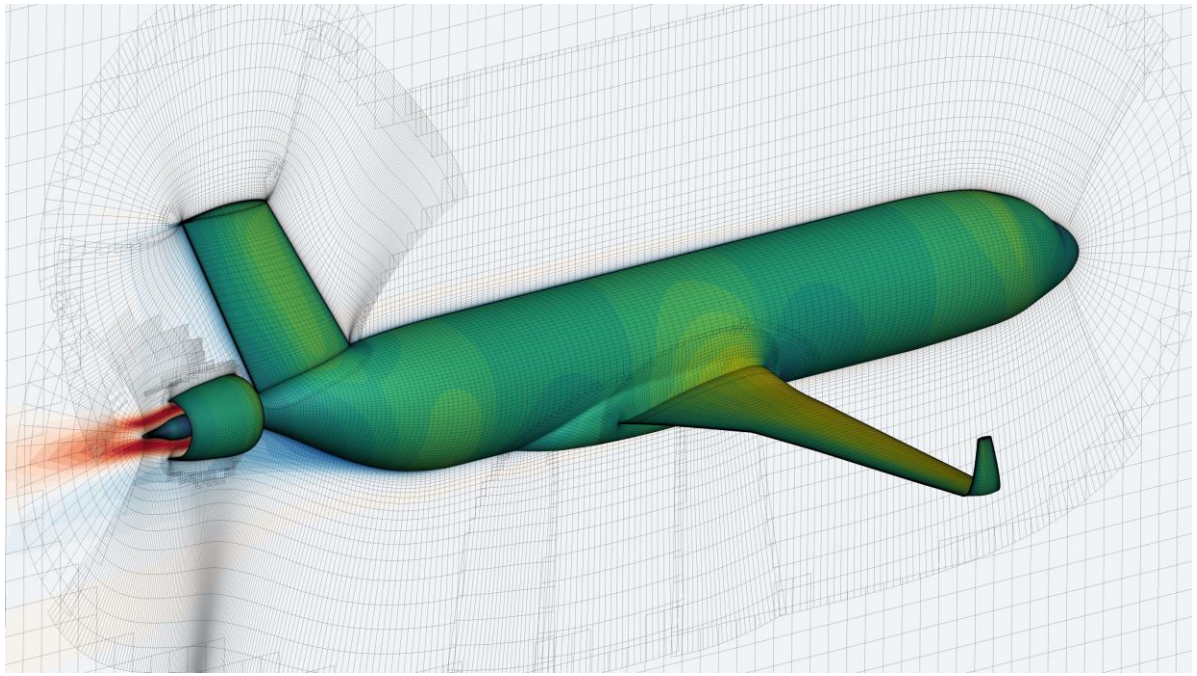
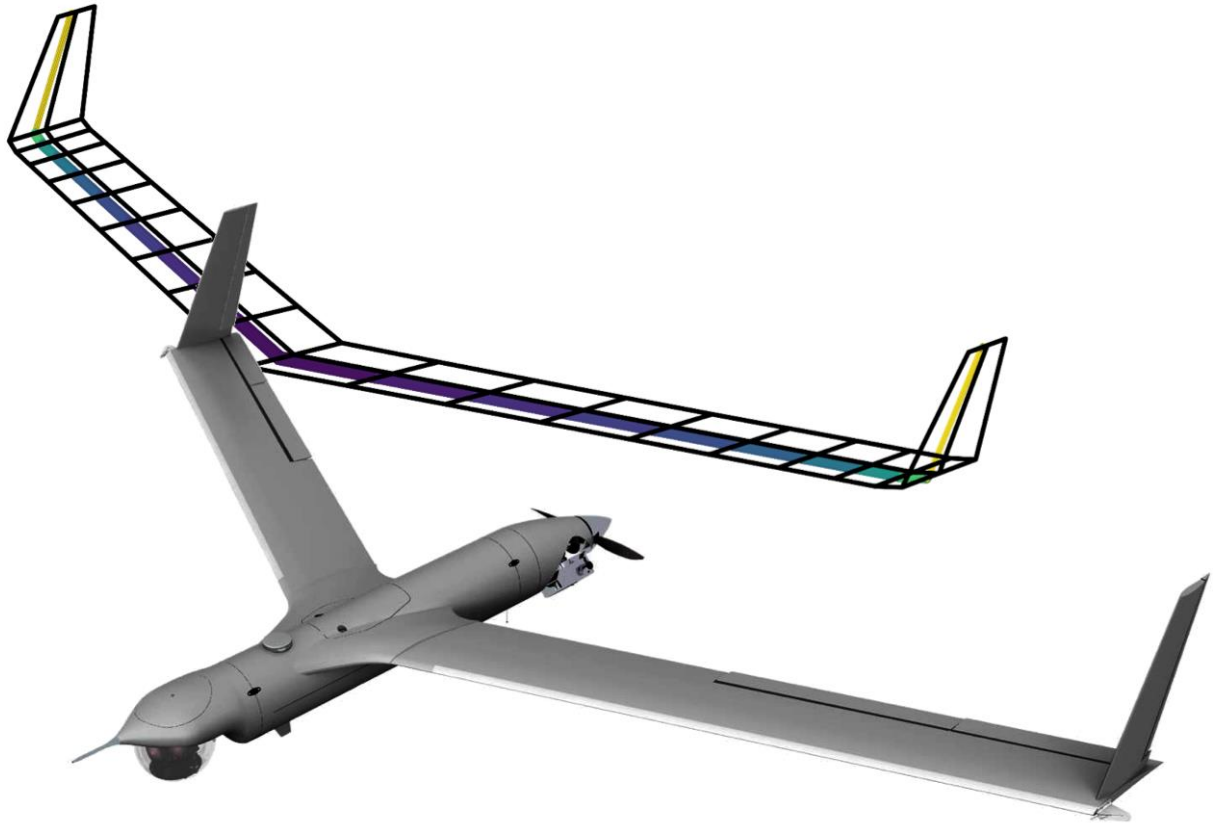
Getting started with OpenMDAO





John Jasa johnjasa@umich.edu

Shamsheer Chauhan sschau@umich.edu

Joaquim R.R.A. Martins jrram@umich.edu

University of Michigan MDO Lab

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Acknowledgements

- **Ben Brelje,** PhD candidate at University of Michigan MDO Lab for creating version 0 of this training

- **Justin Gray,** Engineer and team lead of OpenMDAO at NASA Glenn for refining this workshop

- NASA ARMD's TTT Project has supported OpenMDAO development since 2008

Download these slides and tutorial scripts from GitHub

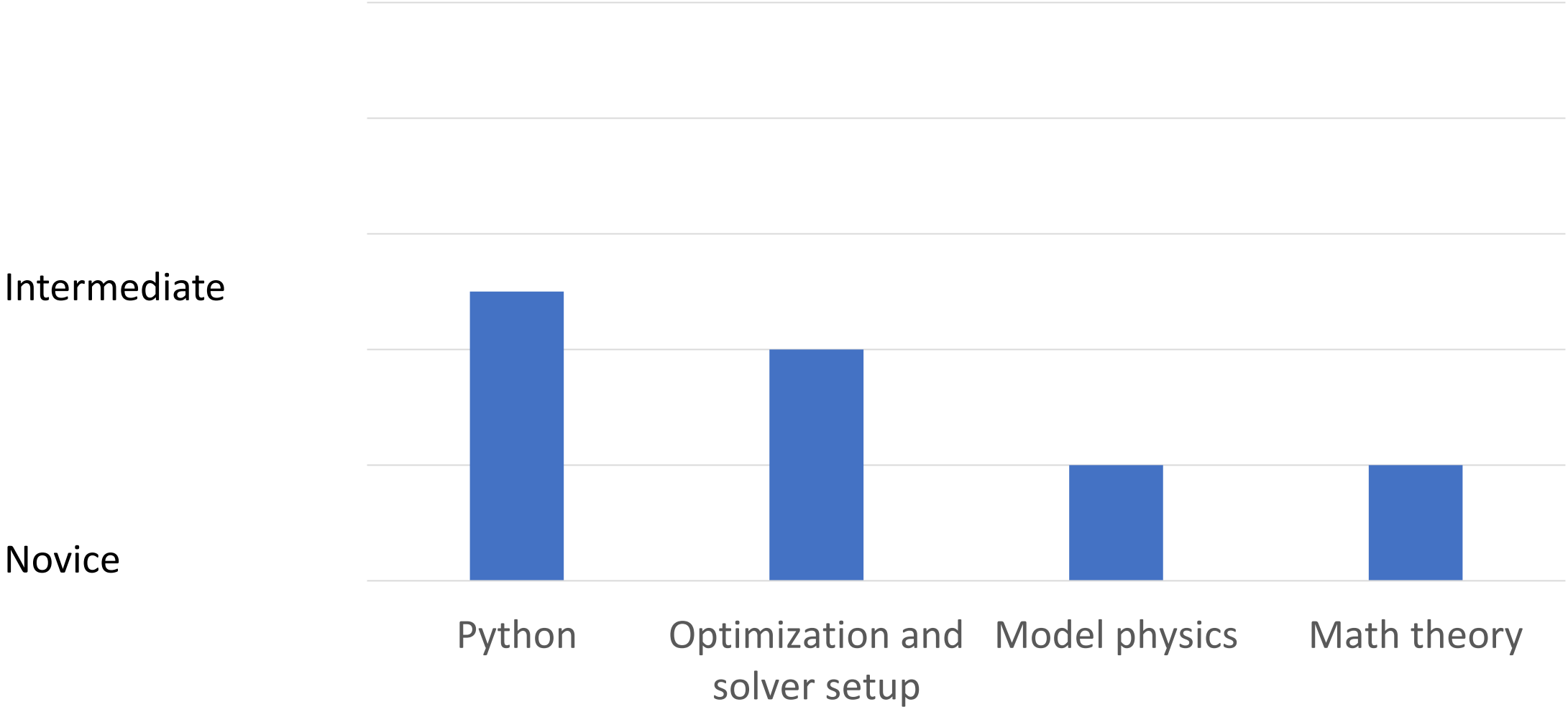
https://github.com/johnjasa/openmdao_training

The OpenMDAO and MAUD papers are worthwhile references

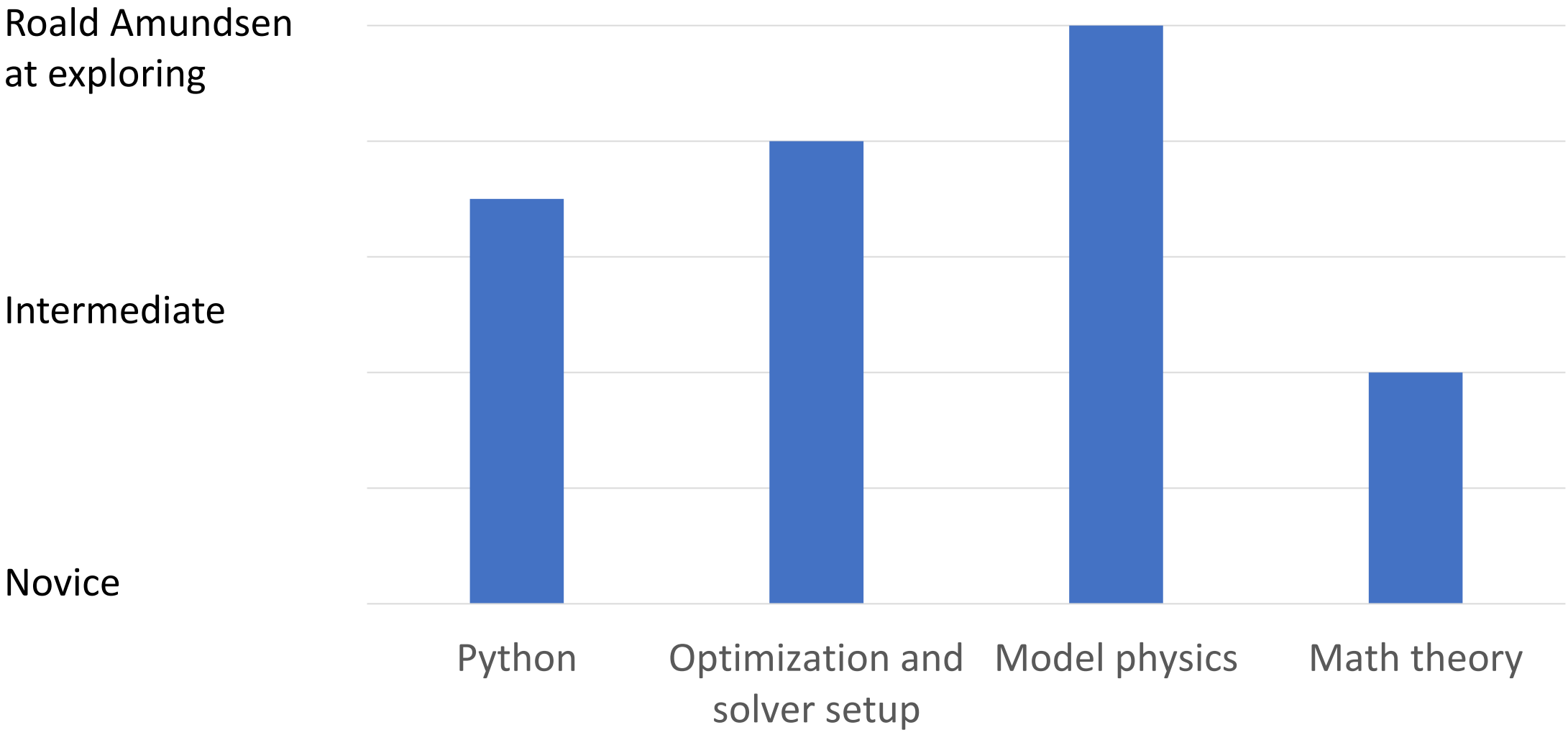
J. S. Gray, J. T. Hwang, J. R. R. A. Martins, K. T. Moore, and B. A. Naylor, “OpenMDAO: An Open-Source Framework for Multidisciplinary Design, Analysis, and Optimization,” Structural and Multidisciplinary Optimization, 2019.

J. T. Hwang and J. R. R. A. Martins, “A computational architecture for coupling heterogeneous numerical models and computing coupled derivatives,” ACM TOMS, 2018.

