Multidisciplinary Analysis with OpenMDAO

Computational Design Laboratory

Department of Aerospace Engineering
Iowa State University

October 3, 2020

Outline

- Installing OpenMDAO
- OpenMDAO structure
- Single-discipline analysis
- Two-discipline analysis
- N-squared diagrams
- Further reading

Installing OpenMDAO

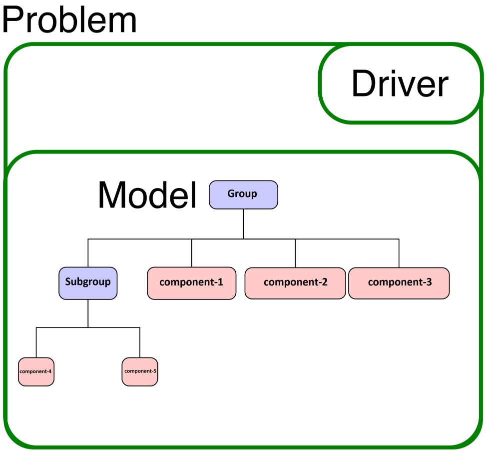
- Download OpenMDAO codes from Github: Link
- Unzip OpenMDAO from where it saved
- Open Anaconda Prompt
- Use cd command to the OpenMDAO directory
- Install OpenMDAO with the command: pip install openmdao[all]

OpenMDAO structure

Structure and classes in OpenMDAO:

- **Component** - does all the numerical calculations
  - e.g. Explicit component (function), Implicit component (implicit function)
- **Group** - contains components and other groups
  - multiple groups and components make **model**
- **Driver** - defines algorithms that iteratively call the model
- **Problem** - defines a top-level container, holding all other objects

OpenMDAO structure

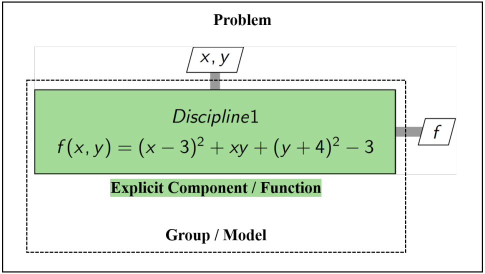


Single-discipline analysis

Consider the paraboloid problem with one discipline and two inputs:

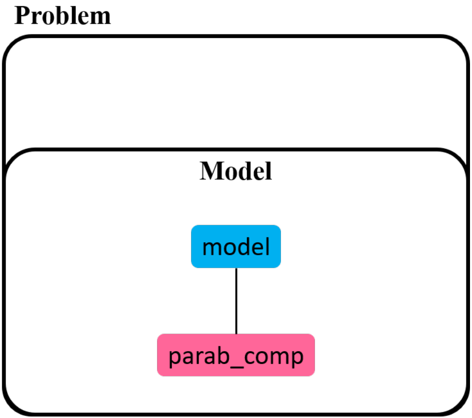
$$f(x, y) = (x - 3)^2 + xy + (y + 4)^2 - 3$$

The group/model of this problem contains only one component:



The XDSM code for this figure: Link

Single-discipline analysis



Single-discipline analysis

- Download code from Github: Link
- Open mda_single_disp.py in Spyder
- Part 0: Import OpenMDAO api
- Part 1: Create a explicit component Paraboloid

```python
# Part 0: OpenMDAO and component imports
import openmdao.api as om

# Part 1: Create a new explicit components for f_xy
class Paraboloid(om.ExplicitComponent):
    """
    Evaluates the equation f(x,y) = (x-3)^2 + xy + (y+4)^2 - 3.
    """

    def setup(self):
        self.add_input('x', val=0.0)
        self.add_input('y', val=0.0)

        self.add_output('f_xy', val=0.0)

    def setup_partials(self):
        # Finite difference all partials.
        self.declare_partials('*', '*', method='fd')

    def compute(self, inputs, outputs):
        """
        f(x,y) = (x-3)^2 + xy + (y+4)^2 - 3
        Minimum at: x = 6.6667; y = -7.3333
        """
        x = inputs['x']
        y = inputs['y']

        outputs['f_xy'] = (x - 3.0)**2 + x * y + (y + 4.0)**2 - 3.0
```

