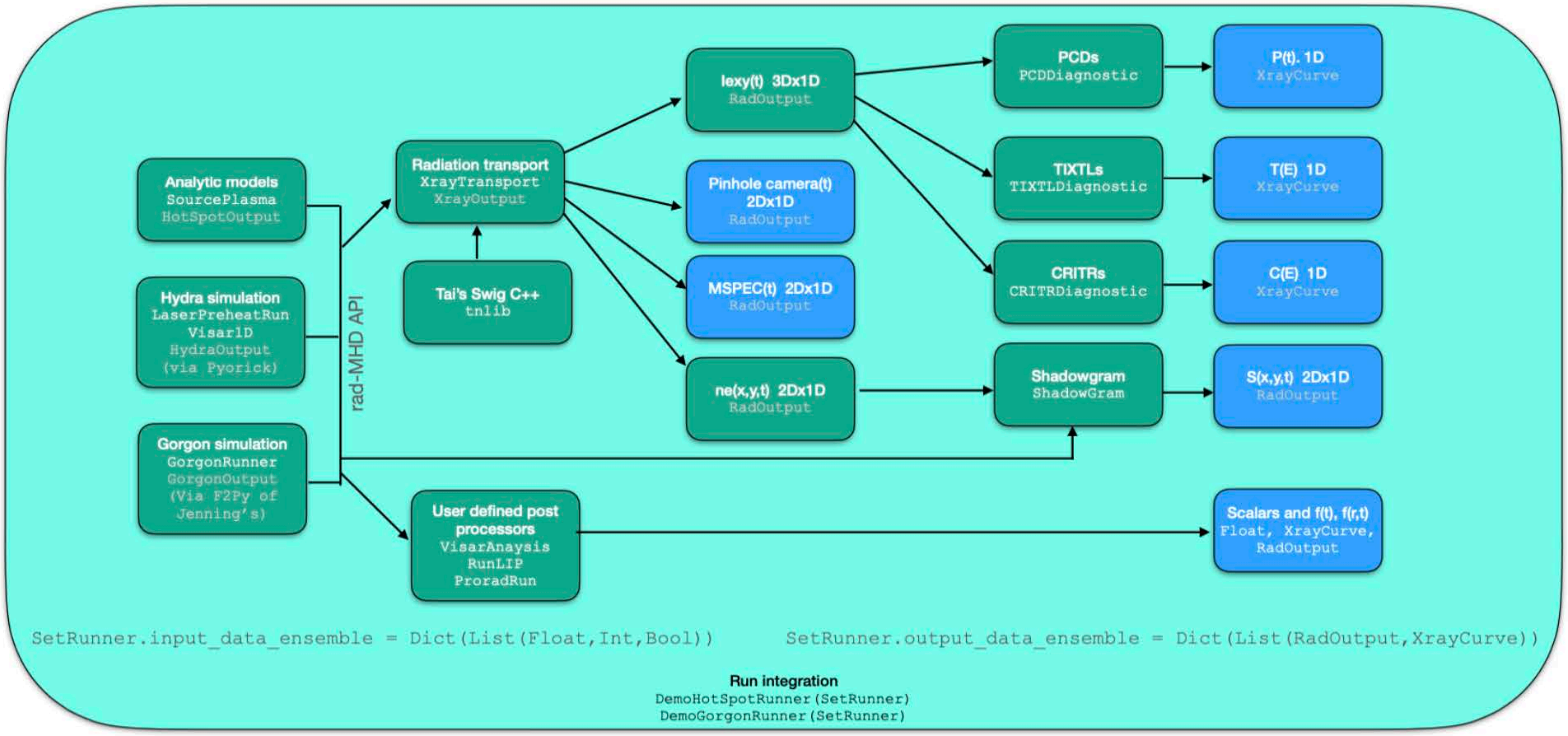
The workflow



All classes inherit `AppRunner(HasTraits)` class giving:
(1) Saved meta data state
(2) GUI editor
(3) CLI
(4) `TraitsUI` (type checking, initial value, label, help)
(5) `__str__()` method

This is the flow diagram of the workflow. Classes that implement the components of the workflow, and of the self described data (axis, labels, units, description, etc.). Put in links to Sphnix docs.