Levels of expertise suggested to use OpenMDAO



Levels of expertise suggested to use OpenMDAO like a pro

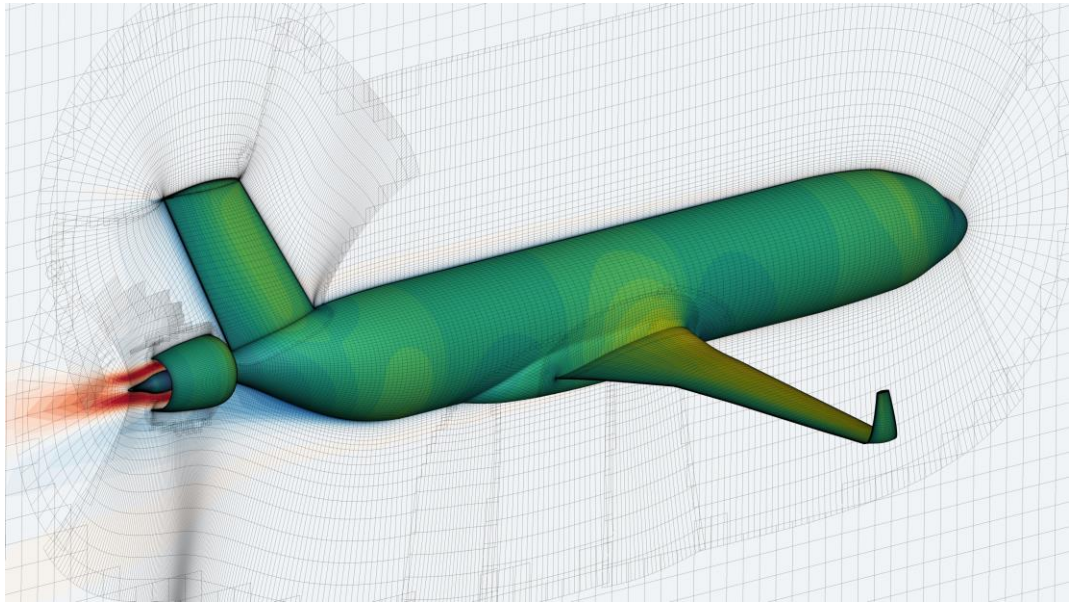


OpenMDAO is an open-source framework for efficient MDAO

- Developed and supported a team at NASA Glenn since 2008
- Apache 2.0 license is very permissive (no “copyleft”)
- Fast enough for high-fidelity, expensive problems but easy enough for cheap conceptual models
- Can be used as a framework,
  or a low level library for building stand alone codes

OpenMDAO is a reasonable choice for a wide array of MDAO problems

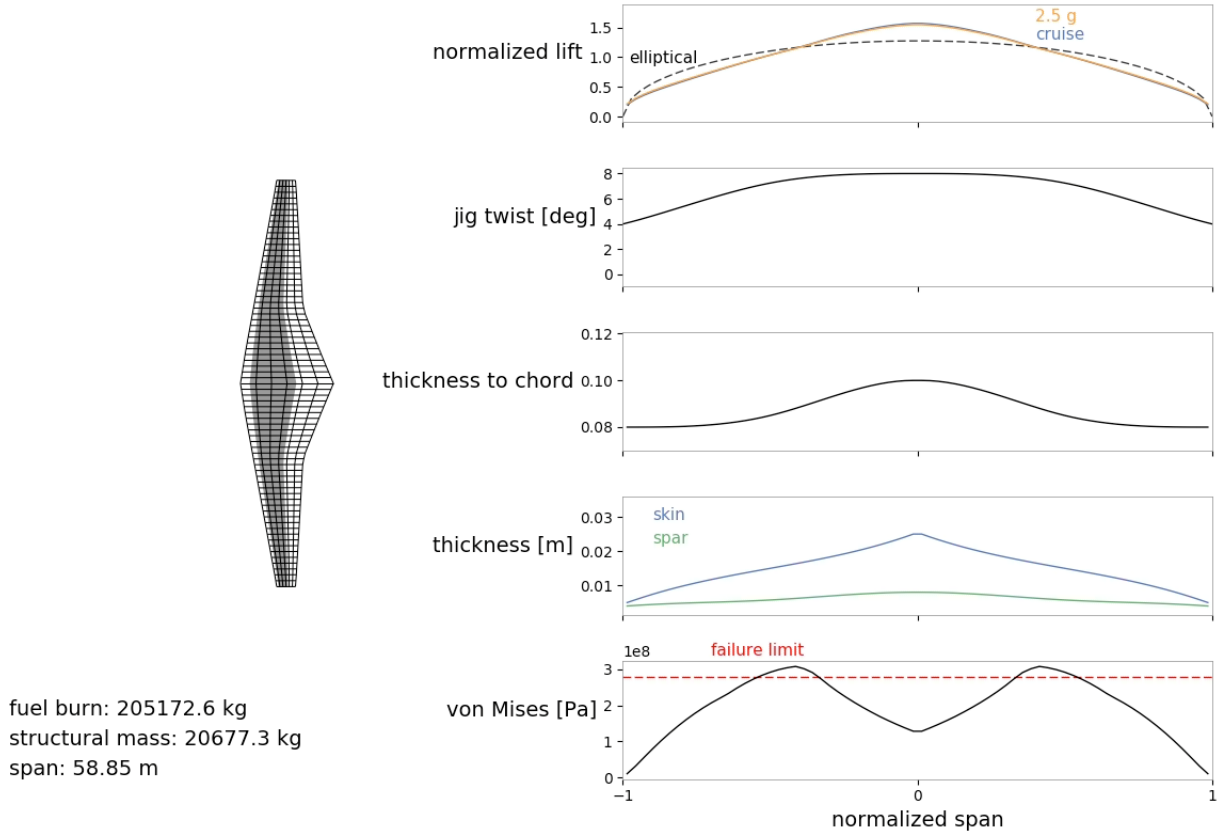
- High-fidelity aeropropulsive optimization with RANS and CEA



Gray et al., 2018, AIAA Aviation

OpenMDAO is a reasonable choice for a wide array of MDAO problems

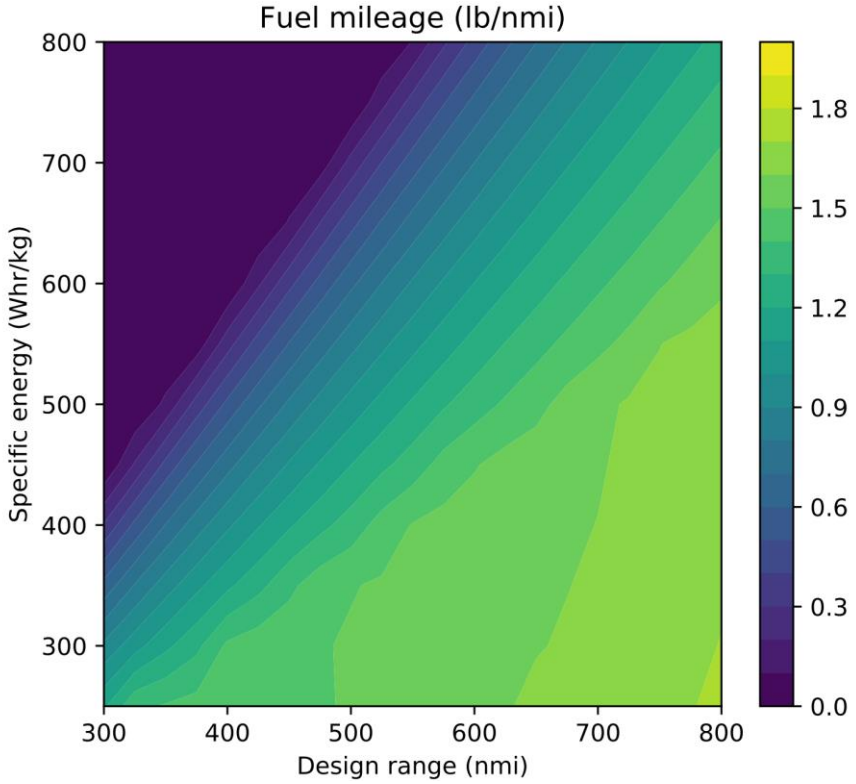
- High-fidelity aeropropulsive optimization with RANS and CEA
- Medium-fidelity aerostructural optimization (VLM/beam)



[Chauhan and Martins, SMO 2019]

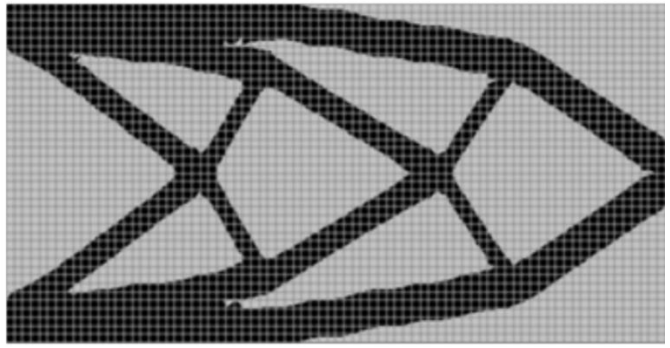
OpenMDAO is a reasonable choice for a wide array of MDAO problems

- High-fidelity aeropropulsive optimization with RANS and CEA
- Medium-fidelity aerostructural optimization (VLM/beam)
- Conceptual-fidelity sizing and tradespace exploration

