pyAeroOpt

DRAFT OF THE MANUAL

Introduction

pyAeroOpt[1] is a Python library that helps to quickly and cleanly build Python programs that can script calls to FRG codes. It provides interfaces with all of the FRG codes including Aero-F, Aero-S, sower, matcher, partnmesh, Sdesign, and Blender, and is easily extensible to any other command-line based code as well. It's primary use-case is in the scripting of these codes -- that is, cases where many calls to these codes must be made in sequence, and especially where some of these calls must be parameterized (e.g. the input files must be constructed as functions of a vector of values provided by an optimizer).

pyAeroOpt is particularly well-suited to scripting FRG codes for optimization. It provides access to several well-regarded optimization packages and an interface for cleanly scripting FRG codes to evaluate objective functions and constraints.

The manual is structured as follows. First, instructions for obtaining the code are provided. Then, an example problem (including Python code) is given to show how the provided classes are used to solve an optimization problem. Lastly, the features of pyAeroOpt are listed in more detail, to act as a programmer's reference when using or potentially extending pyAeroOpt.

[1] pyAeroOpt was originally created by Matt Zahr and has since been extended by other FRG members.

Table of Contents

Obtaining, Installing, and Running pyAeroOpt

pyAeroOpt is a Python package that must be run in a specific Python environment (it must be run in Python 3.4 with several important dependencies installed). To get started with pyAeroOpt on Independence, do the following:

1. Get the pyAeroOpt source by cloning from https://bitbucket.org/spenseranderson/pyaeroopt

```
hg clone https://bitbucket.org/spenseranderson/pyaeroopt
```

2. Download the latest Python3 version of Anaconda from https://www.anaconda.com/download. Anaconda is a Python distribution that comes with common scientific computing packages such as numpy and scipy, and also comes with an environment manager, conda

3. Install anaconda3 by running the shell script you just downloaded. The installer will suggest an install location in your home directory, which is fine. Let it prepend its anaconda3 install to your PATH, and source your .bashrc to let this change take effect.

```
bash /path/to/downloaded/Anaconda3-x.x.x.x-Linux-x86_64.sh
source ~/.bashrc
```

4. Obtain the pyaeroopt34 conda environment. This consists of a self-contained Python installation that runs Python 3.4, and that already contains the correct dependencies compiled for Independence. Python 3.4 must be used to ensure compatibility with both SNOPT (one of the more powerful optimizers that pyAeroOpt gives you access to) and the MATLAB Python engine, which will let you call MATLAB from Python and is generally very useful.

```
cp -r /home/aspenser/anaconda3/envs/pyaeroopt34 \
      ~/anaconda3/envs/pyaeroopt34
```

5. Before running pyAeroOpt, switch to the pyaeroopt34 conda environment. Anaconda comes with conda, an environment manager that will be helpful ensuring pyAeroOpt is run with the correct environment configuration. conda will let you run pyAeroOpt in Python 3.4 with its dependencies, and will also let you easily switch back to whatever Python configuration you may need for other projects.

```
source activate pyaeroopt34

# To switch back to the default Python environment, run:
source activate root
```

6. Set up environmental variables with paths to sower, partmesh, xp2exo, matcher, Aero-F, Aero-S, Sdesign, and Blender. For instance, place the following lines in your .bashrc.

```
export SOWER=sower
export PARTMESH=partnmesh
export XP2EXO=xp2exo
export MATCHER=matcher
```

```
export AEROF=aerof.opt
export AEROS=aeros
export SDESIGN=sdesign.Linux.opt
export BLENDER=blender
```

7. Run your script

```
python ~/pyaeroopt/test/naca0012/workflow/tutorial.py
```

How to Set up an Optimization Problem

pyAeroOpt is used by writing a Python script to solve your problem, and using pyAeroOpt classes and functions to streamline the development of this Python code. Below we explain a sample script for using pyAeroOpt to solve an optimization problem. The steps below outline how to use pyAeroOpt to find the angle of attack of a NACA 0012 airfoil that maximizes its lift-to-drag ratio at Mach 0.5. The code below can be run in the directory "pyaeroopt/test/naca0012/workflow/" after interactively allocating 12 cores. It can be directly copied-and-pasted into a Python interpreter one step at a time, or found in a Python script in the Python script "tutorial.py" located in this directory. pyAeroOpt classes are called out in red during this example, to distinguish between, for instance, the Aero-F code and the Aerof object provided by pyAeroOpt.