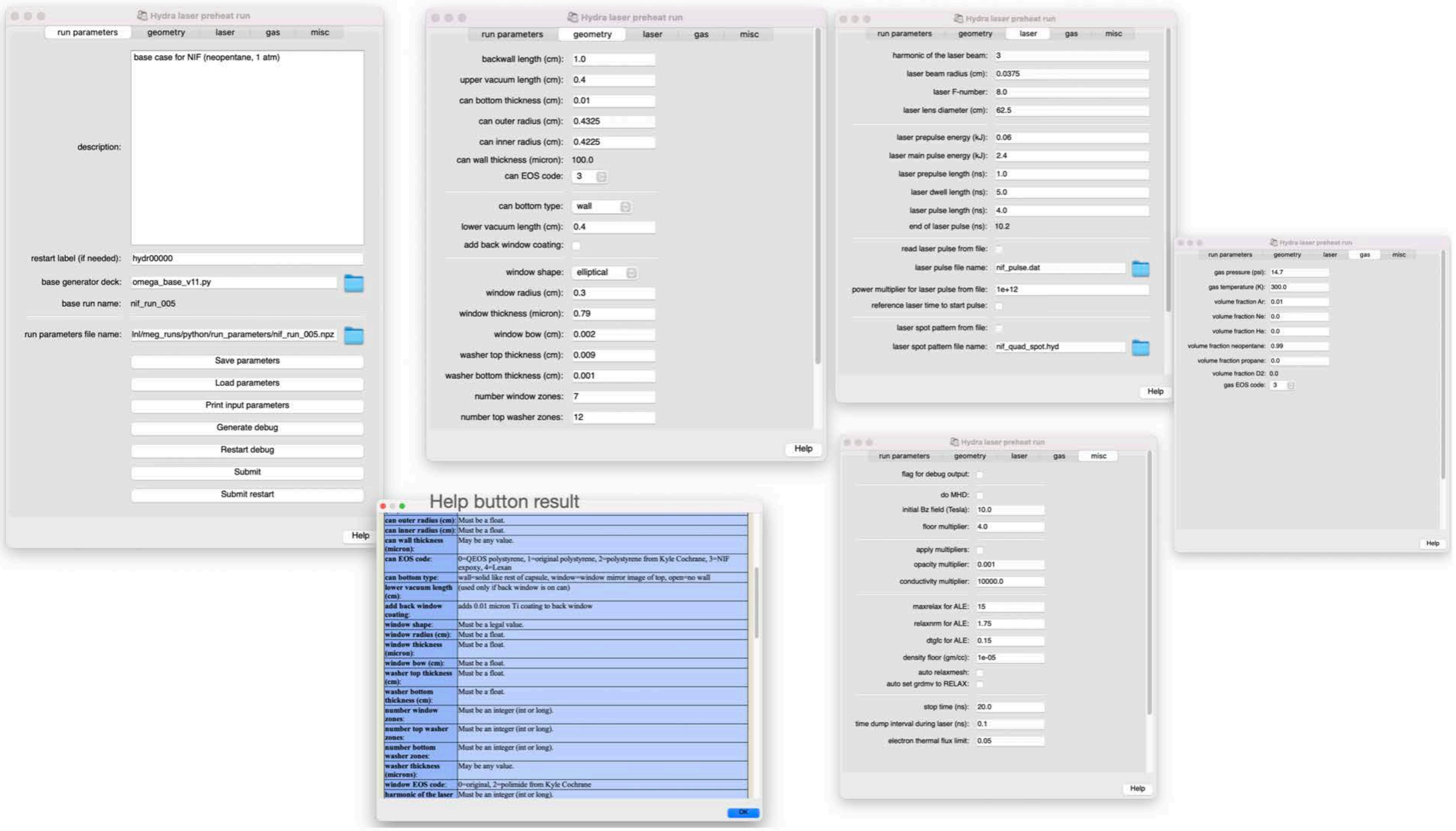
General description of the software architecture

What has systematically been constructed over since I arrived at Sandia in 2016 is realization of a software architecure (maintained from the start) in Git and documented using the Sphinx documentation system (http://s1008570.srn.sandia.gov:8080, using Markdown syntax in the source code to facilitate this). It was developed using concepts of object oriented design in `Python`. Classes were most times, subclasses of `HasTraits`. This allowed for strict variable typing & error checking (dramatically reducing parameter input errors), intial value, variables labels, and variable help. In addition to setting the variables and objects from a CLI with error checking, `TraitsUI` provides an interactive GUI editor view of the object. This was enhanced by the creation of the `AppRunner` class that supplemented `Traits` and `TraitsUI` to give:

1. saved and restored meta data state, so that data formats would be self descriptive and templates of application parameters can be retained
2. app type and app version with an version conversion facility to maintain backward version compatibility
3. provision of variables such as `run_name`, `input_directory`, and `output_directory` that are almost always needed, along with methods such as `run()`, `save()`, `load()`, `save_results()`, `load_results()`, `plot()`, `default_output_file()`, and `save_figs()`.
4. a better default GUI editor view, with fine grain control of Traited parameters that are displayed and common methods displayed as buttons.
5. a better default `__str__()` method with more fine grain control of what is printed.

The `AppRunner` class is not only used for applications but also for data objects. It is used to run Hydra and Gorgon in the following way. A generic Hydra run deck is constucted that is controlled by a set of variables. A set of parameters is imported into this run deck, first thing, from the saved state of an `AppRunner` class constructed with these parameters as edited properties (from both the CLI and GUI editor). In fact there is a template class `HydraRun` that supplements the `AppRunner` class with common methods such as `srun()`, `submit()`, `archive()`, and `restore()`, along with the infrastrure to inject the parameters into the `Hydra` deck. A similiar method can now be implemented to run `Gorgon` since Chris Jennings has refactored the code to give it an input deck.