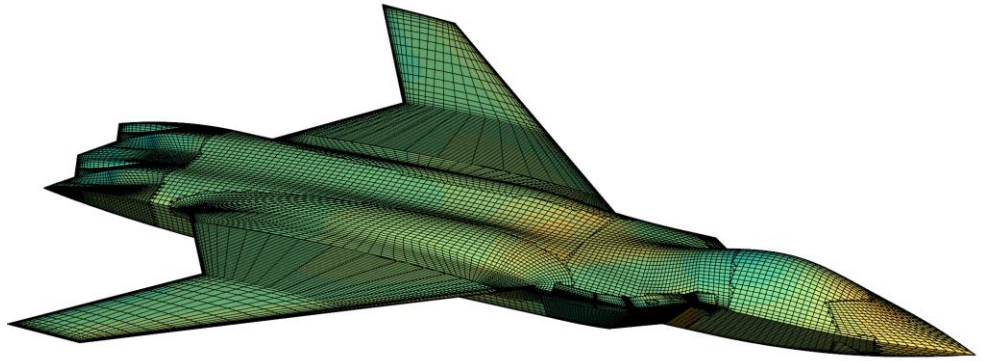


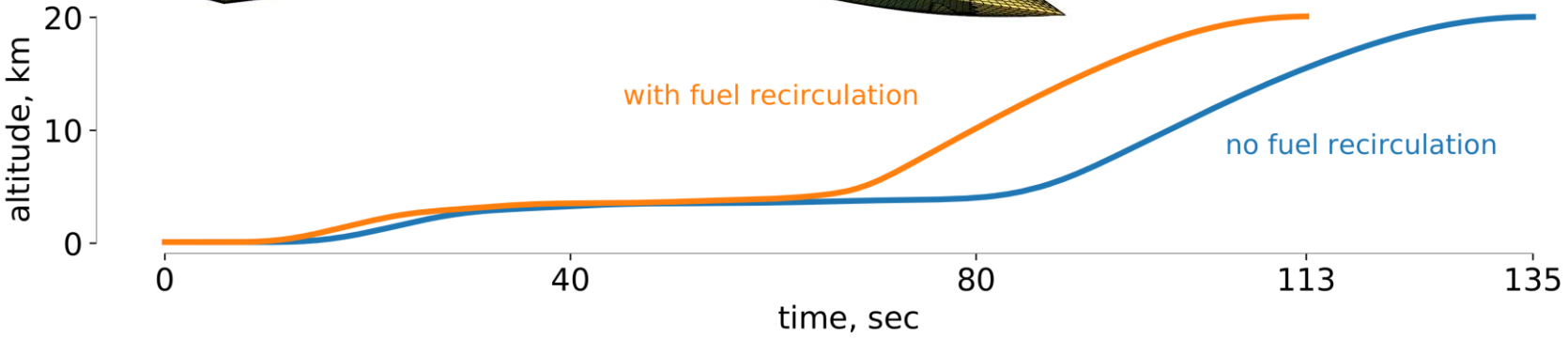
Brelje and Martins, EATS 2018

OpenMDAO is a reasonable choice for a wide array of MDAO problems

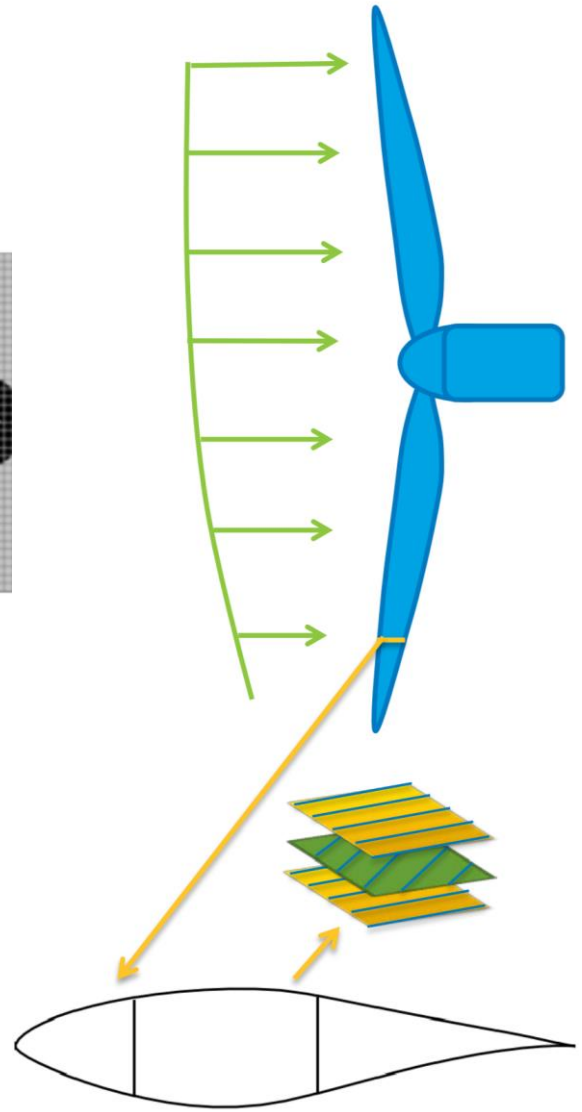


Chung et al., 2019, SMO



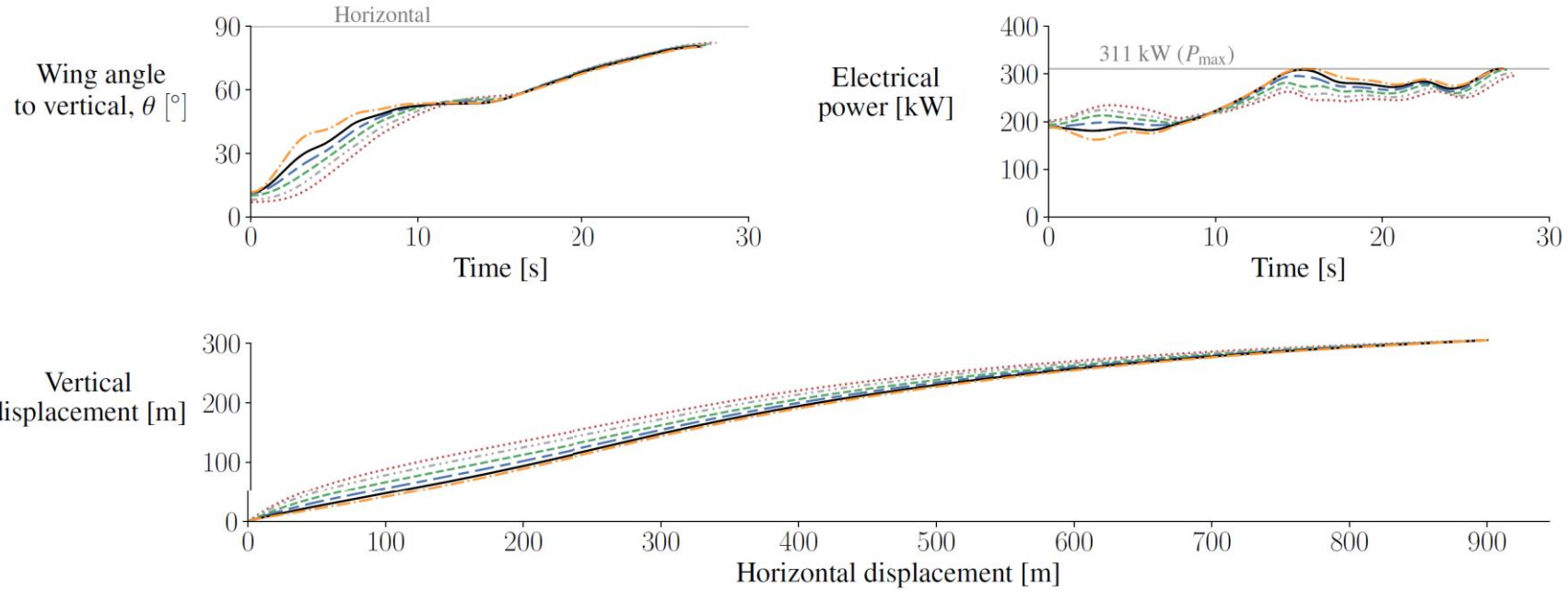


Jasa et al., 2018, AIAA Aviation

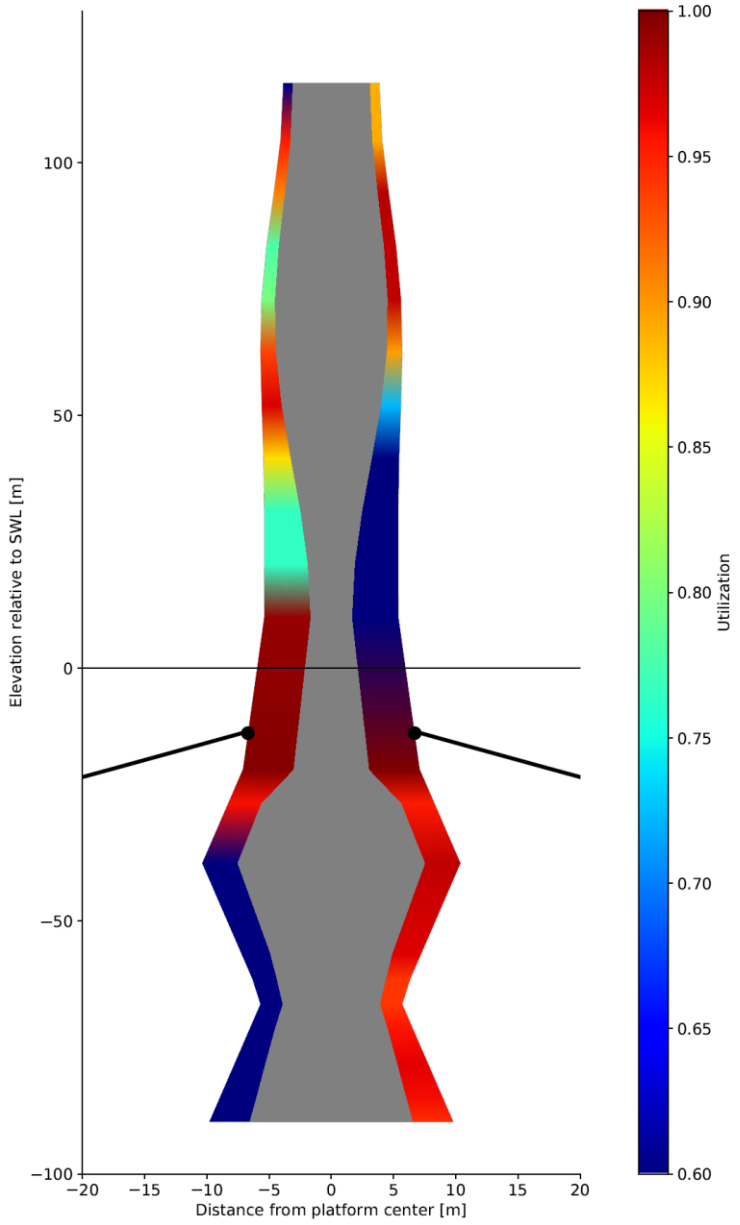


Dykes et al., 2015
NREL presentation

OpenMDAO is a reasonable choice for a wide array of MDAO problems



Chauhan and Martins, Journal of Aircraft 2019



Hegseth and Bachynski, 2019

Goals for today

- Learn enough to run a useful OpenMDAO model
- Gain some intuition about the operating theory
- Get excited about using analytic derivatives for optimization

Brief introduction to optimization terms

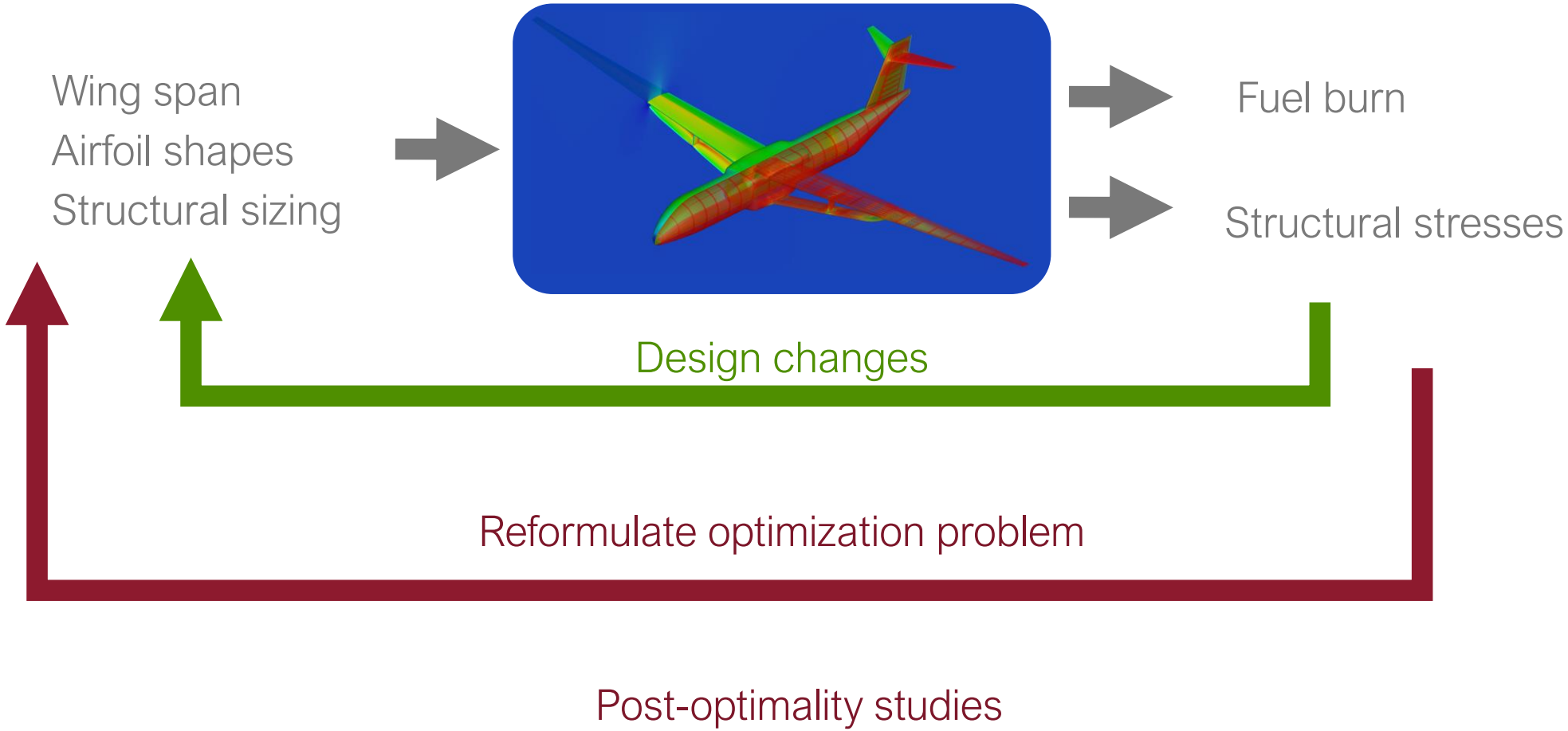
- What is an optimization problem?
- Gradient-free vs gradient-based optimizers
- How to get derivatives

Numerical optimization provides a way to fully automate the design process

Design optimization problem:

minimize	f(x)	objective
with respect to	x	design variables
subject to	c(x) ≤ 0	constraints

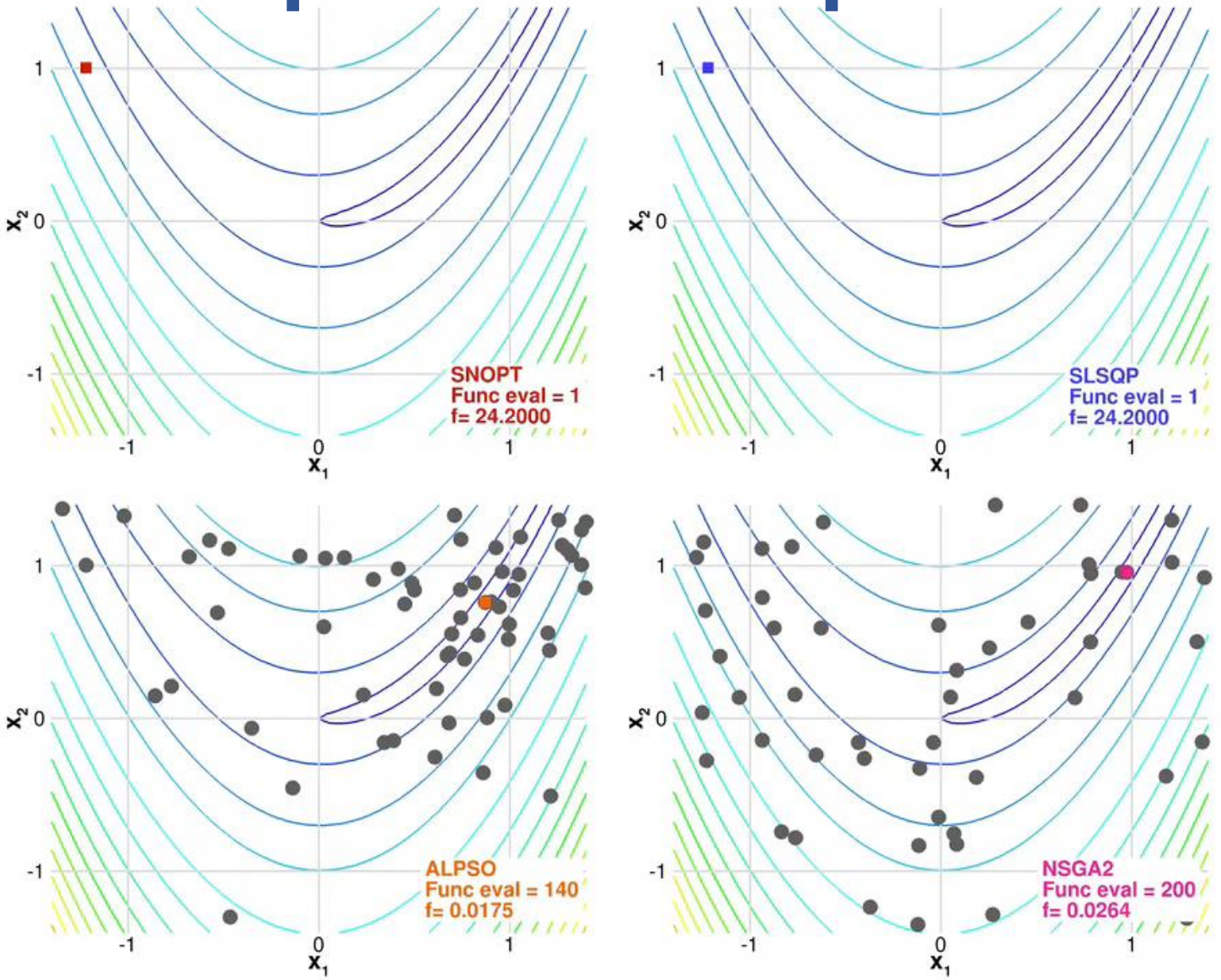
Numerical optimization provides a way to fully automate the design process



