

## Basics

This document provides a brief overview of the core concepts and constructs involved in defining and solving a problem using OpenMDAO. We start by explaining the *System*, which forms the mathematical foundation of OpenMDAO, then discuss how *System* relates to *Component* and *Group*. *Component* is the computational class of OpenMDAO, where you build models and wrap external analysis codes. *Group* represents collections of *Components* and other *Groups* with data passing and an execution sequence. Lastly we discuss *Problem*, which serves as the container for your whole model.

## System

OpenMDAO uses a unique abstraction for representing large models of coupled systems, based on the Modular Analysis and Unified Derivatives ([\[MAUD\]](#)) mathematical architecture developed by Hwang and Martins. The fundamental concept in MAUD is that an entire system model can be represented as a hierarchical set of systems of non-linear equations.

[\[MAUD\]](#) Hwang, J. T., A modular approach to large-scale design optimization of aerospace systems, Ph.D. thesis, University of Michigan, 2015. [Thesis pdf](#). (168 pages)

Hence the most basic building block in OpenMDAO is the *System* class. This class represents a system of equations that need to be solved together so that a single solution satisfies them all.

A system of equations is specified by one or more parameters (input values) and one or more unknowns (output values). For example, a really simple explicit system may contain only a single equation, such as:

$$y = x^2 + 2$$

For this system,  $x$  is the parameter (input variable) and  $y$  is the unknown (output variable). Implicit equations, where you vary a state value to drive a residual to 0, can also be part of a system. The above equation could be restated implicitly as:

$$R(y) = x^2 + 2 - y = 0$$

There is now a corresponding residual value for  $y$  as well. The *System* class has *params*, *unknowns*, and *resids* attributes that store the lists of parameters and unknowns as vectors for efficient processing.

### Note

Unknowns come in two flavors: *outputs* are computed by explicit equations. *states* are unknowns that are varied to drive residuals to 0.

The equations themselves are encapsulated in a member function of the *System* class called *solve\_nonlinear*. For explicit equations *solve\_nonlinear* will compute the unknown values for the given parameter values and put them into the unknown vector. For implicit relationships, *solve\_nonlinear* will find the correct values for the state variables that converge the residuals. A second function, *apply\_nonlinear*, is used to compute the residual values for a given state value (it does not fully converge the system of equations). *apply\_nonlinear* is not used if you have only explicit equations.

There are a few other attributes and member functions in the *System* interface, mostly related to calculating derivatives and supporting more complex systems, but this is the essential base abstraction.

There are two subclasses of *System* that are used to actually build a model. They are the *Component* class and the *Group* class.

## Component

The *Component* class is the lowest level system in OpenMDAO. Child classes of *Component* are the only classes allowed to create parameter, output, and state variables. By sub-classing *Component* and defining a *solve\_nonlinear* (and *apply\_nonlinear* if state variables are present), users can build their own models or implement wrappers for existing analysis codes.

Variables are added to the class in the constructor (*\_\_init\_\_* method) via the *add\_parameter*, *add\_output* and *add\_state* functions. For example:

```
class MyComp(Component):
    def __init__(self):
        super(MyComp, self).__init__()
```

```
self.add_param('x', val=0.)
self.add_output('y', shape=1)
self.add_state('z', val=[0., 1.])
```

#### Note

Initial values (or shape and type) are required when adding variables to a component in order to allocate the needed space in the vectors for data passing. If only a shape is given, the type is assumed to be *float* if shape is 1 and a numpy float array if shape has any other value.

The *solve\_nonlinear* function takes three arguments: the parameters vector, the unknowns vector, and a residuals vector. This function will be called using the vector attributes of the *Component* instance, so those vectors will contain entries for the variables declared in the constructor. For example:

```
def solve_nonlinear(self, params, unknowns, resids):
    unknowns['y'] = params['x']**2 + 2
```

When implicit equations are involved, the *apply\_nonlinear* method must be implemented. This function is written to compute the residual values for whatever values of the parameters and state variables are given. The user must then decide how the implicit equations should be converged. There are two choices:

1. Use an OpenMDAO solver
2. Have the component converge itself

We'll talk more about solvers in later docs, but if you go this route then you can stop at *apply\_nonlinear*. But if you would like a component with state variables to converge itself, you will also define the *solve\_nonlinear* method.

```
def apply_nonlinear(self, params, unknowns, resids):
    resids['y'] = params['x']**2 + 2 - unknowns['y']

def solve_nonlinear(self, params, unknowns, resids):
    """
    Only used if the component is able to converge its
    own residuals
    """
    while abs(resids['y']) > 1e-5:
        self.apply_nonlinear(params, unknowns, resids)
        unknowns['y'] += resids['y']
```

## Component Derivatives

If you want to define analytic derivatives for your components, to help make your optimizations faster and more accurate, then your component will also define a *linearize* method, that linearizes the non-linear equations and provides the partial derivatives (derivatives of unknowns w.r.t parameters for a single component) to the framework.

```
def linearize(self, params, unknowns, resids):
    J = {}
    J['y', 'x'] = 2*params['x']
    J['y', 'y'] = 1
```

#### Note

When you're providing derivatives for implicit equations, you give derivatives of the residual with respect to the params and state variables: ('y','x') and ('y','y')

## Group


*Group* is used to build a complex model out of smaller sub-system building blocks, which may be instances of either *Component* or *Group*. So a *Group* is just a *System* composed of the equations from its children that are coupled together via data connections. Because groups can contain other groups, they form a hierarchy that defines the organizational structure of your model.