1. Be sure that you are in the correct Python environment, and then launch Python. Your Python interpreter should tell you it is running Python 3.4.

```
source activate pyaeroopt34
python
```

2. Import pyAeroOpt into your script.

```
import pyaeroopt
```

3. Create an Frg object to describe the problem geometry (.top file, FEM input file, binary prefixes, etc.) and an Hpc object to describe the computational environment (cluster name, number of cores, etc). Then create the binaries for the problem using methods of the Frg object.

```
frg = pyaeroopt.interface.Frg(top="mesh/top",
                              geom_pre="mesh/binary/naca0012")
hpc = pyaeroopt.interface.Hpc(machine="independence",
                              mpi="mpiexec",
                              nproc=12)
frg.part_mesh(hpc.ppn)                          # part the mesh
frg.sower_fluid_top([hpc.ppn], hpc.ppn)         # sower the top file
```

4. Create objects to describe the simulations that need to be run. For this problem we need to run a steady Aero-F simulation and calculate the lift and drag, so we will use the Aerof class from pyAeroOpt. While the entire input file could be passed to the Aerof constructor, it is usually easier to create a subclass for each simulation, so we will do this. An Aerof subclass must implement a method `create_input_file(self, p)` that creates an AerofInputFile and assigns it to `self.infile`. Then this class's `execute()` method will run the simulation.

```
from pyaeroopt.interface import Aerof, AerofInputFile, AerofInputBlock
class NACA_Steady(Aerof):
       # Create an input file that is a function of optimization variable(s)
       def create_input_file(self, p):
       mach = p[0];     alpha = p[1]

       # Define the input file one block at a time. Note that the inlet
       # block is a function of the optimization variable. Also note that
       # input blocks can nest.
       prob = AerofInputBlock('Problem', ['Type', 'Steady'],
                                         ['Mode', 'Dimensional'])
       inp  = AerofInputBlock('Input',   ['GeometryPrefix', frg.geom_pre])
       rest = AerofInputBlock('Restart',    ['Solution', 'data/naca0012.sol'])
       post = AerofInputBlock('Postpro', ['LiftandDrag', 'data/liftdrag'],
                                        ['Pressure', 'data/pressure'])
       outp = AerofInputBlock('Output', ['Postpro', post], ['Restart', rest])
       model= AerofInputBlock('FluidModel[0]', ['Fluid', 'PerfectGas'])
       equa = AerofInputBlock('Equations', ['Type', 'Euler'],
                                         ['FluidModel[0]', model])
       wall = AerofInputBlock('Wall', ['Type', 'Adiabatic'])
       inlet= AerofInputBlock('Inlet', ['Mach', mach],
                              ['Beta', alpha], ['Alpha', 0],
                              ['Pressure', 12.71], ['Density', 1.0193e-07])
       bc   = AerofInputBlock('BoundaryConditions', ['Inlet', inlet],
                                                 ['Wall', wall])
       nav  = AerofInputBlock('NavierStokes', ['Reconstruction', 'Constant'])
       space= AerofInputBlock('Space', ['NavierStokes', nav])
       prec = AerofInputBlock('Preconditioner', ['Type', 'Ras'], ['Fill', 0])
       navNewton = AerofInputBlock('NavierStokes', ['MaxIts', 100],
                                  ['KrylovVectors', 100], ['Eps', 0.001],
                                  ['Preconditioner', prec])
       navLin = AerofInputBlock('LinearSolver', ['NavierStokes', navNewton])
       newton = AerofInputBlock('Newton', ['MaxIts', 200], ['Eps', 0.001],
                                         ['LinearSolver', navLin])
       impl = AerofInputBlock('Implicit', ['Newton', newton])
       cfl  = AerofInputBlock('CflLaw', ['Strategy', 'Hybrid'], ['Cfl0', 1.],
                                         ['CflMax', 1000])
       time = AerofInputBlock('Time', ['MaxIts', 1000], ['Eps', 1e-8],
                                  ['Implicit', impl], ['CflLaw', cfl])

       #Pass the input blocks into an AerofInputFile
       fname = "naca.input";     log   = "naca.log"
       self.infile = AerofInputFile(fname, [prob, inp, outp, equa,
                                         bc, space, time], log)
```