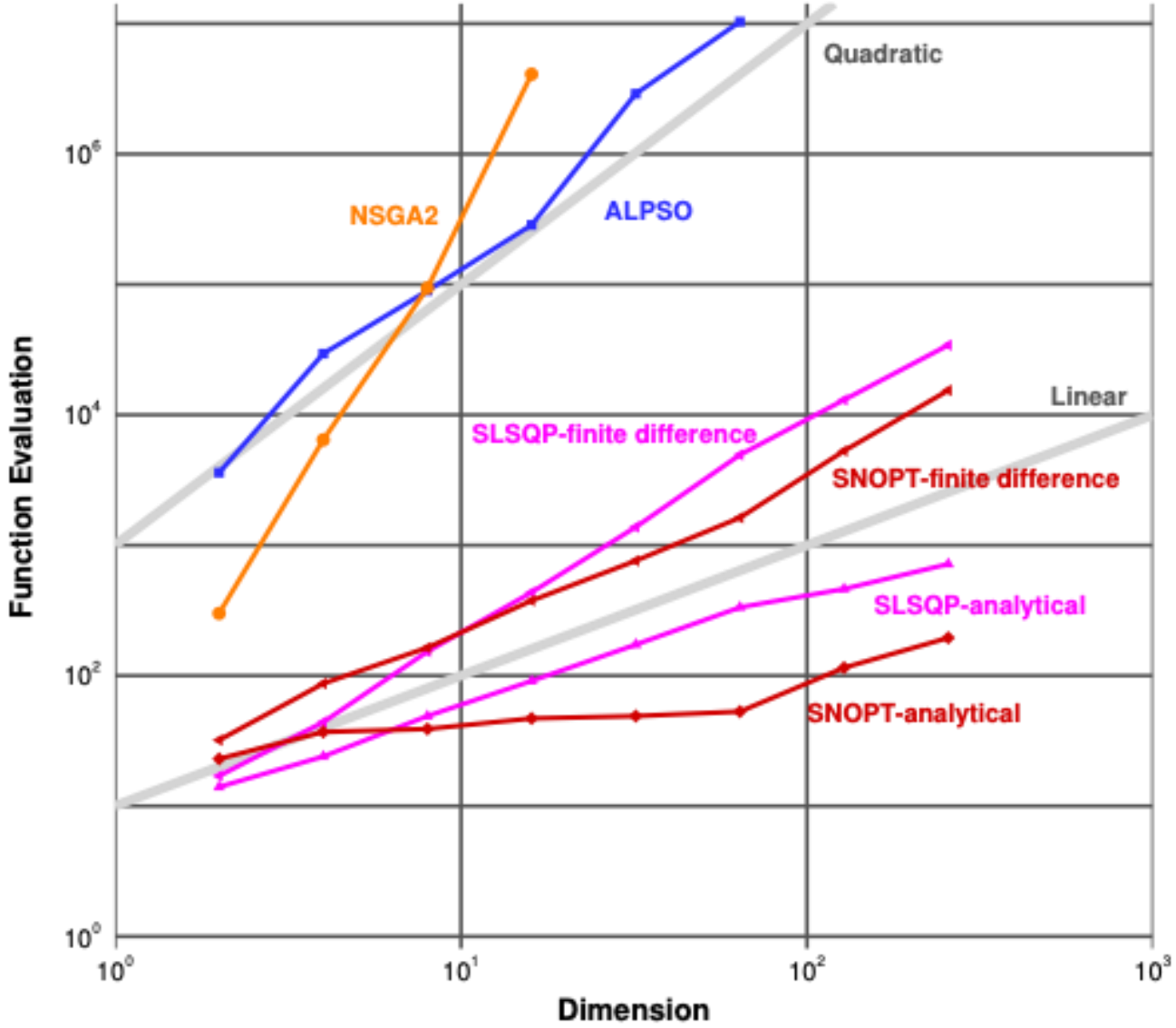
Gradient-based methods take a more direct path to the optimum

Gradient-based

Gradient-free



Gradient-based optimization is the only hope for large numbers of design variables



[Lyu et al. *ICCFD8-2014-0203*]

Methods for computing derivatives

Monolithic ***Black boxes*** ***input and outputs***	Finite-differences	$\frac{df}{dx_j} = \frac{f(x_j + h) - f(x)}{h} + \mathcal{O}(h)$
	Complex-step	$\frac{df}{dx_j} = \frac{\mathrm{Im}\left[f(x_j + ih)\right]}{h} + \mathcal{O}(h^2)$
Analytic ***Governing eqns*** ***state variables***	Direct Adjoint	$\frac{df}{dx} = \frac{\partial f}{\partial x} - \underbrace{\overbrace{\frac{\partial f}{\partial y}\left[\frac{\partial R}{\partial y}\right]^{-1}}^{-\,dy/\,dx}\frac{\partial R}{\partial x}}_{\psi}$
Algorithmic differentiation ***Lines of code*** ***code variables***	Forward Reverse	$\begin{bmatrix} 1 & 0 & \dots & 0 \\ -\frac{\partial T_2}{\partial t_1} & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ -\frac{\partial T_n}{\partial t_1} & \dots & -\frac{\partial T_n}{\partial t_{n-1}} & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & \dots & 0 \\ \frac{dt_2}{dt_1} & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ \frac{dt_n}{dt_1} & \dots & \frac{dt_n}{dt_{n-1}} & 1 \end{bmatrix} = \boldsymbol{I} = \begin{bmatrix} 1 & -\frac{\partial T_2}{\partial t_1} & \dots & -\frac{\partial T_n}{\partial t_1} \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & -\frac{\partial T_n}{\partial t_{n-1}} \\ 0 & \dots & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & \frac{dt_2}{dt_1} & \dots & \frac{dt_n}{dt_1} \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & 1 & \frac{dt_n}{dt_{n-1}} \\ 0 & \dots & 0 & 1 \end{bmatrix}$

[Martins et al., *ACM TOMS*, 2003]

[Martins and Hwang, *AIAA Journal*, 2013]

Overview of today's activities

- OpenMDAO intro and basics
  - Lab 0: Explicit components and connections
- Using solvers with implicit models
  - Lab 1: Comparing gradient-free and gradient-based solvers
- Optimization with and without analytic derivatives
  - Lab 2: Optimizing the thickness distribution of a simple beam
- Wrapping external codes
  - Lab 3: Wrapping external codes as explicit and implicit components

OpenMDAO intro and Basics

- Intro and terminology
- Building explicit components and connecting them together
- Lab 0: Implementing a simple explicit calculation (Breguet Range)

OpenMDAO has nice features

- Units
  - Conversions
  - (In)compatibility checks

OpenMDAO has nice features

- Units
  - Conversions
  - (In)compatibility checks
- Automatic checks for unconnected inputs

OpenMDAO has nice features

- Units
  - Conversions
  - (In)compatibility checks
- Automatic checks for unconnected inputs
- Interactive N2 diagrams

OpenMDAO has nice features

- Units
  - Conversions
  - (In)compatibility checks
- Automatic checks for unconnected inputs
- Interactive N2 diagrams
- Models are Python objects (inheritance!)

OpenMDAO has advanced numerical methods

- Efficient solvers for implicit systems
  - Newton solver
  - Nonlinear block Gauss-Seidel

OpenMDAO has advanced numerical methods

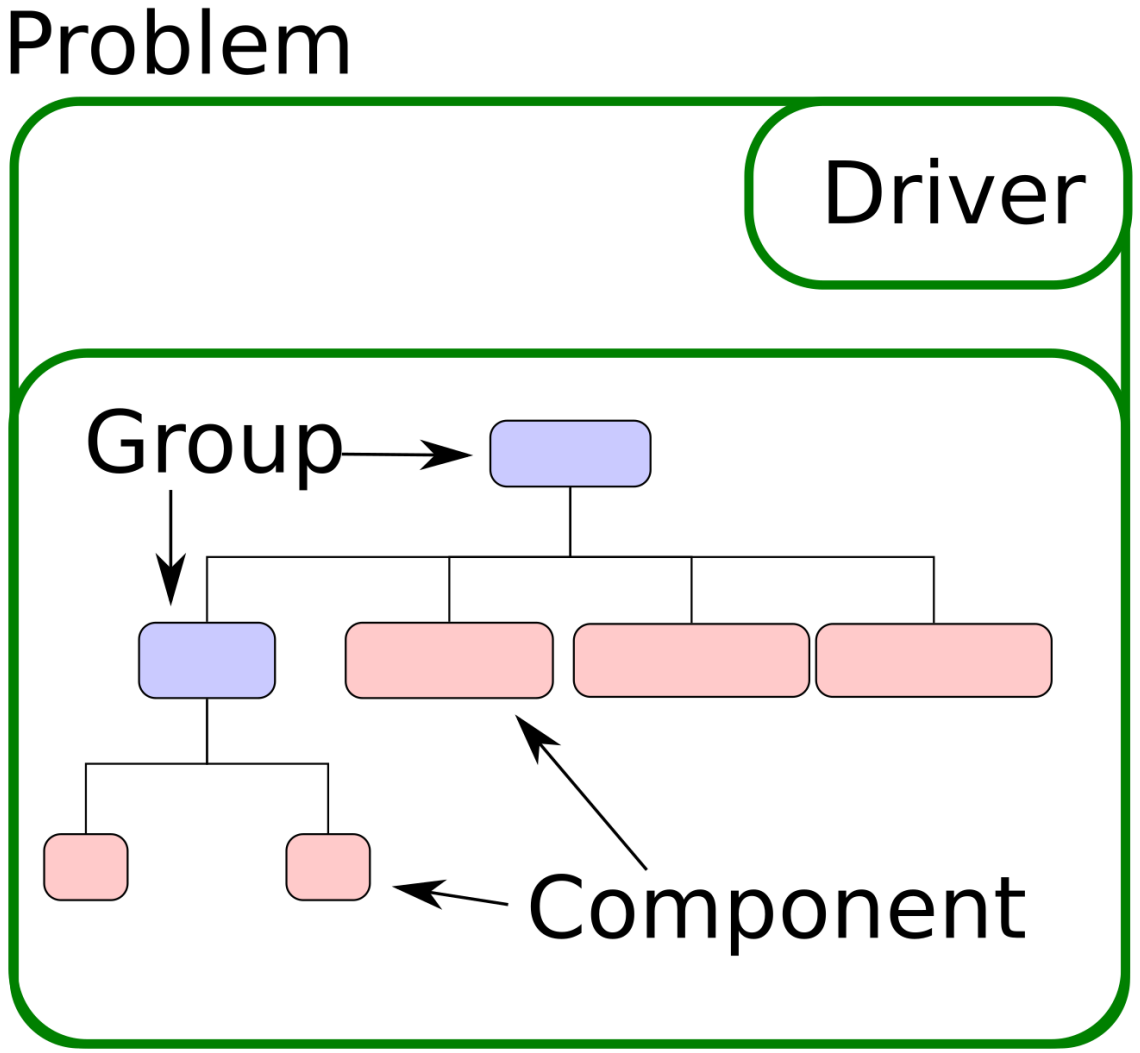
- Efficient solvers for implicit systems
  - Newton solver
  - Nonlinear block Gauss-Seidel
- Derivative computation (for gradient-based opt.)
  - Forward and reverse analytic
  - Finite difference or complex step
  - Mixture of all!

OpenMDAO has advanced numerical methods

- Efficient solvers for implicit systems
  - Newton solver
  - Nonlinear block Gauss-Seidel
- Derivative computation (for gradient-based opt.)
  - Forward and reverse analytic
  - Finite difference or complex step
  - Mixture of all!
- MPI parallelization