A *Group* is created simply by adding one or more *Systems*. For example, we can add a *Group* to another *Group* along with some *Components*:

```
c1 = MyComp()
c2 = MyComp()
c3 = MyComp()
```

```
g1 = Group()
g1.add('comp1', c1)
g1.add('comp2', c2)


g2 = Group()
g2.add('comp3', c3)
g2.add('sub_group_1', g1)
```

Visualize this example: 

Interdependencies between *Systems* in a *Group* are represented as connections between the variables in the *Group*'s subsystems. Connections can be made either explicitly or implicitly.

An explicit connection is made from the output (or state) of one *System* to the input (parameter) of another using the *Group connect* method, as follows:

```
g2.sub_group_1.connect('comp1.y', 'comp2.x')
```

Visualize this example: 


Alternatively, you can use the *promotion* mechanism to implicitly connect two or more variables. When a *System* is added to a *Group*, you may optionally specify a list of variable names that are to be *promoted* from the subsystem to the group level. This means that you can reference the variable as if it were a variable of the *Group* rather than the subsystem. For Example:

```
g2.add("comp3", c3, promotes=['x'])
```

Now you can access the parameter 'x' from 'c3' as if it were a variable of the group: 'g2.x'. If you promote multiple subsystem variables with the same name, then those variables will be implicitly connected:

```
g2.add("sub_group_1", g1, promotes=['comp1.x'])
```

Now setting a value for 'g2.x' will set the value for both 'c3.x' and 'g1.c1.x' and they are said to be implicitly connected. If you promote the output from one subsystem and the input of another with the same name, then that will have the same effect as the explicit connection statement as shown above.

Visualize this example: 

In contrast to a *Component*, which is responsible for defining the variables and equations that map between them, a *Group* has the responsibility of assembling multiple systems of equations and solving them together. A *Group* uses a *Solver* to solve the collection of *Components* as a whole. In fact, a *Group* has two associated solvers: a linear solver and a non-linear solver. The default linear solver is SciPy's GMres and the default non-linear solver is a simple *RunOnce* solver that will just call the `solve_non_linear` method on each system in the *Group* sequentially. A number of other iterative solvers, both linear and non-linear, are available that can be substituted for the defaults.

## Problem

When a model has been fully developed as a *Group* with a collection of *Components* and sub-*Groups* it is time to actually do something with it (e.g. run an analysis, design of experiments, or optimization). This is done by defining a single top level object, a *Problem* instance, that contains your model.

A *Problem* always has a single top-level *Group* called *root*. This can be passed in the constructor or set later:

```
prob = Problem(ExampleGroup())


or

root = ExampleGroup()
prob = Problem(root)
```

A *Problem* also has a driver, which "drives" or controls the solution of the *Problem*. The base *Driver* class in OpenMDAO is the simplest driver possible, which just calls `solve_nonlinear` on the *root Group*. This simple driver may be replaced with more interesting types like optimization, case iteration, and design of experiment drivers. Essentially, the *Driver* determines how the *Problem* will execute your model.

The *Driver* is invoked by calling the `run` method on the *Problem*. Prior to doing that, however, you must perform *setup*. This function does all the necessary initialization of the data vectors and configuration for the data

transfers that must occur during execution. It will also look for and report any potential issues with the *Problem* configuration, including unconnected parameters, conflicting units, etc.

Visualize this example: 

## Summary

The general procedure for defining and solving a *Problem* in OpenMDAO is:

- define *Components* (including their *solve\_nonlinear* and optional *linearize* functions)
- assemble *Components* into Groups and make connections (explicitly or implicitly)
- instantiate a *Problem* with the *root Group*
- perform *setup* on the *Problem* to initialize all vectors and data transfers
- perform *run* on the *Problem*

A very basic example of defining and running a *Problem* with a custom *Component* is shown below. This example makes use of the convenience component *IndepVarComp* to provide a source for the input parameter to the custom *MultiplyByTwoComponent*.

```
from __future__ import print_function

from openmdao.api import Group, Problem, Component, IndepVarComp

class MultiplyByTwoComponent(Component):
    def __init__(self):
        super(MultiplyByTwoComponent, self).__init__() # always call the base class constructor first
        self.add_param('x_input', val=0.) # the input that will be multiplied by 2
        self.add_output('y_output', shape=1) # shape=1 => a one dimensional array of length 1 (a scalar)

        # an internal variable that counts the number of times this component was executed
        self.counter = 0

    def solve_nonlinear(self, params, unknowns, residuals):
        unknowns['y_output'] = params['x_input']*2
        self.counter += 1

root = Group()
root.add('indep_var', IndepVarComp('x', 7.0))
root.add('my_comp', MultiplyByTwoComponent())
root.connect('indep_var.x', 'my_comp.x_input')

prob = Problem(root)
prob.setup()
prob.run()

result = prob['my_comp.y_output']
count = prob.root.my_comp.counter
print(result)
print(count)
```

Running this example produces the output:

```
14.0
1
```

Tags

[System](#) , [Component](#) , [Group](#) , [Problem](#) , [Derivatives](#)