5. Create an Optimize object to describe the optimization problem (objective function, variable bounds, constraints, etc), and then run that optimization problem with your chosen optimizer (SNOPT, SLSQP, genetic algorithm, etc). Define a Python function that executes your simulation(s) and extracts the necessary information to define your objective function. We will extract lift and drag information from an output file for our objective function evaluation.

```
import numpy, math
def getLD(alpha):
    # execute the simulation and extract lift, drag from the output file
    NACA_Steady().execute([0.5, alpha], hpc=hpc)
    D0, L0 = numpy.loadtxt('data/liftdrag', usecols=[4,5], unpack=True)
    # rotate lift and drag into the proper frame
    Df = D0[-1];   Lf = L0[-1];   alpha_r = math.radians(-alpha)
    D = Df*math.cos(alpha_r) - Lf*math.sin(alpha_r)
    L = Df*math.sin(alpha_r) + Lf*math.cos(alpha_r)
    print("...At alpha = {:7.8f}, \t L/D is {}".format(alpha, -L/D))
    return -L/D

# Create the Optimize object, define the problem, and run an optimizer.
optimizer = pyaeroopt.opt.Optimize(name="NACA 0012")
optimizer.add_variables(1, [5], [-90], [90])
optimizer.add_objective(lambda alpha: getLD(alpha[0]), name="L/D")
optimizer.optimize('scipy:L-BFGS-B')
```

After some time, our optimizer finds an optimal L/D at an angle of attack of 3.15 degrees. Experiments find that this angle is around 5.5 degrees. In this example we use an inviscid simulation for the sake of simplicity, which explains the discrepancy.

6. Use the FRG object again to create a .exo file of the pressure field so we can visualize the results.

```
frg.sower_fluid_merge('data/pressure', 'pressure', 'pressure')
frg.run_xp2exo('pressure.exo', ['pressure.xpost'])
```

This example is very simple and most optimization problems will require more moving parts (e.g. a call to Sdesign to deform geometry, a separate Aero-F/S call to compute gradients or constraints, etc). pyAeroOpt provides classes for helping with many of these tasks. All of these classes are documented in the following section. The Examples section describes a few more complicated example problems that can be found in the pyAeroOpt source, in the “pyaeroopt/test/” folder.

pyAeroOpt Class Documentation

Below the classes and utilities provided by pyAeroOpt are described in more detail. We include the directory location of each class (which you will need when importing the class into your script), initialization instructions for each class, and the methods that it provides.

1) CodeInterface import pyaeroopt.interface.CodeInterface

CodeInterface objects are used to represent operations that are to be performed by calling external codes (e.g. sower, matcher, Aero-S, Aero-F, Sdesign). Each instance of a CodeInterface represents a single operation to be performed. pyAeroOpt defines subclasses of CodeInterface for the most common codes, described below.

- ## Frg: import pyaeroopt.interface.Frg

An object to facilitate interfacing to FRG utility codes: matcher, sower, partnmesh, xp2exo. All arguments to the Frg constructor are optional keyword arguments. A constructor for an aeroelastic problem might look like :

```
FRG_Object = Frg(top='path-to-the-file/Fluid-top-file',
                 feminp='path-to-the-file/FEM-input-file",
                 geom_pre='prefix-binaries-files')
```

Additional optional keyword arguments for the Frg class constructor are:

- **geom_pre** = path + prefix of the geometry files (.con, .msh, etc.)
- **top** = path to .top file
- **feminp** = path to FEM input file
- **surf_top** = path to surface .top file
- **surf_nodeset** = path to nodeset file
- Additionally, the executables for these utilities can be provided with a keyword argument “[CODE]=path_to_executable” where [CODE] is one of “sower”, “matcher”, “partmesh”, “xp2exo”, or “matcher.” If this path is not given, the path given in your Linux shell environment (see “Obtaining and Running pyAeroOpt”) will be used.

Methods

The methods provided from the FRG class are the following. All of these methods accept the additional optional keyword arguments:

- **log** = the name of a log file to write.
- **make_call** = True by default. Set to False if you want the command to be printed but not actually executed.

```
part_mesh(ndec, log='partmesh.log', make_call=True)
# Parts the mesh into subdomains
ndec = number of decomposition subdomains

run_matcher( log='match.log', make_call=True)
# Runs matcher

sower_fluid_top( cpus, nclust, log='sower.top.log', make_call=True)
# Sowers the .top file to create .dec, .cpu, and .con files
cpus = number of cpu used
nclust = number of clusters used for the simulation
```

```
sower_fluid_extract_surf(log='sower.extract.log', make_call=True)
# extracts a surface from the .top file

sower_fluid_mesh_motion(mm_file, out, log='sower.mm.log', make_call=True)
# sowers a mesh motion file
mm_file = path to the mesh motion file
out = output filename

run_xp2exo(exo_out, xpost_in, log='xp2exo.log', make_call=True)
# Runs xp2exo to convert xpost files into a Paraview-ready format.
exo_out = name of the output .exo file
xpost_in = name of the Xpost input file

sower_fluid_split(file2split, out, from_ascii=True, log='sower.split.log',
                  make_call=True)
# Splits a file into subdomains to apply as an IC or BC
file2split = name of the file to split into subdomains
out = prefix of the files after being split into subdomains
from_ascii = select true if you want as output an ascii file

sower_fluid_merge(res_file, out, name, from_bin=False, log='sower.merge.log',
                  make_call=True)
# Merges the result of a simulation into an Xpost file.
res_file = prefix of the binary result to be merged into xpost format
out = name of the xpost file to create, minus the '.xpost' suffix
name = field name to be placed in the xpost file header
from_bin = true if you want to set SOWER's '-binary' flag
```