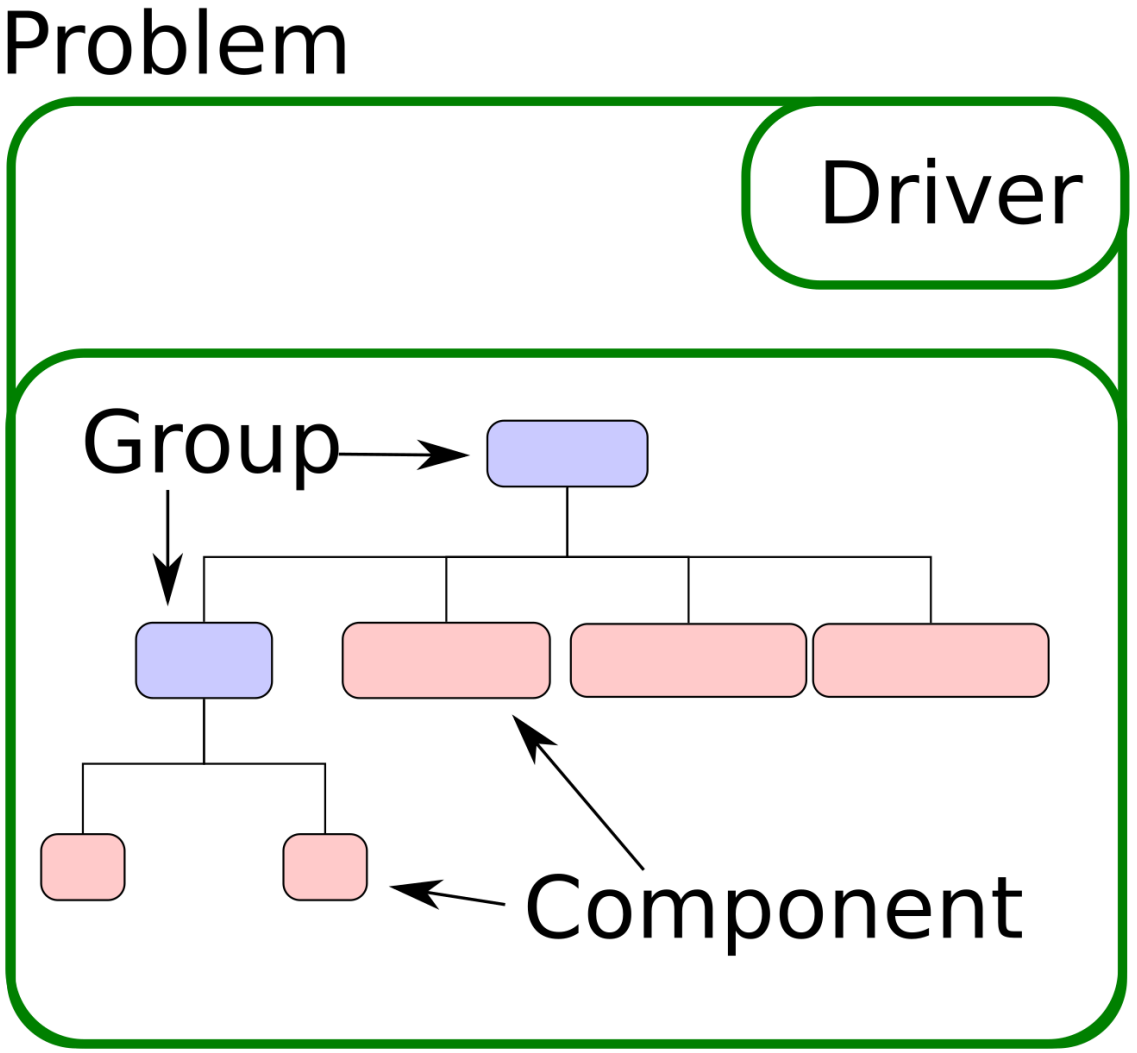
OpenMDAO problem hierarchy

- **Components** implement the actual model computation



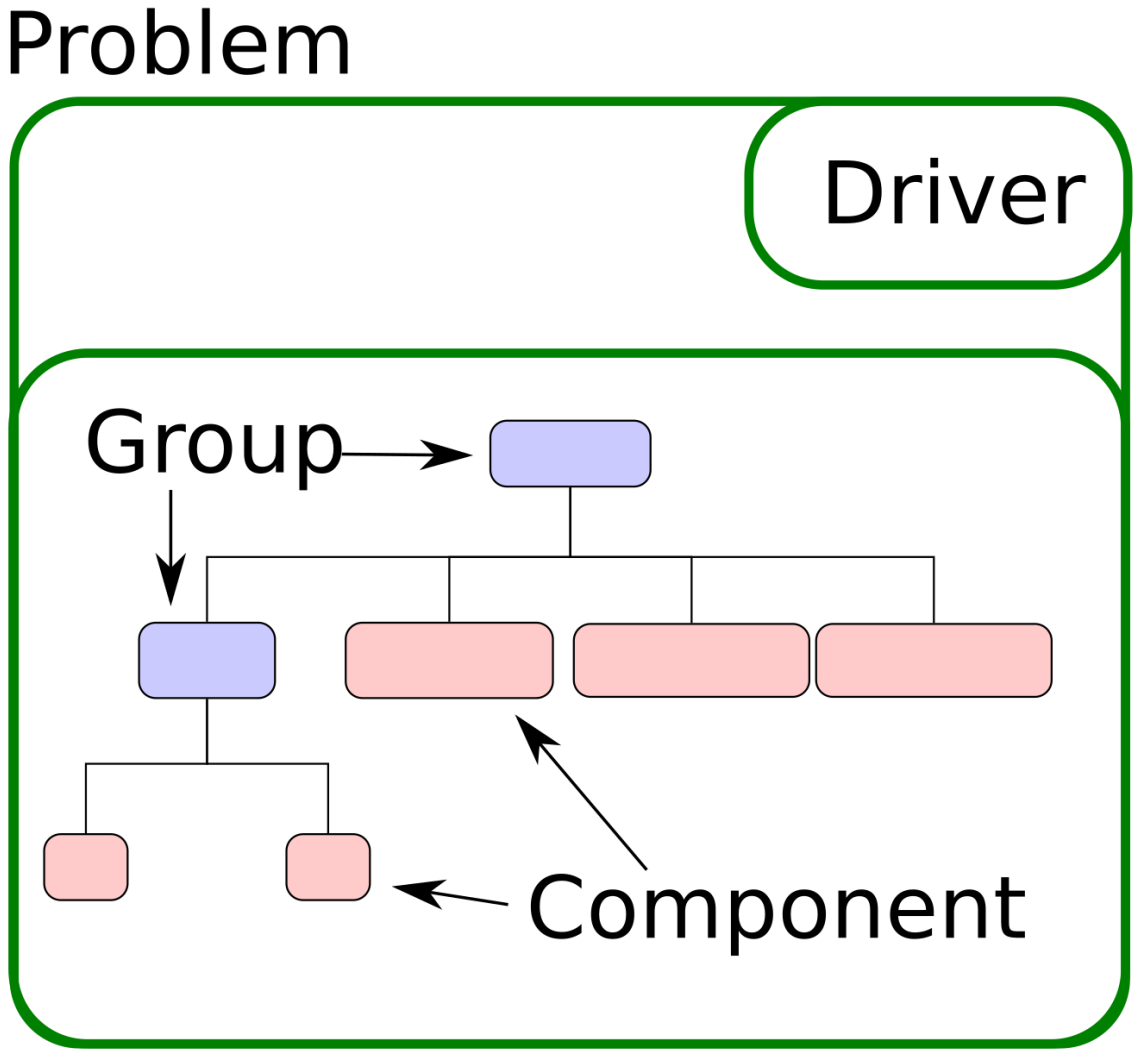
OpenMDAO problem hierarchy

- Components implement the actual model computation
- **Groups** organize the model and enable hierarchical solver strategies



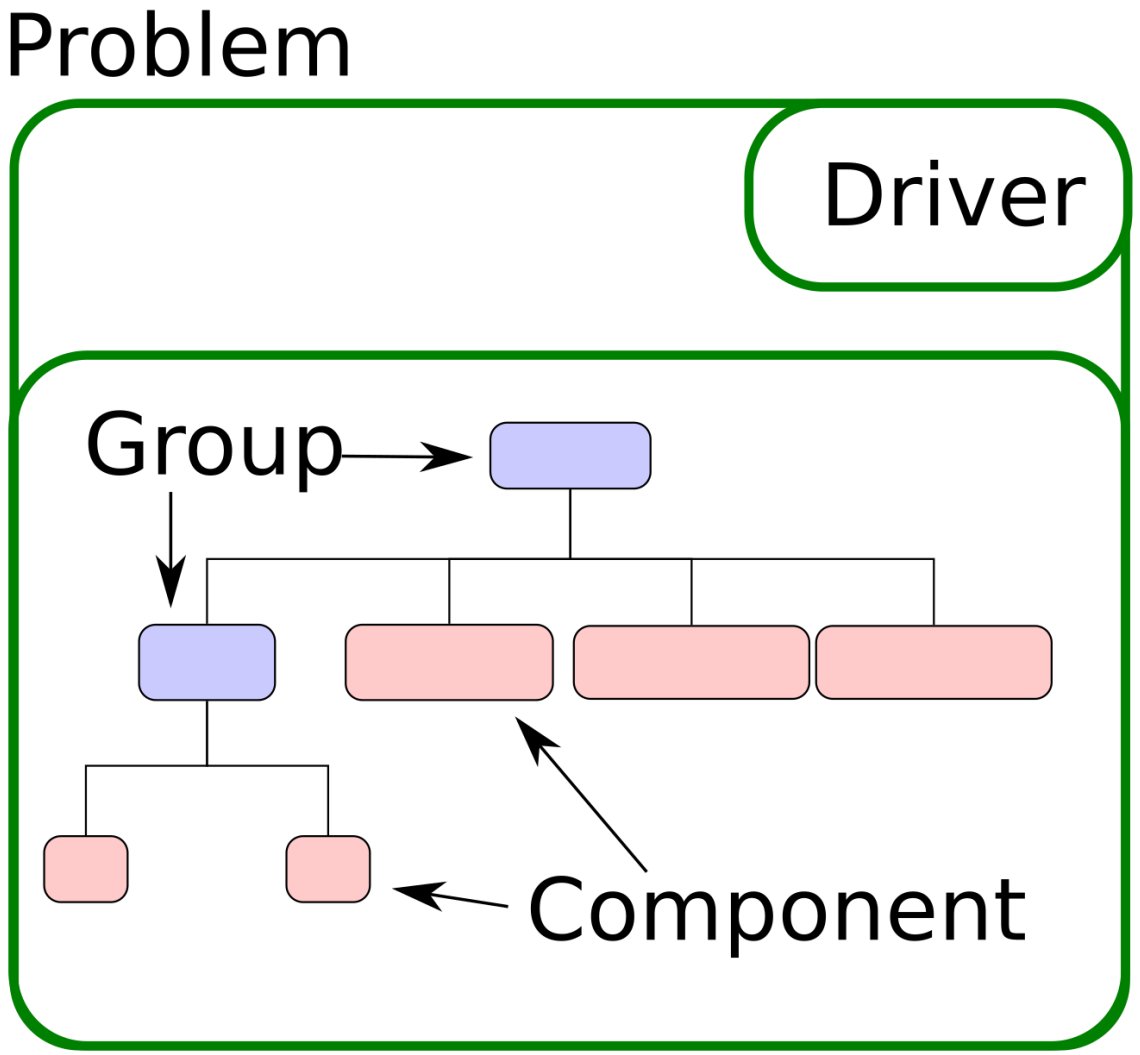
OpenMDAO problem hierarchy

- Components implement the actual model computation
- Groups organize the model and enable hierarchical solver strategies
- **Drivers** iteratively execute the model (optimizers and DOEs)



OpenMDAO problem hierarchy

- Components implement the actual model computation
- Groups organize the model and enable hierarchical solver strategies
- Drivers iteratively execute the model (optimizers and DOEs)
- **Problem** is the top-level container

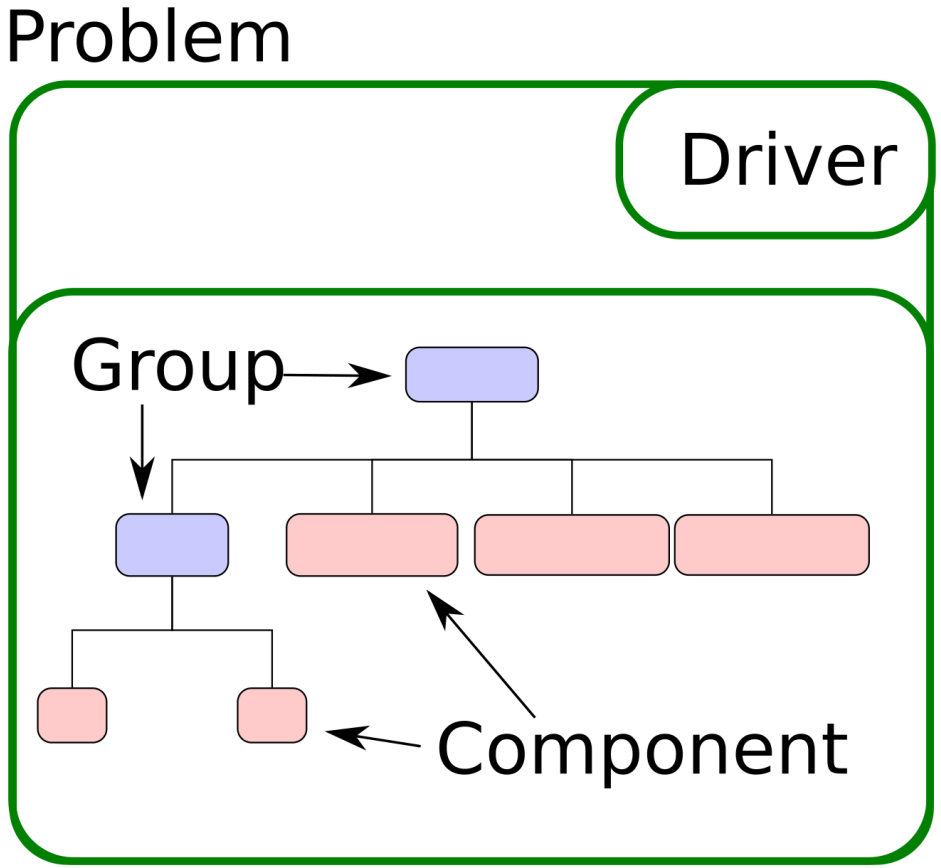


Why use classes in object-oriented programming?