Single-discipline analysis

- Part 2: Create a group (model) and add Paraboloid as its subsystem
- Part 3: Create a problem from group and set it up
- Part 4: Provide input variables x and y to the problem
- Part 5: Run the problem

```python
if __name__ == "__main__":
    # Part 2: Create a group and Paraboloid as subsystem of group
    model = om.Group()
    model.add_subsystem('parab_comp', Paraboloid())

    # Part 3: Create problem from the group and setup the problem
    prob = om.Problem(model)
    prob.setup()

    # Part 4: Provide x and y input to the problem
    prob.set_val('parab_comp.x', 3.0)
    prob.set_val('parab_comp.y', -4.0)

    # Part 5: Run the problem
    prob.run_model()
```

Single-discipline analysis

- Part 6: Print the input and output of the problem
- Part 7: Provide new variables and re-run the problem

```
# Part 6: Print the input and output of the problem
print('x =',prob['parab_comp.x'])
print('y =',prob['parab_comp.y'])
print('f_xy =',prob.get_val('parab_comp.f_xy'))

print('\n----------------\n')
# Part 7: Provide new input variables and print output
prob.set_val('parab_comp.x', 5.0)
prob.set_val('parab_comp.y', -2.0)
prob.run_model()
print('x =',prob['parab_comp.x'])
print('y =',prob['parab_comp.y'])
print('f_xy =', prob.get_val('parab_comp.f_xy'))
```

- For more detailed explanation of the setup: Link

Single-discipline analysis

- Input-1 details: $x = 3, y = -4$
- Input-2 details: $x = 5, y = -2$

```
x = [3.]
y = [-4.]
f_xy = [-15.]

----------------

x = [5.]
y = [-2.]
f_xy = [-5.]

In [2]:
```

IPython console | History log

Single-discipline analysis

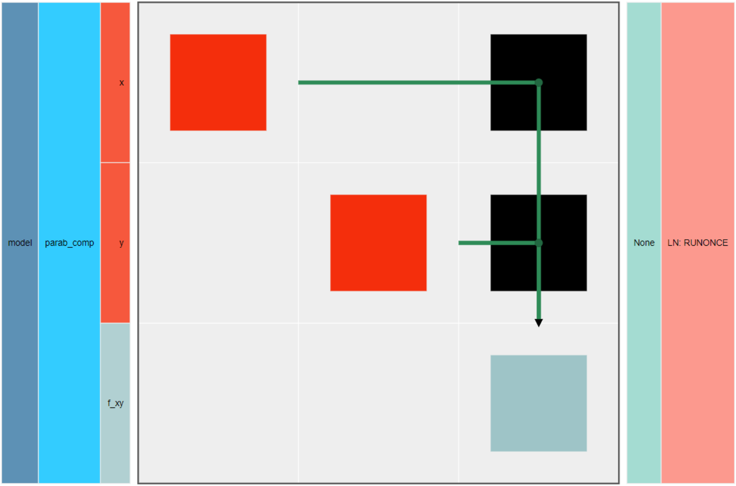
N2 diagram/N-squared diagram:

- Visualize the model hierarchy
- Further reading: ▸ Link ▸ Link

Generating a N2 diagram for the single-discipline problem:

- Open Anaconda Prompt
- Direct to location where mda_single_disp.py is stored
- Use the command: openmdao n2 mda_single_disp.py
- And/or add the following to the end of code:
  - "from openmdao.api import n2; n2(prob)"
- Diagram is displayed in browser