- ## Aerof import pyaeroopt.interface.Aerof

This is an object to facilitate interfacing with Aero-F. It will create the input file and execute the code. To use this class, create a subclass of the Aerof object and then implement the method `create_input_file(self, p)`, which defines the input file for your simulation as a function of your optimization variables. This method needs to accept as an argument an array of design variables, `p`, and should use the design variables to create an AerofInputFile object and assign it to the instance variable `self.infile`. An AerofInputFile object is constructed with

```
input_file = AerofInputFile(fname, blocks, log):
      fname: The filename for the input file
      blocks: A list of AerofInputBlocks defining the input file
      log: the name of the log file when the simulation is run
```

Each of the AerofInputBlocks in the `blocks` list above is constructed with

```
input_block = AerofInputBlock(header, properties):
      header: a string specifying the header for this input block
          e.g. 'Input', 'PostPro', 'NavierStokes'
      properties: the header can be followed by any number of 2-element
          lists of giving the properties in this block. For
```

```
        instance, a 'Problem' block of an Aero-F input file could have
        properties ['Mode', 'Dimensional'] and ['Type', 'Steady'].
```

An example of this approach is given in the section "How to Set Up an Optimization Problem," and is outlined below.

```
from pyaeroopt.interface import Aerof, AerofInputFile, AerofInputBlock
class class_4_my_simulation(Aerof):

    # Construct using the Aerof constructor
    def __init__(self, **kwargs):
        super(class_4_my_simulation, self).__init__(**kwargs)

    # create an AerofInputFile, with blocks as functions of p
    def create_input_file(self, p):
        input = AerofInputBlock('Problem', ['Type', 'Steady'],
                                           ['Mode', 'Dimensional')
        inlet = AerofInputBlock('Inlet', ['Alpha', p[0]], ['Beta', p[1]])
        # Continue for other input blocks

        fname = 'input_filename';      log = 'log_filename'
        self.infile = AerofInputFile(fname, [input, other_block_1,
                                             other_block_2], log)
```

Because this user-defined class is a subclass of the Aerof object provided by pyAeroOpt, it still has access to the methods of the Aerof object, described below.

Methods

The Aerof object provides two methods, one to write the input file without executing the simulation, and another to run the simulation. The function declarations for these methods are:

```
writeInputFile():
    # writes the input file without running the simulation

execute(p, hpc=None, make_call=True):
    # writes the input file and runs the simulation
    hpc: an Hpc object specifying how the simulation should be run
    make_call: True to execute the simulation, False to just print the
         command that would be executed.
```

- ## Aeros import pyaeroopt.interface.Aeros

This is an object to facilitate interfacing with Aero-S. It will create the input file and execute the code. To use this class, create a subclass of the Aerof object and then implement the method `create_input_file(self, p)`, which defines the input file for your simulation as a function of your optimization variables. This method needs to accept as an argument an array of design variables, `p`, and should use the design variables to create an AerosInputFile object and assign it to the instance variable `self.infile`. An AerosInputFile object is constructed with

```
input_file = AerosInputFile(fname, blocks, log):
      fname: The filename for the input file
      blocks: A list of AerosInputBlocks defining the input file
      log: the name of the log file when the simulation is run
```

Each of the AerosInputBlocks in the `blocks` list above is constructed with

```
input_block = AerosInputBlock(header, properties):
      header: a string specifying the header for this input block
          e.g. 'EIGEN', 'STATIC', 'AERO'
      properties: the header can be followed by any number of lists giving
          the properties in this block. For instance, a 'AERO' block of an
          Aero-F input file could have properties ['MPP', 0.1] and
          ['Matcher', 'path_to_match.fem'].
```

An example of this approach is given in the section "How to Set Up an Optimization Problem," and is outlined below.

```
from pyaeroopt.interface import Aeros, AerosInputFile, AerosInputBlock
class class_4_my_simulation(Aeros):

    # Construct using the Aeros constructor
    def __init__(self, **kwargs):
        super(class_4_my_simulation, self).__init__(**kwargs)

    # create an AerosInputFile, with blocks as functions of p
    def create_input_file(self, p):
        eigen = AerosInputBlock('EIGEN', ['nsbspv', 14],
                                         ['neigpa', 6])
        dynam = AerosInputBlock('DYNAMICS', ['Newmark'], ['mech', 0.5, 0.5)
        # Continue for other input blocks

        fname = 'input_filename';      log = 'log_filename'
        self.infile = AerosInputFile(fname, [eigen, other_block_1,
                                             other_block_2], log)
```

Because this user-defined class is a subclass of the Aeros object provided by pyAeroOpt, it still has access to the methods of the Aeros object, described below.