- Convenient containers of data and methods
- It allows inheritance of features
- Can reuse component and group templates easily

ExplicitComponent class

- Used for doing explicit calculations
- Inputs → Outputs
- Can be as simple as a one-line calculation, or as complex as an adjoint CFD solver



ExplicitComponent example

```
class ComputeLift(ExplicitComponent):
    """Compute lift on a wing"""

    def initialize(self):
        self.options.declare('num_pts', default=1, desc="n analysis pts")

    def setup(self):
        npts = self.options['num_pts']
        # Inputs
        self.add_input('Sref', 10.0, units="m**2", desc="Wing ref area")
        self.add_input('rho', 1.225, units="kg/m**3", shape=(npts, ), desc="Air density")
        self.add_input('U', 80.0, units="m/s", shape=(npts, ), desc="Airspeed")
        self.add_input('CL', 0.5, units=None, shape=(npts, ), desc="Lift coefficient")

        # Outputs
        self.add_output('L', 0.0, units="N", shape=(npts, ), desc="Lift force")

        # Partial derivatives
        self.declare_partials('L', ['*'])

    def compute(self, inputs, outputs):
        q = 1 / 2 * inputs['rho'] * inputs['U'] ** 2
        outputs['L'] = q * inputs['Sref'] * inputs['CL']
```

$$L = \frac{1}{2}\rho U^2 S_{ref} C_L$$

ExplicitComponent example

```
class ComputeLift(ExplicitComponent):
    """Compute lift on a wing"""

    def initialize(self):
        self.options.declare('num_pts', default=1, desc="n analysis pts")

    def setup(self):
        npts = self.options['num_pts']
        # Inputs
        self.add_input('Sref', 10.0, units="m**2", desc="Wing ref area")
        self.add_input('rho', 1.225, units="kg/m**3", shape=(npts, ), desc="Air density")
        self.add_input('U', 80.0, units="m/s", shape=(npts, ), desc="Airspeed")
        self.add_input('CL', 0.5, units=None, shape=(npts, ), desc="Lift coefficient")

        # Outputs
        self.add_output('L', 0.0, units="N", shape=(npts, ), desc="Lift force")

        # Partial derivatives
        self.declare_partials('L', ['*'])

    def compute(self, inputs, outputs):
        q = 1 / 2 * inputs['rho'] * inputs['U'] ** 2
        outputs['L'] = q * inputs['Sref'] * inputs['CL']
```

All your model components will subclass OpenMDAO base classes

$$L = \frac{1}{2}\rho U^2 S_{ref} C_L$$

ExplicitComponent example

```
class ComputeLift(ExplicitComponent):
    """Compute lift on a wing"""

    def initialize(self):
        self.options.declare('num_pts', default=1, desc="n analysis pts")

    def setup(self):
        npts = self.options['num_pts']
        # Inputs
        self.add_input('Sref', 10.0, units="m**2", desc="Wing ref area")
        self.add_input('rho', 1.225, units="kg/m**3", shape=(npts, ), desc="Air density")
        self.add_input('U', 80.0, units="m/s", shape=(npts, ), desc="Airspeed")
        self.add_input('CL', 0.5, units=None, shape=(npts, ), desc="Lift coefficient")

        # Outputs
        self.add_output('L', 0.0, units="N", shape=(npts, ), desc="Lift force")

        # Partial derivatives
        self.declare_partials('L', ['*'])

    def compute(self, inputs, outputs):
        q = 1 / 2 * inputs['rho'] * inputs['U'] ** 2
        outputs['L'] = q * inputs['Sref'] * inputs['CL']
```

Define any options / run flags which do not change during evaluation

$$L = \frac{1}{2}\rho U^2 S_{ref} C_L$$

ExplicitComponent example

```
class ComputeLift(ExplicitComponent):
    """Compute lift on a wing"""

    def initialize(self):
        self.options.declare('num_pts', default=1, desc="n analysis pts")

    def setup(self):
        npts = self.options['num_pts']
        # Inputs
        self.add_input('Sref', 10.0, units="m**2", desc="Wing ref area")
        self.add_input('rho', 1.225, units="kg/m**3", shape=(npts, ), desc="Air density")
        self.add_input('U', 80.0, units="m/s", shape=(npts, ), desc="Airspeed")
        self.add_input('CL', 0.5, units=None, shape=(npts, ), desc="Lift coefficient")

        # Outputs
        self.add_output('L', 0.0, units="N", shape=(npts, ), desc="Lift force")

        # Partial derivatives
        self.declare_partials('L', ['*'])

    def compute(self, inputs, outputs):
        q = 1 / 2 * inputs['rho'] * inputs['U'] ** 2
        outputs['L'] = q * inputs['Sref'] * inputs['CL']
```

Define model inputs, output, and (optionally) partial derivs using the setup() method. Called once before solve / optimization

$$L = \frac{1}{2}\rho U^2 S_{ref} C_L$$

ExplicitComponent example

```
class ComputeLift(ExplicitComponent):
    """Compute lift on a wing"""

    def initialize(self):
        self.options.declare('num_pts', default=1, desc="n analysis pts")

    def setup(self):
        npts = self.options['num_pts']
        # Inputs
        self.add_input('Sref', 10.0, units="m**2", desc="Wing ref area")
        self.add_input('rho', 1.225, units="kg/m**3", shape=(npts, ), desc="Air density")
        self.add_input('U', 80.0, units="m/s", shape=(npts, ), desc="Airspeed")
        self.add_input('CL', 0.5, units=None, shape=(npts, ), desc="Lift coefficient")

        # Outputs
        self.add_output('L', 0.0, units="N", shape=(npts, ), desc="Lift force")

        # Partial derivatives
        self.declare_partials('L', ['*'])

    def compute(self, inputs, outputs):
        q = 1 / 2 * inputs['rho'] * inputs['U'] ** 2
        outputs['L'] = q * inputs['Sref'] * inputs['CL']
```

$$L = \frac{1}{2}\rho U^2 S_{ref} C_L$$

Do the actual computation using compute(). Need to fill in values for all your declared outputs by the end of this method. Called every time the model is evaluated

ExplicitComponent example

```
class ComputeLift(ExplicitComponent):
    """Compute lift on a wing"""

    def initialize(self):
        self.options.declare('num_pts', default=1, desc="n analysis pts")

    def setup(self):
        npts = self.options['num_pts']
        # Inputs
        self.add_input('Sref', 10.0, units="m**2", desc="Wing ref area")
        self.add_input('rho', 1.225, units="kg/m**3", shape=(npts, ), desc="Air density")
        self.add_input('U', 80.0, units="m/s", shape=(npts, ), desc="Airspeed")
        self.add_input('CL', 0.5, units=None, shape=(npts, ), desc="Lift coefficient")

        # Outputs
        self.add_output('L', 0.0, units="N", shape=(npts, ), desc="Lift force")

        # Partial derivatives
        self.declare_partials('L', ['*'])

    def compute(self, inputs, outputs):
        q = 1 / 2 * inputs['rho'] * inputs['U'] ** 2
        outputs['L'] = q * inputs['Sref'] * inputs['CL']
```

Variable name

Default value (before model runs)

Units (need not match upstream)

Human-readable description (optional)

Variable dimension (in this case, a n x 1 vector)

Will default to expecting user provided analytic derivatives, but we can specify `fd` later…

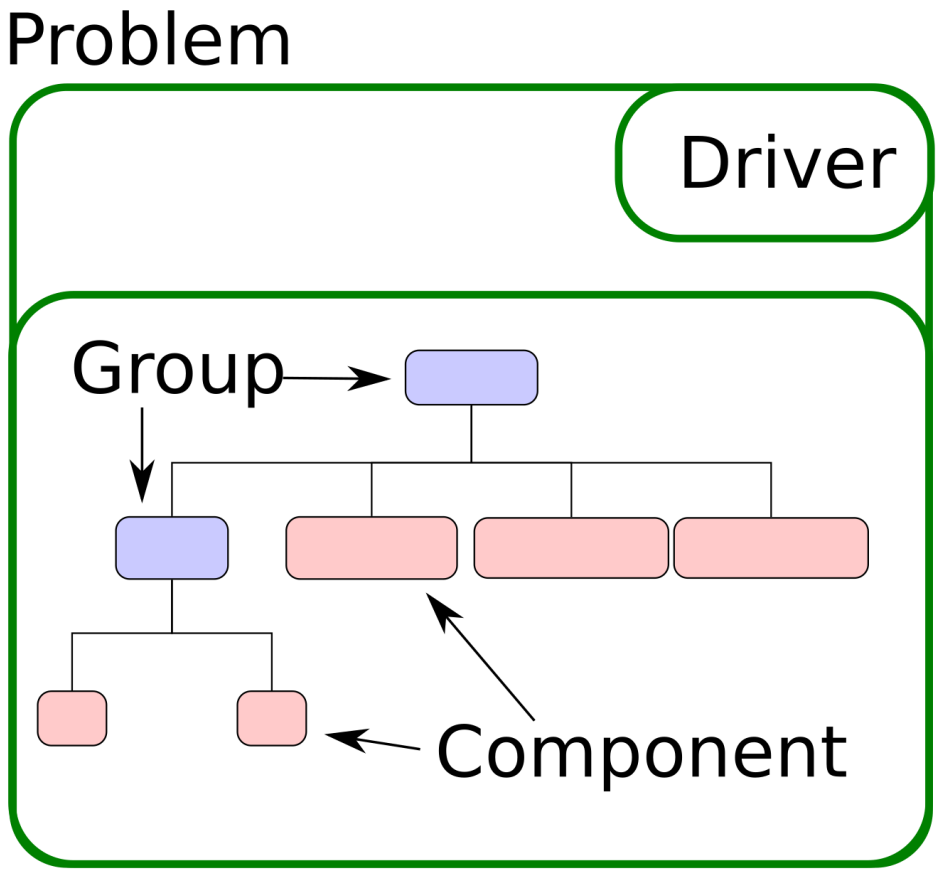
IndepVarComp example

```
indeps = self.add_subsystem('indeps', om.IndepVarComp())
indeps.add_output(‘h’, 10000, units=“ft“)
indeps.add_output(‘U’, 200, units=“kn")
indeps.add_output(‘Sref', 200, units=“ft**2”)
```

- IndepVarComps only have only outputs; they do not take in any input variables
- IVCs are used in an optimization context to allow the optimizer to control design variables

Groups: organizing components

- Combine components into a Group to build organized models
- Groups can hold other Groups, which allows hierarchal solution strategies
- Groups can used to create portable chunks of analysis chains

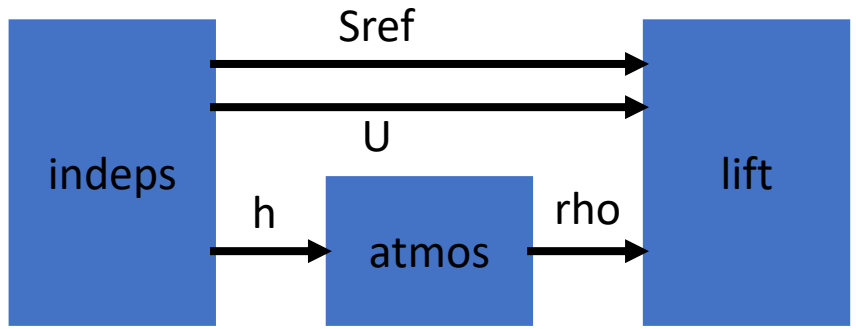


Grouping components

```
class ConnectExample(Group):

    def setup(self):
        # set some input values - optimizers act on independent variables
        indeps = self.add_subsystem('indeps', IndepVarComp())
        indeps.add_output('h', 10000, units="ft")
        indeps.add_output('U', 200, units="kn")
        indeps.add_output('Sref', 200, units="ft**2")

        # add your disciplinary models to the group
        self.add_subsystem('atmos', StdAtmComp())
        self.add_subsystem('lift', ComputeLift(num_pts = 1))

        # connect variables together
        self.connect('indeps.h', 'atmos.h')
        self.connect('atmos.rho', 'lift.rho')
        self.connect('indeps.Sref', 'lift.Sref')
        self.connect('indeps.U', 'lift.U')


if __name__ == "__main__":
    prob = Problem()
    prob.model = ConnectExample()
    prob.setup()
    prob['indeps.U'] = 150.
    prob.run_model()
    print(prob['lift.L'])
```

Define a model by subclassing Group

This is the model structure that is being connected



Grouping components

```
class ConnectExample(Group):

    def setup(self):
        # set some input values - optimizers act on independent variables
        indeps = self.add_subsystem('indeps', IndepVarComp())
        indeps.add_output('h', 10000, units="ft")
        indeps.add_output('U', 200, units="kn")
        indeps.add_output('Sref', 200, units="ft**2")

        # add your disciplinary models to the group
        self.add_subsystem('atmos', StdAtmComp())
        self.add_subsystem('lift', ComputeLift(num_pts = 1))

        # connect variables together
        self.connect('indeps.h', 'atmos.h')
        self.connect('atmos.rho', 'lift.rho')
        self.connect('indeps.Sref', 'lift.Sref')
        self.connect('indeps.U', 'lift.U')


if __name__ == "__main__":
    prob = Problem()
    prob.model = ConnectExample()
    prob.setup()
    prob['indeps.U'] = 150.
    prob.run_model()
    print(prob['lift.L'])
```

Include components using the add_subsystem(<name>, <component>) method

Python note:
These are *instances* of a component class

```
StdAtmComp())
ComputeLift(num_pts = 1))
```

Grouping components

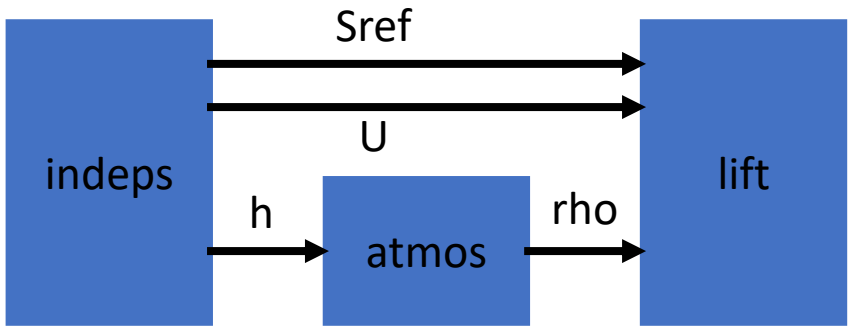
```
class ConnectExample(Group):

    def setup(self):
        # set some input values - optimizers act on independent variables
        indeps = self.add_subsystem('indeps', IndepVarComp())
        indeps.add_output('h', 10000, units="ft")
        indeps.add_output('U', 200, units="kn")
        indeps.add_output('Sref', 200, units="ft**2")

        # add your disciplinary models to the group
        self.add_subsystem('atmos', StdAtmComp())
        self.add_subsystem('lift', ComputeLift(num_pts = 1))

        # connect variables together
        self.connect('indeps.h', 'atmos.h')
        self.connect('atmos.rho', 'lift.rho')
        self.connect('indeps.Sref', 'lift.Sref')
        self.connect('indeps.U', 'lift.U')


if __name__ == "__main__":
    prob = Problem()
    prob.model = ConnectExample()
    prob.setup()
    prob['indeps.U'] = 150.
    prob.run_model()
    print(prob['lift.L'])
```

Your custom components can be defined in the same .py file, or imported from a package for modularity

Specify flags/options now

Grouping components

```
class ConnectExample(Group):

    def setup(self):
        # set some input values - optimizers act on independent variables
        indeps = self.add_subsystem('indeps', IndepVarComp())
        indeps.add_output('h', 10000, units="ft")
        indeps.add_output('U', 200, units="kn")
        indeps.add_output('Sref', 200, units="ft**2")

        # add your disciplinary models to the group
        self.add_subsystem('atmos', StdAtmComp())
        self.add_subsystem('lift', ComputeLift(num_pts = 1))

        # connect variables together
        self.connect('indeps.h', 'atmos.h')
        self.connect('atmos.rho', 'lift.rho')
        self.connect('indeps.Sref', 'lift.Sref')
        self.connect('indeps.U', 'lift.U')


if __name__ == "__main__":
    prob = Problem()
    prob.model = ConnectExample()
    prob.setup()
    prob['indeps.U'] = 150.
    prob.run_model()
    print(prob['lift.L'])
```