Single-discipline analysis



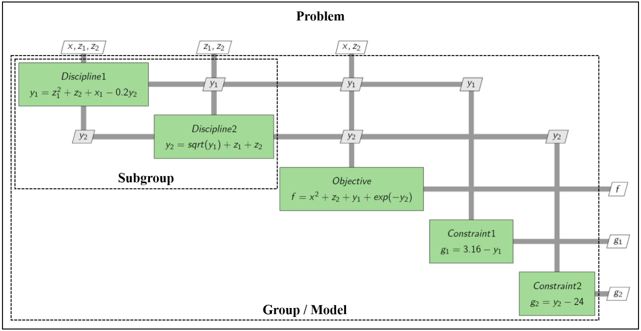
Two-discipline analysis

The Sellar problem with two disciplines and one scalar input:

$$
\begin{aligned}
\text{Objective function}: &\quad f = x^2 + z_2 + y_1 + \exp(-y_2) \\
\text{Discipline 1}: &\quad y_1 = z_1^2 + z_2 + x_1 - 0.2 y_2 \\
\text{Discipline 2}: &\quad y_2 = \sqrt{y_1} + z_1 + z_2 \\
\text{Constraint 1}: &\quad g_1 = 3.16 - y_1 \leq 0 \\
\text{Constraint 2}: &\quad g_2 = y_2 - 24.0 \leq 0
\end{aligned}
$$

Two-discipline analysis

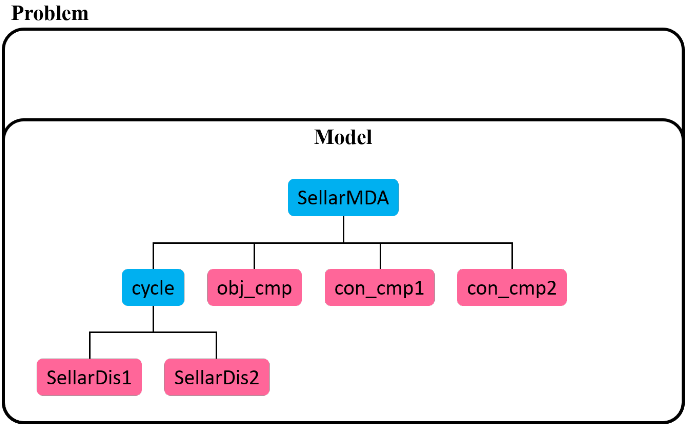
The subgroup uses nonlinear solver to solve themselves iteratively for convergence



The XDSM code can be found here: Link

For more detailed explanation of the setup: Link

Two-discipline analysis



Two-discipline analysis

- Download code from Github: Link
- Open mda_sellar.py in Spyder
- Part 1: Import required packages
- Part 2: Create new components for Discipline1 and 2

```python
# Part 1: Import required packages
import openmdao.api as om
import numpy as np

# Part 2: Create new components for Discipline1 and 2
class SellarDis1(om.ExplicitComponent):
    """
    Component containing Discipline 1 -- no derivatives version.
    """
    def setup(self):
        # Global Design Variable
        self.add_input('z', val=np.zeros(2))

        # Local Design Variable
        self.add_input('x', val=0.)

        # Coupling parameter
        self.add_input('y2', val=1.0)

        # Coupling output
        self.add_output('y1', val=1.0)

        # Finite difference all partials.
        self.declare_partials('*', '*', method='fd')

    def compute(self, inputs, outputs):
        """
        Evaluates the equation
        y1 = z1**2 + z2 + x1 - 0.2*y2
        """
        z1 = inputs['z'][0]
        z2 = inputs['z'][1]
        x1 = inputs['x']
        y2 = inputs['y2']
```