Methods

The Aeros object provides two methods, one to write the input file without executing the simulation, and another to run the simulation. The function declarations for these methods are:

```
writeInputFile():
    # writes the input file without running the simulation

execute(p, hpc=None, make_call=True):
    # writes the input file and runs the simulation
    hpc: an Hpc object specifying how the simulation should be run
    make_call: True to execute the simulation, False to just print the
```

```
        command that would be executed.
```

- ## Aerofs import pyaeroopt.interface.Aerofs

  A class that allows an Aerof object and an Aeros object to be bundled for joint execution. First, create an Aerof object and an Aeros object (explained in the previous sections). Create an Hpc object whose instance variable `nproc` is a list of two numbers, one for the Aero-F simulation and one for the Aero-S simulation (e.g. `hpc.nproc=[12, 1]`). Then, create the Aerofs object as follows:

```
Aerofs_object = Aerofs( Aerof_object , Aeros_object )
```

  Execute the joint simulation just as with any other pyAeroOpt code interface.

```
hpcCoupled = Hpc(machine='Independence', nproc=[12,1])
Aerofs_object.execute( x, hpc=hpcCoupled )   # x is a parameter vector
```

- ## Blender

  This is an object to facilitate interfacing with Blender[2], a powerful shape deformation tool. Blender is not an input-file based code, so pyAeroOpt's interface to Blender is slightly different than its interface to other codes (like Aero-S, Aero-F, and Sdesign).

  In Blender, a lattice of points surrounding the object being deformed is defined. Then some number of lattice points is allowed to move in one, two, or three dimensions. An interpolation procedure is performed to map the motion of these lattice points onto the motion of the object contained in the lattice. If the position of each lattice point is moved directly, the displacements of these points are called "design variables" in pyAeroOpt's terminology. If instead the nodes of the lattice are moved indirectly by global variables (for instance, a 'sweep angle' is specified, and the position of the lattice nodes is a function of this sweep angle), then these global variables are called "abstract variables."

  The process of using pyAeroOpt to define a lattice deformation has the following steps:

  1. Define a Lattice object (or nested list of Lattice objects) to describe each deformation to be performed.
  2. Create Modifier objects for each deformation (e.g. sweep, twist, extension) and add these Modifiers to the Blender object.
  3. Declare each of these modifiers as their own design variables so that you can specify the value of each modifier, which will in turn specify the lattice deformation.
  4. Now the Blender object is ready to have its `execute(p)` method perform a shape deformation operation.

[2] The Python interface to Blender was built by George Anderson. His paper on the subject is: George Anderson, Michael Aftosmis, and Marian Nemec. "Parametric Deformation of Discrete Geometry for Aerodynamic Shape Design", 50th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition, Aerospace Sciences Meetings

An example of each of these steps is given below.

Lattice Objects import pyaeroopt.interface.Lattice

First, define a lattice. An example is below.

```
from pyaeroopt.interface import Lattice, Modifier
def create_lattices():
    ## Lattice 0: stretches over outboard portion of wing, aligned with
    ##         trailing edge. Each degree of freedom is controlled by the
    ##         abstract variable 'length'
    part0 = (16, 16, 8)     # number of nodes in the x, y, and z directions
    rot0 = {'vec': np.array([0.0, 0.0, 1.0]),  #  rotation angle for lattice
            'angle': -np.pi/4.85}
    center0 = (1675.0, 900.0, 437.5)    # the center of the lattice
    scale0  = (900.0, 1150.0, 850.0)    # lattice length in each direction
    latt0 = Lattice(0, part0, center0, scale0, rot0, interp='BSPLINE')

    # Make every lattice point's 3D coordinates a design variable
    id=-1
    for k in np.arange(part0[2]):
        for j in np.arange(1, part0[1]):
            for i in np.arange(part0[0]):
                for dir in [0, 1, 2]:
                    id+=1
                    latt0.add_degree_of_freedom(id, [(i,j,k)], [dir],
                                                [length])
    # 'length' is a Python function (not shown) that maps from the abstract
    # variable 'length' to the value of each lattice point deformation.
    latt0.declare_degrees_of_freedom('absvar', 1)
    return latt0
```