Connect parameters using the connect(<from>, <to>) method

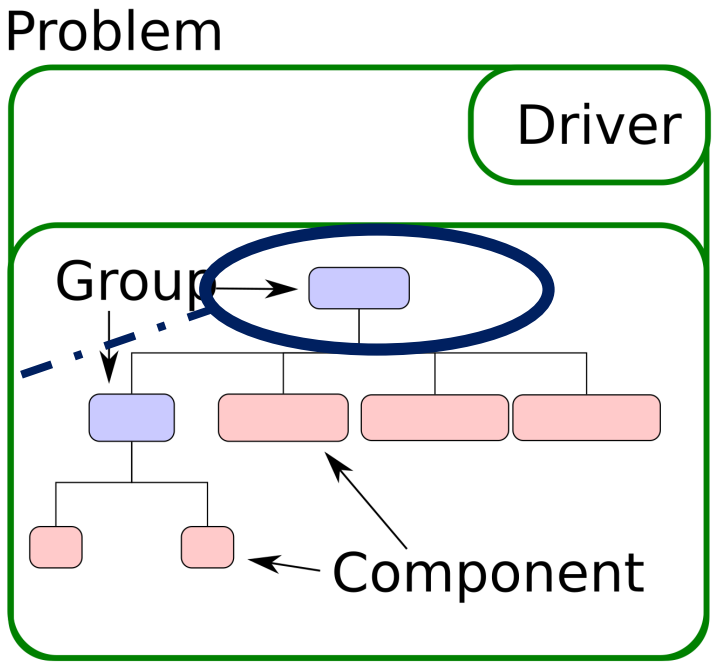
Grouping components

```
class ConnectExample(Group):

    def setup(self):
        # set some input values - optimizers act on independent variables
        indeps = self.add_subsystem('indeps', IndepVarComp())
        indeps.add_output('h', 10000, units="ft")
        indeps.add_output('U', 200, units="kn")
        indeps.add_output('Sref', 200, units="ft**2")

        # add your disciplinary models to the group
        self.add_subsystem('atmos', StdAtmComp())
        self.add_subsystem('lift', ComputeLift(num_pts = 1))

        # connect variables together
        self.connect('indeps.h', 'atmos.h')
        self.connect('atmos.rho', 'lift.rho')
        self.connect('indeps.Sref', 'lift.Sref')
        self.connect('indeps.U', 'lift.U')


if __name__ == "__main__":
    prob = Problem()
    prob.model = ConnectExample()
    prob.setup()
    prob['indeps.U'] = 150.
    prob.run_model()
    print(prob['lift.L'])
```

Note the namespace / address string:
component_name.variable

Grouping components

```
class ConnectExample(Group):

    def setup(self):
        # set some input values - optimizers act on independent variables
        indeps = self.add_subsystem('indeps', IndepVarComp())
        indeps.add_output('h', 10000, units="ft")
        indeps.add_output('U', 200, units="kn")
        indeps.add_output('Sref', 200, units="ft**2")

        # add your disciplinary models to the group
        self.add_subsystem('atmos', StdAtmComp())
        self.add_subsystem('lift', ComputeLift(num_pts = 1))

        # connect variables together
        self.connect('indeps.h', 'atmos.h')
        self.connect('atmos.rho', 'lift.rho')
        self.connect('indeps.Sref', 'lift.Sref')
        self.connect('indeps.U', 'lift.U')


if __name__ == "__main__":
    prob = Problem()
    prob.model = ConnectExample()
    prob.setup()
    prob['indeps.U'] = 150.
    prob.run_model()
    print(prob['lift.L'])
```



When you run an MDA/MDO problem, you will add one top-level Group instance to the problem

Grouping components

```
class ConnectExample(Group):

    def setup(self):
        # set some input values - optimizers act on independent variables
        indeps = self.add_subsystem('indeps', IndepVarComp())
        indeps.add_output('h', 10000, units="ft")
        indeps.add_output('U', 200, units="kn")
        indeps.add_output('Sref', 200, units="ft**2")

        # add your disciplinary models to the group
        self.add_subsystem('atmos', StdAtmComp())
        self.add_subsystem('lift', ComputeLift(num_pts = 1))

        # connect variables together
        self.connect('indeps.h', 'atmos.h')
        self.connect('atmos.rho', 'lift.rho')
        self.connect('indeps.Sref', 'lift.Sref')
        self.connect('indeps.U', 'lift.U')


if __name__ == "__main__":
    prob = Problem()
    prob.model = ConnectExample()
    prob.setup()
    prob['indeps.U'] = 150.
    prob.run_model()
    print(prob['lift.L'])
```

The subgroups and components are executed in the order that they're added to the system

Another way to connect…

- If parameters are widely used among many components, writing many connect() statements can be tedious
- Variable *promotion* is another way to make connections

```
class PromoteExample(Group):

    def setup(self):
        # set some input values - optimizers act on independent variables
        indeps = self.add_subsystem('indeps', IndepVarComp(), promotes_outputs=['h','U','Sref'])
        indeps.add_output('h', 10000, units="ft")
        indeps.add_output('U', 200, units="kn")
        indeps.add_output('Sref', 200, units="ft**2")

        # add your disciplinary models to the group
        self.add_subsystem('atmos', StdAtmComp(), promotes_inputs=['h'], promotes_outputs=['rho'])
        self.add_subsystem('lift', ComputeLift(num_pts = 1), promotes_inputs=['rho','Sref','U'])
```

What does variable promotion do?

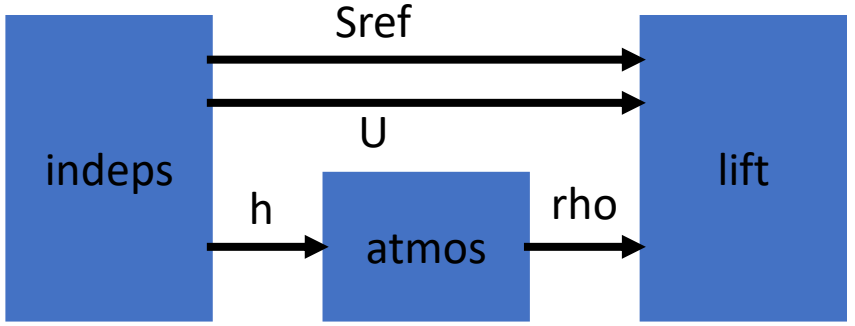
- Creates an alias for the variable one level up in the namespace
  - (*atmos.h → h*)
  - (*lift.rho → rho)*
- Automatically connects any matching I/O variable names
- Promote variables with wildcard (e.g. *_in or just *)

```
class PromoteExample(Group):

    def setup(self):
        # set some input values - optimizers act on independent variables
        indeps = self.add_subsystem('indeps', IndepVarComp(), promotes_outputs=['h','U','Sref'])
        indeps.add_output('h', 10000, units="ft")
        indeps.add_output('U', 200, units="kn")
        indeps.add_output('Sref', 200, units="ft**2")

        # add your disciplinary models to the group
        self.add_subsystem('atmos', StdAtmComp(), promotes_inputs=['h'], promotes_outputs=['rho'])
        self.add_subsystem('lift', ComputeLift(num_pts = 1), promotes_inputs=['rho','Sref','U'])
```

What does variable promotion do?

```
class PromoteExample(Group):

    def setup(self):
        # set some input values - optimizers act on independent variables
        indeps = self.add_subsystem('indeps', IndepVarComp(), promotes_outputs=['h','U','Sref'])
        indeps.add_output('h', 10000, units="ft")
        indeps.add_output('U', 200, units="kn")
        indeps.add_output('Sref', 200, units="ft**2")

        # add your disciplinary models to the group
        self.add_subsystem('atmos', StdAtmComp(), promotes_inputs=['h'], promotes_outputs=['rho'])
        self.add_subsystem('lift', ComputeLift(num_pts = 1), promotes_inputs=['rho','Sref','U'])


if __name__ == "__main__":
    prob = Problem()
    prob.model = PromoteExample()
    prob.setup()
    # both of these are correct
    prob['indeps.U'] = 150.
    prob['U'] = 150.
    prob.run_model()
    print(prob['lift.L'])
```

Input/output connection established automatically



What does variable promotion do?

Multiple promoted outputs with same name: *not* allowed (will raise an exception)

```
class PromoteExample(Group):

    def setup(self):
        # set some input values - optimizers act on independent variables
        indeps = self.add_subsystem('indeps', IndepVarComp(), promotes_outputs=['h','U','Sref'])
        indeps.add_output('h', 10000, units="ft")
        indeps.add_output('U', 200, units="kn")
        indeps.add_output('Sref', 200, units="ft**2")
        indeps2 = self.add_subsystem('indeps2', IndepVarComp(), promotes_outputs=['U'])
        indeps2.add_output('U', 200, units="kn")

        # add your disciplinary models to the group
        self.add_subsystem('atmos', StdAtmComp(), promotes_inputs=['h'], promotes_outputs=['rho'])
        self.add_subsystem('lift', ComputeLift(num_pts = 1), promotes_inputs=['rho','Sref','U'])
        self.add_subsystem('lift2', ComputeLift(num_pts = 1), promotes_inputs=['rho','Sref','U'])
```

What does variable promotion do?

```
class PromoteExample(Group):

    def setup(self):
        # set some input values - optimizers act on independent variables
        indeps = self.add_subsystem('indeps', IndepVarComp(), promotes_outputs=['h','U','Sref'])
        indeps.add_output('h', 10000, units="ft")
        indeps.add_output('U', 200, units="kn")
        indeps.add_output('Sref', 200, units="ft**2")
        indeps2 = self.add_subsystem('indeps2', IndepVarComp(), promotes_outputs=['U'])
        indeps2.add_output('U', 200, units="kn")

        # add your disciplinary models to the group
        self.add_subsystem('atmos', StdAtmComp(), promotes_inputs=['h'], promotes_outputs=['rho'])
        self.add_subsystem('lift', ComputeLift(num_pts = 1), promotes_inputs=['rho','Sref','U'])
        self.add_subsystem('lift2', ComputeLift(num_pts = 1), promotes_inputs=['rho','Sref','U'])
```

Multiple promoted inputs with identical name: *OK* and *encouraged* (all connections automatically made)

What does variable promotion do?

```
class PromoteExample(Group):

    def setup(self):
        # set some input values - optimizers act on independent variables
        indeps = self.add_subsystem('indeps', IndepVarComp(), promotes_outputs=['h','U'])
        indeps.add_output('h', 10000, units="ft")
        indeps.add_output('U', 200, units="kn")
        indeps.add_output('Sref', 200, units="ft**2")

        # add your disciplinary models to the group
        self.add_subsystem('atmos', StdAtmComp(), promotes_inputs=['h'], promotes_outputs=['rho'])
        self.add_subsystem('lift', ComputeLift(num_pts = 1), promotes_inputs=['rho','U'])
        self.connect('indeps.Sref', 'lift.Sref')
```

Mixing promotion with connect statements: allowed / appropriate

Lab 0: Implementing simple explicit calculations (Breguet Range)

Step 1: Install OpenMDAO

Step 2: Write your first component

Step 3: World domination!

Lab 0.a: Install

- Open `cmd` prompt
- Internet install: `pip install openmdao`
- Local Install:
  - cd to wherever OpenMDAO is downloaded:
    `` `pip install .` `` (note the period)
  - This installs from local source files, not PyPI
- `cd ../openmdao_training`
- `python paraboloid.py`
  - If this works without error, your installation should be good

Lab 0.b: Aircraft Range

- Open *lab_0_template.py* in a text editor or IDE and rename it to lab_0.py
- The `BreguetRange` component implements the electric Breguet equation:

$$R_b = \frac{L}{D}\eta_e\eta_{int}\eta_p\frac{e_b}{g}\frac{m_b}{m_{TO}} \qquad m_{TO} = m_{empty} + m_{payload} + m_{battery}$$

- Your goal is to compute the maximum range of an airplane given a certain payload

Lab 0.b: Aircraft Range

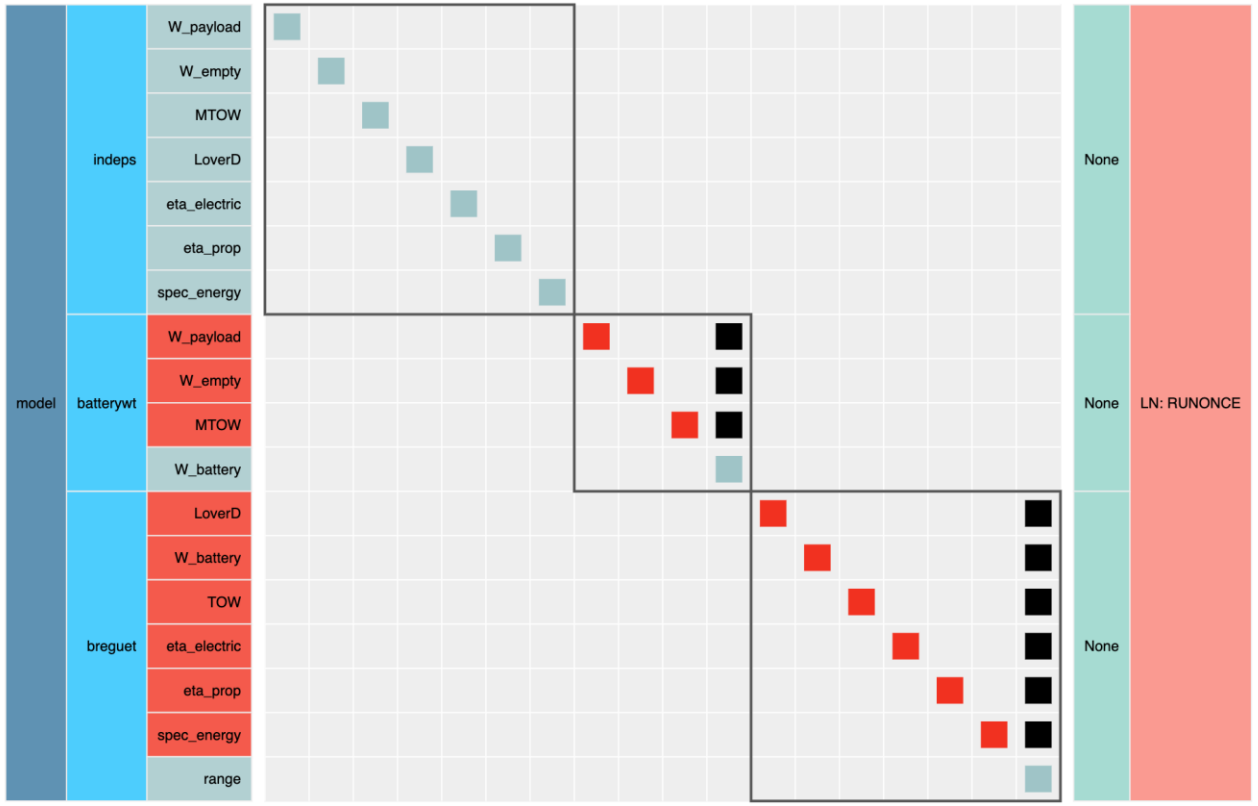
- Complete the TODOs in the `BatteryWeight` component
- Complete the TODOs in the `ElecRangeGroup` definition by connecting the two components

- `cd` into project folder to check and run the model:
  - `openmdao view_model lab_0.py` (creates model diagram)
  - `python lab_0.py` (runs the model)

- Answer key in the *lab_0_solution.py* file (don't cheat!)