A general API was constucted for rad-hydro simulation results (for `Gorgon`, `Hydra`, and a more general implementation for analytic construction). The interface grid format is based off of `tvtk`, a `Traited` form of `VTK`. A class was constructed, `XrayTransport`, that uses this API for its input, then performs the radiation transport using the methods developed by Tai in C++ and made available using `Swig` to `Python`. This code also directly calculates MSPEC and Pinhole imaging dianostics. The output also has been piped into subsequent applications that calculate PCD, TIXTL, and CRITR synthetic diagnostics. Many other diagnostics are anticipated to be developed in the near future. Where appropriate `C`, `C++`, and `FORTRAN`, such as for the re-gridder of block structured grids (which was done in `C`. An example of the GUI editor for the `Hydra` laser preheat run follows:



description: base case for NIF (neopentane, 1 atm)
restart label (if needed): hydr00000
base generator deck: omega_base_v11.py
base run name: nif_run_005
run parameters file name: lnl/meg_runs/python/run_parameters/nif_run_005.npz

backwall length (cm): 1.0
upper vacuum length (cm): 0.4
can bottom thickness (cm): 0.01
can outer radius (cm): 0.4325
can inner radius (cm): 0.4225
can wall thickness (micron): 100.0
can EOS code: 3
can bottom type: wall
lower vacuum length (cm): 0.4
add back window coating:
window shape: elliptical
window radius (cm): 0.3
window thickness (micron): 0.79
window bow (cm): 0.002
washer top thickness (cm): 0.009
washer bottom thickness (cm): 0.001
number window zones: 7
number top washer zones: 12

harmonic of the laser beam: 3
laser beam radius (cm): 0.0375
laser F-number: 8.0
laser lens diameter (cm): 62.5
laser prepulse energy (kJ): 0.06
laser main pulse energy (kJ): 2.4
laser prepulse length (ns): 1.0
laser dwell length (ns): 5.0
laser pulse length (ns): 4.0
end of laser pulse (ns): 10.2
read laser pulse from file:
laser pulse file name: nif_pulse.dat
power multiplier for laser pulse from file: 1e+12
reference laser time to start pulse:
laser spot pattern from file:
laser spot pattern file name: nif_quad_spot.hyd

gas pressure (psi): 14.7
gas temperature (K): 300.0
volume fraction Ar: 0.01
volume fraction Ne: 0.0
volume fraction He: 0.0
volume fraction neopentane: 0.99
volume fraction propane: 0.0
volume fraction D2: 0.0
gas EOS code: 3

flag for debug output:
do MHD:
initial Bz field (Tesla): 10.0
floor multiplier: 4.0
apply multipliers:
opacity multiplier: 0.001
conductivity multiplier: 10000.0
maxrelax for ALE: 15
relaxnrm for ALE: 1.75
dtglc for ALE: 0.15
density floor (gm/cc): 1e-05
auto relaxmesh:
auto set grdmv to RELAX:
stop time (ns): 20.0
time dump interval during laser (ns): 0.1
electron thermal flux limit: 0.05

can outer radius (cm):	Must be a float.
can inner radius (cm):	Must be a float.
can wall thickness (micron):	May be any value.
can EOS code:	0=QEOS polystyrene, 1=original polystyrene, 2=polystyrene from Kyle Cochrane, 3=NIF expoxy, 4=Lexan
can bottom type:	wall=solid like rest of capsule, window=window mirror image of top, open=no wall
lower vacuum length (cm):	(used only if back window is on can)
add back window coating:	adds 0.01 micron Ti coating to back window
window shape:	Must be a legal value.
window radius (cm):	Must be a float.
window thickness (micron):	Must be a float.
window bow (cm):	Must be a float.
washer top thickness (cm):	Must be a float.
washer bottom thickness (cm):	Must be a float.
number window zones:	Must be an integer (int or long).
number top washer zones:	Must be an integer (int or long).
number bottom washer zones:	Must be an integer (int or long).
washer thickness (microns):	May be any value.
window EOS code:	0=original, 2=polimide from Kyle Cochrane
harmonic of the laser	Must be an integer (int or long).

Many of the applications are parallized asynchronously, through MPI or Posix subprocess forks, using the `bsync` package developed by John Field for the LLNL data science effort.

The data classes have many common mathematical and utility methods defined such as: `plot()`, `convolve()`, `integrate()`, `__mult__()`, `__add__()`, and `__str__()`.

The ultimate goal of this infrastucture is embodied in the `SetRunner` class. This class captures the complete workflow. It also has the ensemble construct, `Distribution` objects, where distributions can be given for a set of input parameters. Ensembles can then be generated from these distributions, and the workflow executed for each member of the ensemble. The sythetic diagnostics for each member of the ensemble are then available for analysis using statistical and MLDL methods.

This infrasture has already been used to generate movies of GXD and MSPEC images for MagLIF preheat Hydra simultions, and to generate stochastic ensembles of 1D VISAR measurements for statistical and MLDL applications. It is also an integral part of critical deliverables for a REHEDS LDRD on applications of MLDL for Z, and a CIS LDRD on the same subject.

The instructions on how to set up the environment can be found in `pysnl/docs/build_soe.txt`. It is highly recommende that this infrastructure be executed via a Jupyter notebook served from a Docker image set up using `build_soe.txt`.