A Lattice object is initialized with the constructor:

```
lattice_obj = Lattice(id, part=(2,2,2), center=(0.0,0.0,0.0),
                scale=(1.0,1.0,1.0), rot=None, **kwargs):
id: an integer identifier for this object
part: tuple giving the number of lattice points in the x, y, z directions
center: the center location of the lattice, as a 3-tuple
scale: the length of the lattice in the x, y, z directions
rot: a dictionary {'vec': np.array([x, y, z]); 'angle': theta}, where the
    'vec' field is an array specifying a vector to rotate about, and theta
    is an angle in radians
```

A Lattice object has the following methods, for defining it's DOFs and stating the type of variables for these DOFs:

```
add_degree_of_freedom(id, nodes_lattice, dirs, expr=None):
id: an integer identifier for this set of DOFs
nodes_lattice: Node no. of nodes whose dofs will be lumped into a Design
               Variable
dirs: list of degrees of freedom to lump into this design variable
```

```
expr: an optional Python function mapping from a vector of abstract
      variables to the value of each design variable.

declare_degrees_of_freedom(vtype, nabsvar=0):
vtype: either 'dsgvar' for a design variable, or 'absvar' for an abstract
       variable
nabsvar: number of abstract variables being declared
```

Modifier Objects import pyaeroopt.interface.Modifier

Now that we have created a lattice, we need to make a corresponding Modifier to add to the Blender object. To add the lattice we made above, we can use

```
latt0 = create_lattices()          # lattice defined above
mod0 = Modifier(0, [latt0])        # Add the lattice to a Modifier
```

The Modifier constructor is

```
mod_obj = Modifier(id, list_def):
id: an integer identifier for the modifier
list_def: list of lattices, which will be performed on each other
     in sequence. The abstract variables will be applied to list_def[0],
     then list_def[i] will deform the nodes of list_def[i+1] (ignoring
     how its DOFs were declared), and list_def[-1] will modify the final
     set of nodes on the mesh.
```

Now add the Modifier to the Blender object. We only have one Modifier in our running example, but a full problem would have one for each abstract variable (sweep, twist, etc), and each would need to be added to the Blender object. Add this modifier to the Blender object with the Blender methods `make_degrees_of_freedom_from_modifiers()` and `declare_degrees_of_freedom('dsgvar')`, which are explained in the Blender methods section below.

Lastly, to have a functional pyAeroOpt CodeInterface, we need to implement `create_input_file`. A full Blender subclass, with Modifiers added to it and with a `create_input_file` method implemented, is shown below:

```
from pyaeroopt.interface.blender import Blender
class class_4_my_deformation(Blender):
    def __init__(self, **kwargs):
        super(class_4_my_deformation, self).__init__(**kwargs)

        # Create Lattices to give as input to Blender
        latt0 = create_lattices()          # lattice defined above
        mod0 = Modifier(0, [latt0])        # Add the lattice to a Modifier
        self.add_modifier(0, mod0)         # add Modifier to Blender object

        self.make_degrees_of_freedom_from_modifiers()
        self.declare_degrees_of_freedom('dsgvar')
```

```
    def create_input_file(self, p, desc_ext, db=None):
        self.ptcloud = "my_nodeset_file"    # filename for points to deform
        self.log     = "my_logfile"         # log file name
        self.vmo     = "my_vmo_file"        # filename to write displacements
        self.der     = "my_der_file"        # filename to write derivatives
        self.xpost   = "my_xpost_file"      # filename for deformed xpost
```

Like other code interfaces, a `create_input_file(p, desc_ext=None, db=None)` method must be implemented, but rather than creating an InputFile object as in other interface objects, this method must assign strings to the instance variables:

- `self.ptcloud`: the filename of the points to be moved. This file must be formatted as a .top file NodeSet.
- `self.log`: the filename of the log file to be used when Blender is run.
- `self.vmo`: the filename for the .vmo file (a file of node displacements) that will be written by Blender.
- `self. der`: the filename of the shape derivative file to be written by Blender
- `self.xpost`: the filename of the shape-deformed xpost file that will be written

Once all of this has been done, the Blender object can be executed to perform a shape deformation with the usual pyAeroOpt execute command:

```
hpc_blender = Hpc(machine='independence', mpi='mpiexec',
                  batch=False, bg=True, nproc=1)
myobject_blender = class_4_my_deformation()
myobject_blender.execute(x, hpc=hpc_blender)  # x is a design variable array
```

