_time, plot_wco2_rms_diff_corr, plot_wco2_albedo_time,
  plot_wco2_rad_mean_diff_hist, plot_wco2_rad_mean_diff_time
plot_windspeed_diff_hist, plot_windspeed_time, plot_windspeed_diff_corr,
  plot_windspeed_diff_time
plot_xco2_time, plot_xco2_diff_time, plot_xco2_diff_corr, plot_xco2_diff_hist
surface_type
L2A: In <2>:
```

The prompt has the prefix "L2A" in it to differentiate it from a default IPython shell. The routine names listed are callable from the command line without requiring any arguments. The tool handles extracting the data from the supplied files for only the correlated soundings and passing this data to the routines that perform the analysis behaviour.

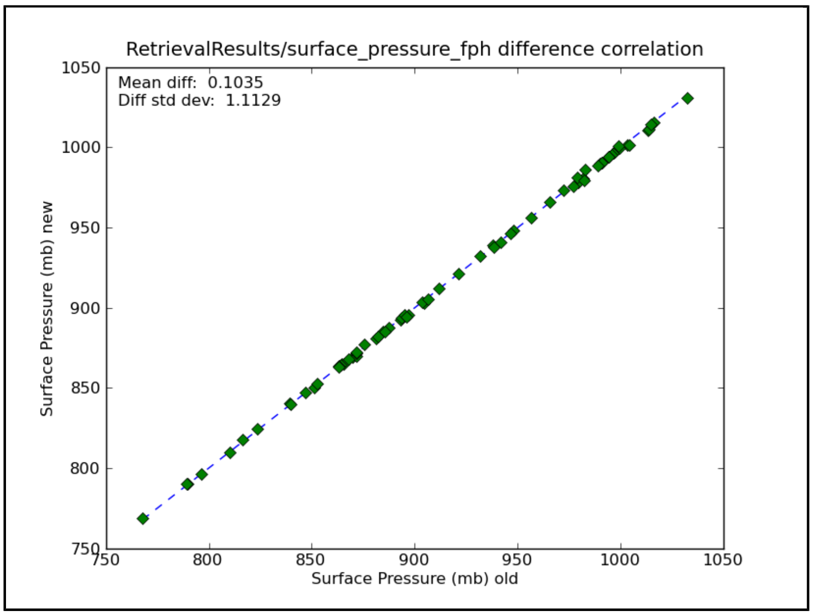
Plotting Examples

The following plotting examples are presented with the command used from the L2 Analysis tool prompt and their resulting plot using two example input files.

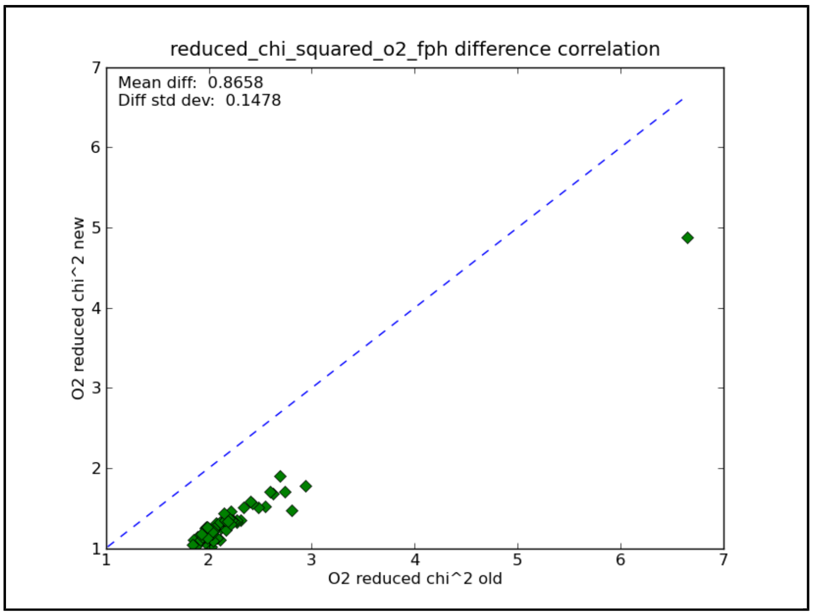
L2A: In <2>: plot_xco2_diff_corr()



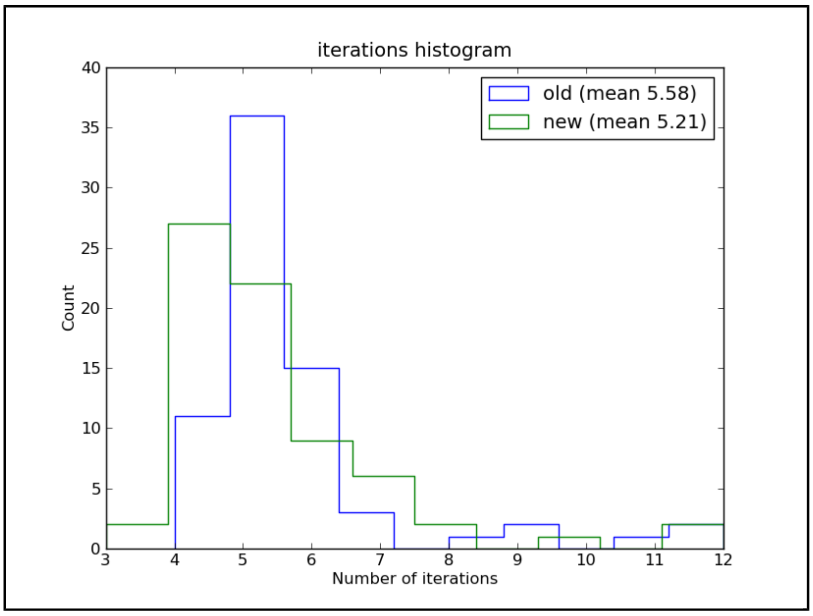
L2A: In <3>: plot_psurf_diff_corr()



L2A: In <4>: plot_abo2_chi2_diff_corr()