Two-discipline analysis

- Part 3: Create group SellarMDA
  - SellarMDA group hold subgroup cycle and other components obj_cmp(f), con_cmp1 (g_1), and con_cmp2 (g_2)
  - cycle group holds discipline1 and discipline2

```python
# Part 3: Create group SellarMDA
class SellarMDA(om.Group):
    """
    Group containing the Sellar MDA.
    """
    def setup(self):
        indeps = self.add_subsystem('indeps', om.IndepVarComp(), promotes=['*'])
        indeps.add_output('x', 1.0)
        indeps.add_output('z', np.array([5.0, 2.0]))

        cycle = self.add_subsystem('cycle', om.Group(), promotes=['*'])
        cycle.add_subsystem('d1', SellarDis1(), promotes_inputs=['x', 'z', 'y2'],
                            promotes_outputs=['y1'])
        cycle.add_subsystem('d2', SellarDis2(), promotes_inputs=['z', 'y1'],
                            promotes_outputs=['y2'])

        # Nonlinear Block Gauss Seidel is a gradient free solver
        cycle.nonlinear_solver = om.NonlinearBlockGS(iprint=1)  # try iprint=2

        self.add_subsystem('obj_cmp', om.ExecComp('obj = x**2 + z[1] + y1 + exp(-y2)',
                                                  z=np.array([0.0, 0.0]), x=0.0),
                           promotes=['x', 'z', 'y1', 'y2', 'obj'])

        self.add_subsystem('con_cmp1', om.ExecComp('con1 = 3.16 - y1'), promotes=['con1', 'y1'])
        self.add_subsystem('con_cmp2', om.ExecComp('con2 = y2 - 24.0'), promotes=['con2', 'y2'])
```

Two-discipline analysis

- Part 4: Setup model and problem
- Part 5: Provide input to the problem and run the model
- Part 6: Print problem details

```python
# Part 4: Setup model and problem
prob = om.Problem()
prob.model = SellarMDA()
prob.setup()

# Part 5: Provide input to the problem
prob['x'] = 2.
prob['z'] = [-1., -1.]

prob.run_model()

#  Part 6:  print details
print('\nInput ---')
print('x :',prob['x'])
print('z1 :',prob['z'][0])
print('z2 :',prob['z'][1])

print('\nDiscipline output ---')
print('y1 :',prob['y1'])
print('y2 :',prob['y2'])

print('\nObjective and constraints---')
print('obj :',prob['obj'])
print('con1 :',prob['con1'])
print('con2 :',prob['con2'])
```

- For more detailed explanation of the setup: Link

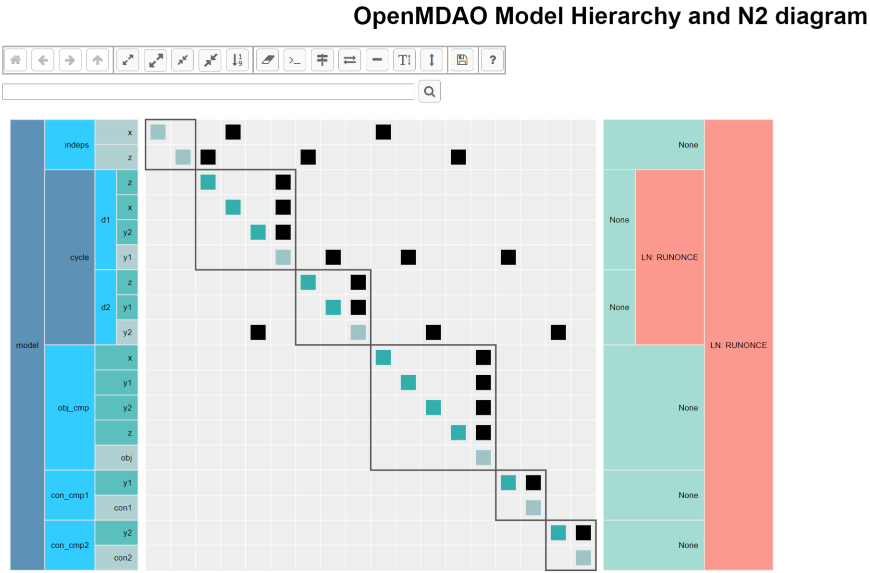
Two-discipline analysis

- Input details: $x = 2, z_1 = -1, z_2 = -1$

```
=====
cycle
=====
NL: NLBGS Converged in 9 iterations

Input ---
x : [2.]
z1 : -1.0
z2 : -1.0

Discipline output ---
y1 : [2.10951651]
y2 : [-0.54758253]

Objective and constraints---
obj : [6.8385845]
con1 : [1.05048349]
con2 : [-24.54758253]

In [3]:
```

IPython console History log

Two-discipline analysis



Further reading

- Basic user guide: Single-Discipline Optimization ▸ Link
- Nonlinear Solvers (NonlinearBlockGS): ▸ Link