Blender Methods

The methods provided for the Blender class are the following:

```
execute( p, desc_ext=None, hpc=None)
# Solve for displacements or derivatives using Blender from value of DOFs.
p: a vector of deformation variables
desc_ext: an optional additional parameter, for if you need to pass any
    information into this class other than the parameter array
hpc: hpc object

add_modifier( id, obj, weight=1.0)
# Add a modifier to the blender object. The deformation will be the result
# of all modifiers acting on the object.
id : integer unique identifier
obj : Modifier object

add_degree_of_freedom( id, mod_ids, mod_dofs, expr=None)
# Add degrees of freedom to Blender object.  Can combine DOFs of multiple
# nodes into single DOF (called Design Variable in SDESIGN terminology). Can
# also use an expression (python function) that will accept a vector of
# Abstract Variables (in SDESIGN terminology) and return the value of the
# Design Variable. If 'expr' specified for one Design Variable, it must be
# specified for all, and each must take arguments of the same length
```

```
# (nabsvar in declare_degrees_of_freedom).
id = int
mod_ids = ndarray (or iterable) of int. Modifier ids whose dofs (all or some)
will be lumped into a Design Variable
mod_dofs = list (len = len(mod_ids)) of lists of int. Degree of Freedom for each
modifier that will be lumped into the current Design Variable
expr = function. Map from the vector of (all) Abstract Variables to the present
Design Variable (returns scalar)

declare_degrees_of_freedom( vtype, nabsvar=0)
# Function to declare degrees of freedom (type and number).  If Abstract
# Variables are used by specifying 'expr' in add_degre_of_freedom, they must #
all accept the same number of arguments (nAbsVar).
vtype = str. Type of degree of freedom
nabsvar = int. Number of abstract variables
```

- ## Sdesign

This is an object to facilitate interfacing with Sdesign, an input-file based shape deformation tool that is simpler than Blender. It will create the input file and execute the code. To use this class, create a subclass of the Sdesign object and then implement the method `create_input_file(self, p)`, which defines the input file for your shape deformation as a function of your optimization variables. This method needs to accept as an argument an array of design variables, `p`, and should use the design variables to create an SdesignInputFile object and assign it to the instance variable `self.infile`. An SdesignInputFile object is constructed with

```
input_file = SdesignInputfile(fname, blocks, log):
      fname: The filename for the input file
      blocks: A list of SdesignInputBlocks defining the input file
      log: the name of the log file when Sdesign is run
```

Each of the SdesignInputBlocks in the `blocks` list above is constructed with

```
input_block = SdesignInputBlock(header, properties):
      header: a string specifying the header for this input block
          e.g. 'PATCH', 'DSGVAR', 'ABSVAR'
      properties: the header can be followed by any number of
          lists giving the properties in this block. For
          instance, a 'DEFINE' block of an Sdesign input file could have
          properties ['x1', 220] and ['z2', 'x2 - x1'].
```

An example of this approach is given in the section "How to Set Up an Optimization Problem," and is outlined below.

```
from pyaeroopt.interface import Sdesign, SdesignInputFile, SdesignInputBlock
class class_4_my_deformation(Sdesign):

    # Construct using the Sdesign constructor
```

```
    def __init__(self, **kwargs):
        super(class_4_my_simulation, self).__init__(**kwargs)

    # create an SdesignInputFile, with blocks as functions of p
    def create_input_file(self, p):
        output = SdesignInputBlock('SDOUTPUT', ['varmode', 'position', 0])
        define = SdesignInputBlock('DEFINE', ['x1', p[0]], ['x2', p[1]])
        # Continue for other input blocks

        fname = 'input_filename';      log = 'log_filename'
        self.infile = SdesignInputFile(fname, [define, other_block_1,
                                               other_block_2], log)
```

Because this user-defined class is a subclass of the Sdesign object provided by pyAeroOpt, it still has access to the methods of the Sdesign object, described below.

Methods

The Sdesign object provides two methods, one to write the input file without executing Sdesign, and another to run the simulation. The function declarations for these methods are:

```
writeInputFile():
# writes the input file without running the simulation

execute(p, hpc=None, make_call=True):
# writes the input file and runs the simulation
hpc: an Hpc object specifying how the simulation should be run
make_call: True to execute the simulation, False to just print the
         command that would be executed.
```

3) HPC: import pyaeroopt.interface.Hpc

An object to facilitate executing codes on a cluster: Independence, Kratos, or Copper. It defines how many nodes to use and other specifics related to a particular simulation (maximum runtime, queue, etc.) All of its constructor arguments are optional keyword arguments. A typical initialization of an Hpc object might look like

```
hpc = Hpc(machine='independence', mpi='mpiexec', nproc=32)
```

Optional keyword arguments for the Hpc class constructor are:

- **machine** = either “independence,” “kratos,” or “copper.”
- **mpi** = name of the mpi command to use. Typically “mpiexec” or “mpirun”
- **nproc** = number of processes to create.
- **bg** = Boolean, whether or not to execute commands in the background.
- **nompi** = True if you want to execute without MPI.