```
L2A: In <5>: plot_iter_histogram()
```



Informational Examples

The tool is not limited to only creating plots. There are also routines defined that return informational analysis on the files loaded into the environment.

```
L2A: In <2>: master_quality()
L2A: Out<2>: {'new': {'Bad': 50, 'Caution': 5, 'Good': 16}, 'old': {'Bad': 71}}
L2A: In <3>: surface_type()
L2A: Out<3>: {'new': {'Lambertian': 71}, 'old': {'Lambertian': 71}}
```

Getting Help

If an analysis routine defines documentation it can be queried from the command line as follows:

```
L2A: In <2>: plot_xco2_time?
Type:         function
Base Class:    <type 'function'>
String Form:    <function <lambda> at 0xea056e0>
Namespace:     Interactive
File:         Dynamically generated function. No source code available.
```

```
Definition:    plot_xco2_time(**kwargs)
Docstring:
    Plots multiple sets dataset values (multiple files) on the y-axis versus the
    sounding time (sounding id) on the x-axis
```

Filtering Data Sent to Routines

By default the sounding ids present in all files are used for the extraction of data passed to analysis routines. However, a keyword argument can be passed to any routine to use a different set of soundings. There are several predefined filter routines which are simply analysis routines that just return a list of sounding ids that can then be fed to other routines. Filter can be understood through examining this example:

```
L2A: In <2>: surface_type()
L2A: Out<2>: {'example': {'Coxmunk,Lambertian': 63, 'Lambertian': 71}}
L2A: In <3>: filter_surface_type(by='^Lambertian')
L2A: Out<3>:
((20090627210055,
 20090627210247,
 20090627210309,
 # ....... truncated for clarity .......
 20090627211734,
 20090627211738,
 20090627211752),)
L2A: In <4>: surface_type(ids=filter_surface_type(by='^Lambertian'))
L2A: Out<4>: {'example': {'Lambertian': 71}}
```

Note that in the example above, the value passed through the "by" keyword argument is a regular expression.

Naming Objects for Plots

By default the program will base the names passed to plotting routines based on filenames. It tries to parse out the common parts of the filenames and paths and leave the unique parts as the object names. But, this mechanism does not always work adequately. Explicit names can be supplied on the command line using the "-n" argument, specified once per filename as seen in the following example:

```
analyze_l2.py expected/l2_rrv_output.h5 output/l2_rrv_output.h5 -n old -n new
```

In the analysis environment you can query what the names are from the analysis_env object as follows:

```
L2A: In <2>: analysis_env.obj_names
L2A: Out<2>: ['old', 'new']
```

Missing Soundings

For the curious, the analysis_env object saves which sounding ids are present in each file but are missing from any of the other files. Querying the class's attribute as follows the user will see a list of ids per file:

```
L2A: In <2>: analysis_env.missing_ids
L2A: Out<2>: ((), (20090627210318,))
```

Adding Analysis Routines

Analysis routines can either be added to the l2_analysis module directory in level_2/support/python_lib or dynamically from within the interactive environment from a separate file.

Routine Argument Names

The tools determines which data to extract from the input files based on the name of the arguments in the signature of the analysis routine. The argument names can be either partial or full data set names. In the case of full data set names the "/" character from the HDF5 nomenclature is replaced by "__" since "/" is not a valid identifier in Python. Partial names can only be used if they match uniquely a data set in the input files.

For arguments named to match data sets the value of the argument will be a list of Numpy array objects. There will be one array for each file containing only the data matching the correlated sounding list in the sounding dimension. The data will be in the same order as the files were given on the command line.

Additionally any of the attributes of the analysis_env object itself can be passed to a routine if it is named as an argument. The most useful of which is the "obj_names" attribute which contains the names automatically determined or explicitly passed from the command line per each file.

Simple Example Routine

A very simple example that will over plot the XCO2 for each file is given below:

```
from matplotlib.pyplot import *
def plot_xco2(obj_names, RetrievalResults__xco2, **kwargs):
   ax = self.new_axis(**kwargs)
  for xco2 in RetrievalResults__xco2:
      ax.plot(xco2*1e6, '*')
   ax.set_title("example plot")
   ax.set_ylabel("XCO2 (ppm)")
   ax.legend(obj_names)
  return ax
```

In the argument list above "obj_names" is an analyis_env object attribute and RetrievalResults__xco2 refers to the RetrievalResults/xco2 data set. In the above the figure object is returned and is a good practice when creating new routines. This object can be used on the command line to perform additional operations on the plot such as saving it as a new page in a PDF file.

This routine can either be added to the l2_analysis module directory or defined in a separate script which is run from the interactive command line. In the case of using a separate file one would define the routine as above and additionally add the following line to tell the analysis_env object about the routine:

```
analysis_env.add_routines(plot_xco2)
```

This script can then be run from the command prompt as follows:

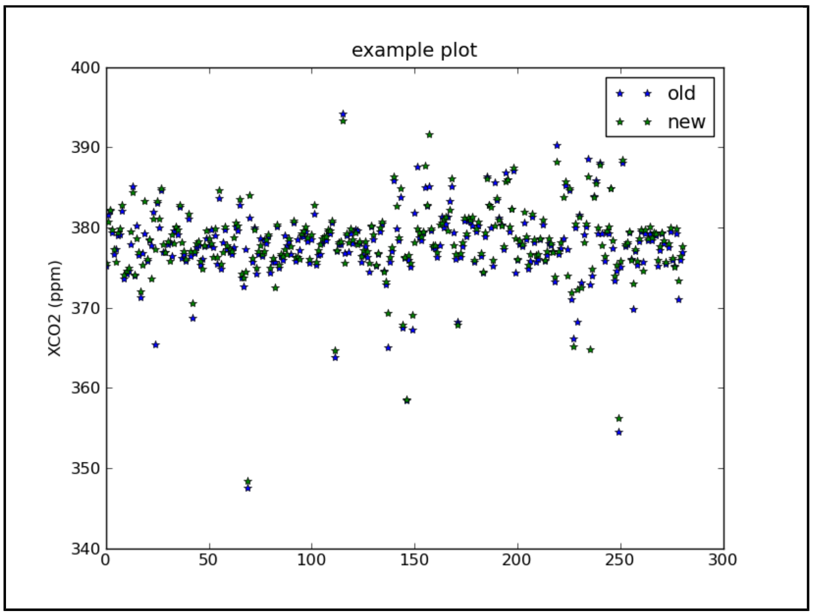
```
L2A: In <2>: %run -i example_script.py
```

Note that you must use the "-i" argument to %run in order for the interactive sessions' environment to be made available to the script.

The program will automatically create a new helper function whenever routines are added. The newly added routine can be called as follows and the proper data will be passed to it:

```
L2A: In <2>: plot_xco2()
```

Calling the routine we defined earlier will result in a plot similar to this one:



Running the script successively after making changes to the file will install the new version in the environment. This enables more rapid development of routines than placing them in l2_analysis module directory where changes are only available after quitting and restarting the analysis program. Note that if you change the name of our routines then reload, the old version will still be present under the old name.

Scripting

Any script that can be run from the interactive environment via "%run -i" can be run from the command line using the "-s" argument. The script has the same routines and objects available as would be present from an interactive shell. For example one could use a script such as the following to make a standard set of plots and save them into a PDF:

```
from matplotlib.backends.backend_pdf import PdfPages
from matplotlib import pyplot
import matplotlib.axes
pdf_filename = "plots_%s-%s.pdf" % (analysis_env.obj_names[0], '_'.join(analysis_env.obj_names[1:]))
print "Creating plot file: %s" % pdf_filename
pdf = PdfPages(pdf_filename)
def save_res(res):
  if hasattr(res, "__iter__"):
     for item in res:
         save_res(item)
  else:
     if isinstance(res, matplotlib.axes.Axes):
         res = res.get_figure()
      pdf.savefig(res)
      pyplot.close(res)
  return res
save_res( plot_xco2_diff_corr() )
save_res( plot_psurf_diff_corr() )
save_res( plot_iter_histogram() )
save_res( plot_div_histogram() )
save_res( plot_abo2_chi2_diff_corr() )
save_res( plot_wco2_chi2_diff_corr() )
save_res( plot_sco2_chi2_diff_corr() )
save_res( plot_abo2_rad_mean_uncert_diff_corr() )
save_res( plot_wco2_rad_mean_uncert_diff_corr() )
save_res( plot_sco2_rad_mean_uncert_diff_corr() )
pdf.close()
```

This script could then be run from the command line as follows:

```
#$ analyze_l2.py old.h5 new.h5 -s example_script.py
```

External System Setup

Linux System Setup

The following instructions are intended for those attempting to deploy the L2 development environment on a non ACOS/OCO SCF Linux system.

Memory Requirements

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

Python Requirements

The level_2/support directory requires at least Python version 2.7 (but not 3.0 yet). If you Linux distribution is fairly recent then you most likely have a fairly recent version of Python.

There are additional Python packages that some L2 programs require. Most likely your Linux distribution already has packages available using it's respective package management tool. Otherwise, follow the following links for details on the following necessary Python packages:

- 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

Fortran Compilers

One of the following Fortran compilers is required for compiling:

- GNU Fortran 4.4
- Intel Fortran Compiler 11.1 Free non-commercial license available

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 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 Environment Setup steps.

Compile according to L2 Full Physics Compilation instructions for a Non JPL Build

```
mymachine% mkdir build
mymachine% cd build
mymachine% /your/path/to/level_2/configure THIRDPARTY=build
mymachine% make all
```

Fedora/CentOS Deployment Steps

Under CentOS you will need to update to a newer version of gcc/gfortran:

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

Fedora has different package names for GCC 4.4

```
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:

```
wget http://production.cf.rubygems.org/rubygems/rubygems-1.3.5.tgz
tar zfvx rubygems-1.3.5.tgz
cd rubygems-1.3.5
ruby setup.rb
```

Note that 1.3.5 is not the latest version of Ruby Gems but is the latest one that works with CentOS's Ruby version 1.8.5.

For CentOS also install:

```
yum install ruby-rdoc
```

Use rubygems to install narray:

```
gem install narray
```

Follow Environment Setup steps.

Compile according to L2 Full Physics Compilation instructions for a Non JPL Build.

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
```

Install gfortran if not already installed:

```
apt-get install gfortran
```

Most other distributions provide gfortran along with gcc, but Ubuntu keeps it in separate packages.

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 Environment Setup steps.

Compile according to L2 Full Physics Compilation instructions for a Non JPL Build.

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 Environment Setup steps.

Compile according to L2 Full Physics Compilation instructions for a Non JPL Build.

OS X System Setup

The following instructions are intended for those attempting to deploy the L2 development environment on a Mac OS X system.

Select MacPort or Fink

Additional tools are needed on the Mac to build the L2 full physics. 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.

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/gcc44/libgfortran.dylib" \
   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-4.4 \
    CC=/opt/local/bin/gcc-mp-4.4 F77=/opt/local/bin/gfortran-mp-4.4 --enable-debug --enable-maintainer-mode
```

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.

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.

Deprecated

Deprecated sections are not included in the main user's guide, instead they are in a separate document.

Other Documentation

Doxygen

For developers, there is L2 Doxygen Documentation. This is automatically generated from the Subversion trunk of the software.

Python

There is L2 Python Documentation. This is automatically generated from the Subversion trunk of the software.

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