Methods:

This class has no methods. It is passed into other objects and functions to specify how a code should be executed.

4) Optimize import pyaeroopt.opt.Optimize

The Optimize object is pyAeroOpt's interface for optimization. It helps interfacing with various optimization solvers such as pyOpt, scipy.optimize, SNOPT, etc. This interface allows the user to set up an optimization problem once, and then solve it with multiple solvers.

An Optimize object is initialized with a constructor that only takes optional keyword arguments. A typical initialization might look like

```
opt_object = Optimize(name="My Optimization Problem")
```

Optional keyword arguments are:

- **inf:** The value to use for positive infinity. Defaults to 1e20.
- **minf**: The value to use for negative infinity. Defaults to -1e20.
- **name:** A name for the problem. Some optimizers produce summary printouts with a problem name.

Methods

The Optimize class has the following methods. They are demonstrated in the example at the start of this manual.

```
add_variables(self, nvar, init, low, up):
nvar = the number of variables to add
init = an array of initial values for the variables being added
low = an array of lower bounds for the variables being added
up = an array of upper bounds for the initial variables being added

add_objective(self, obj, name=None, grad=None, hess=None,
                    hess_vec=None, obj_failure=lambda x: False,
                    grad_failure=lambda x: False):
obj = a function that returns a single number, to be minimized
name = a name that some optimizers will include in a summary printout
grad = A function that returns a gradient array whose length is the number
      of design variables
hess = A function that returns a Hessian matrix with shape (n_vars, n_vars)
hess_vec = A function that returns the product of the Hessian with a vector x,
    as a 1D array
obj_failure = A function that returns True if the objective evaluation failed
grad_failre = A function that returns True if the gradient evaluation failed

add_lin_constraints(self, nconstr, mat, low, up):
nconstr = The number of variables in the linear constraint
```

```
mat = The constraint matrix A in the constraint a <= A*x <= b
low = The lower bound a in the linear constraint a <= A*x <= b
up = The upper bound b in the linear constraint a <= A*x <= b

add_nonlin_constraints(self, nconstr, constr, jac, low, up, name=None,
                             constr_fail=lambda x: False,
                             jac_fail=lambda x: False):
nconstr = The number of variables in the nonlinear constraint
constr = a function returning an array of constraint function evaluations
jac = A function returning the derivative of the constraints as a 2D array with
    shape (n_constrains, n_var)
low = an array of lower bounds in the nonlinear constraint a <= g(x) <= b
up = an array of upper bounds in the nonlinear constraint a <= g(x) <= b
name = a name for this constraint. Some optimizers will print a results summary
    that displays the name of active constraints.
constr_fail = a function that returns True if the constraint evaluation failed
jac_fail = a function that returns True if the gradient evaluation failed

optimize(self, solver, sens=None, options=None, callback=None):
solver = a string specifying a solver to use. Takes the form "package:solver"
    where "package" is either "scipy" or "pyoptsparse" and "solver" is the name
    of any solver accessible from one of those packages (e.g. "scipy:SLSQP")
options = a dictionary of additional options to pass to the solver
callback = a function to be called after each iteration as callback(xk), where
    xk is the parameter vector at the end of iteration k.
```

Examples

The pyAeroOpt source code 'pyaeroopt/test' directory contains several well-documented sample problems of varying complexity.

- The directory 'pyaeroopt/test/naca0012' solves a very simple optimization problem that finds the angle of attack that maximizes a symmetric airfoil's lift-to-drag ratio. The entirety of the problem's code is in the file 'pyaeroopt/test/naca0012/workflow/tutorial.py.'
- The directory 'pyaeroopt/test/ARW2' solves a much more complicated design optimization problem, varying structural and shape parameters for a transport aircraft wing in order to maximize its lift-to-drag ratio subject to several nonlinear constraints. The simulation and shape deformation classes are defined in the directory 'pyaeroopt/test/ARW2/pyao.' There are classes here for shape deformation, steady analysis, aeroelastic analysis, structural analysis, and ROM construction. The optimization problem is run with the script 'pyaeroopt/test/ARW2/workflow/ workflow.py.' The binaries are created with the script 'pyaeroopt/test/ARW2/workflow/ makebin.py.' This problem uses Aero-F/S to provide gradients for constraints and the objective function, and calls a MATLAB code to evaluate a flutter constraint and the gradient of this constraint. The example is complicated but demonstrates most of pyAeroOpt's functionality.