Lab 0.b: Aircraft Range

- `openmdao view_model lab_0.py`
- If you forgot any connections, you'll see some red



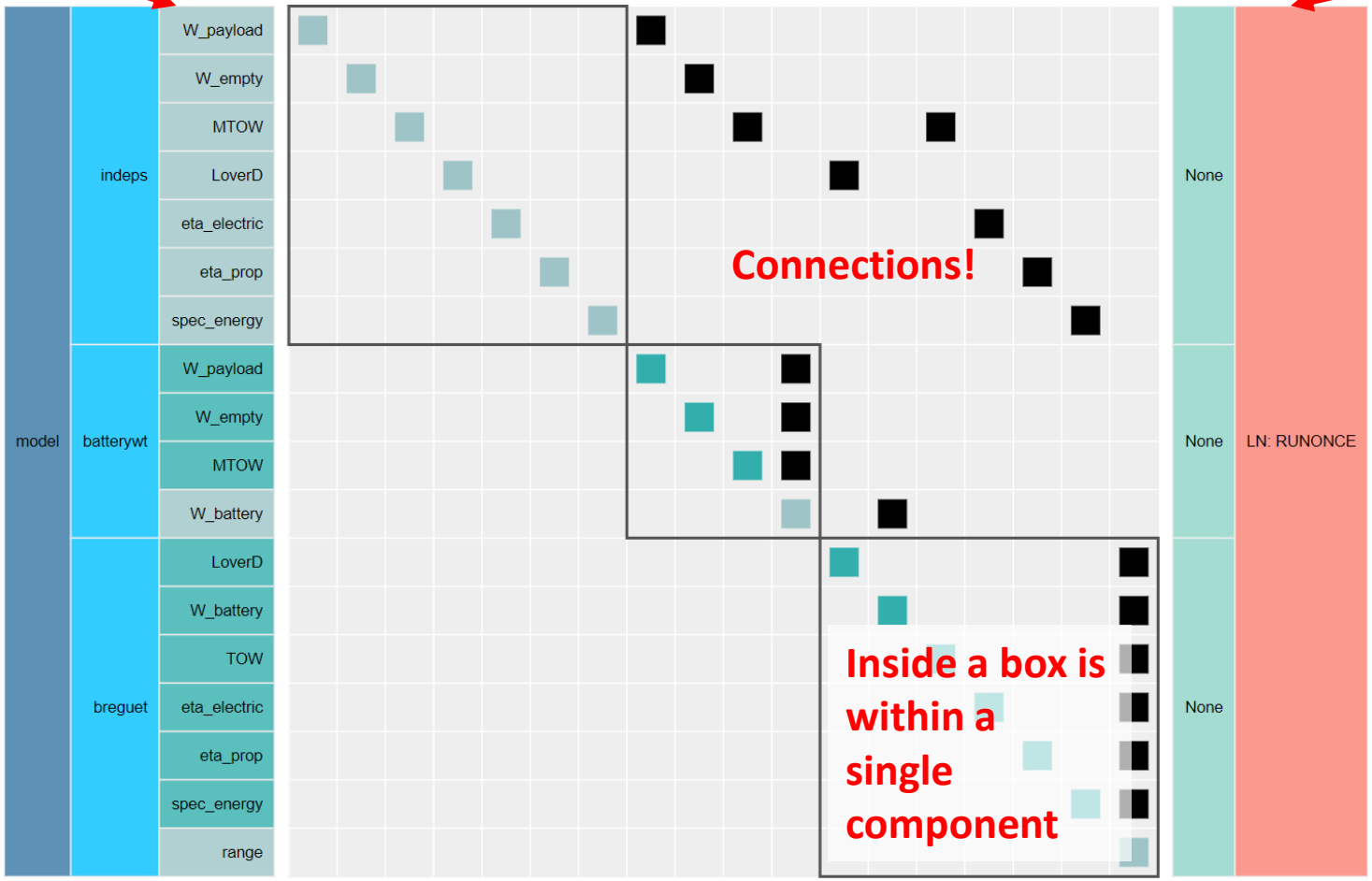
All the red boxes represent unconnected inputs to components.

Use explicit connection or variable promotion to get rid of all the red

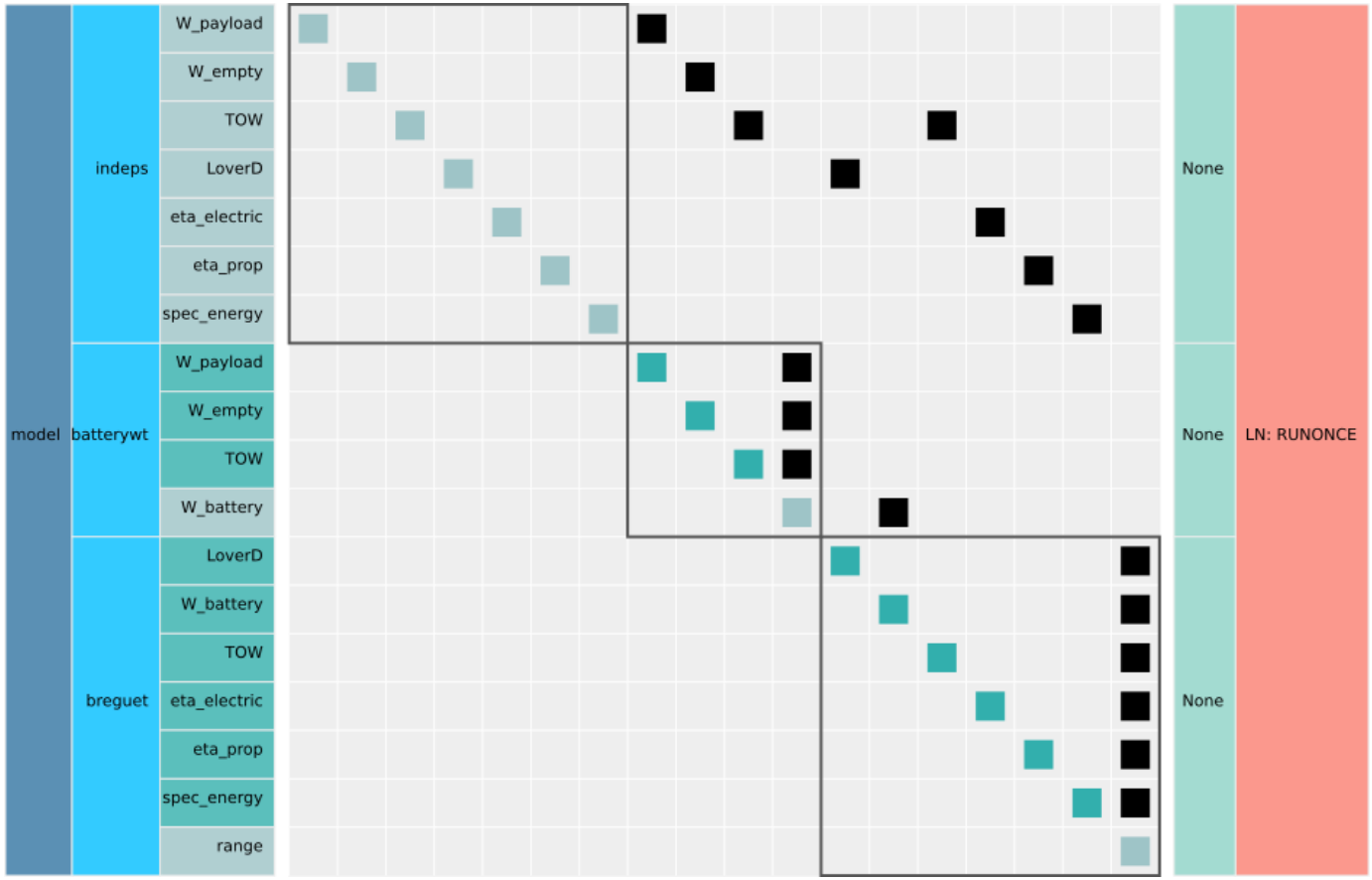
Lab 0.b: The n2 is your best friend!

Model hierarchy (groups, components, variables)

Linear and nonlinear solver hierarchy (more on this later…)



Lab 0.b: Aircraft Range



It should look like this!

This diagram is interactive:

Right click on the components/groups to collapse and expand them

Click on any black colored square to trace connections between components

N2 diagram is upper-triangular:
Pure explicit computation (feed-forward)

Lab 0 summary:

Everything we just did we can do faster in Excel.

This is a tutorial so the models are extremely simple and cheap, but …

- Real-world models have hierarchies of dozens or hundreds of logical components
- Real-world models often lack a closed form solution and require some kind of *solver* or iteration strategy

Using solvers with implicit models

- Gradient-free solver: Nonlinear Block Gauss-Seidel (i.e. Fixed point iteration)
- Gradient-based solver: Newton's Method
- Lab 1: Simple aircraft sizing and experimenting with different solver algorithms

OpenMDAO Nonlinear Solvers

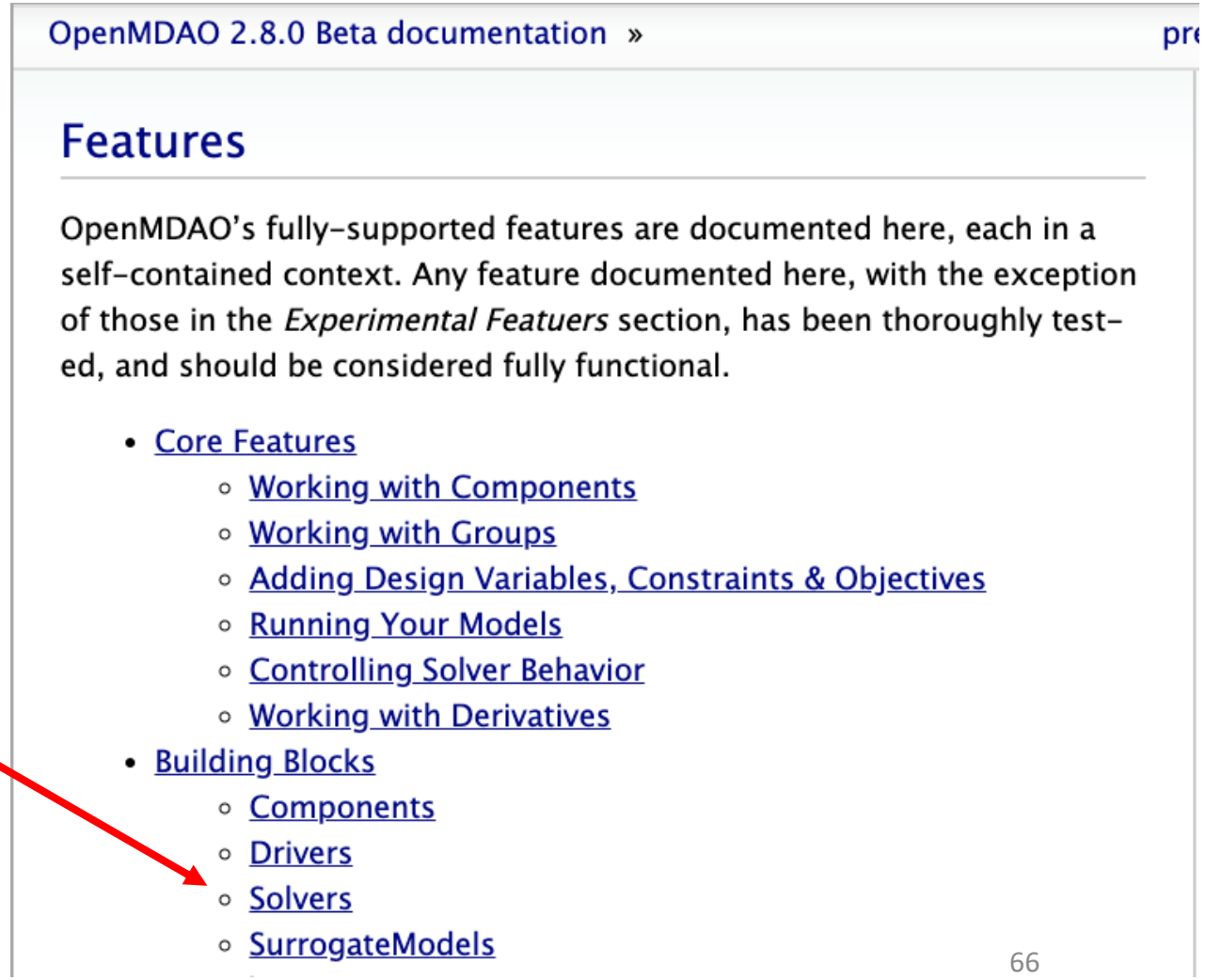
When a circular dependency is detected, OpenMDAO needs a solver to converge the problem:

- Choice #1: NonlinearBlockGS()
- Choice #2: NewtonSolver()
- Choice #3: BroydenSolver()
- Choice #4: NonlinearBlockJac()

Check out the docs for more info!

Look at the docs for OpenMDAO's standard library:

Lots of details on all the different solvers!

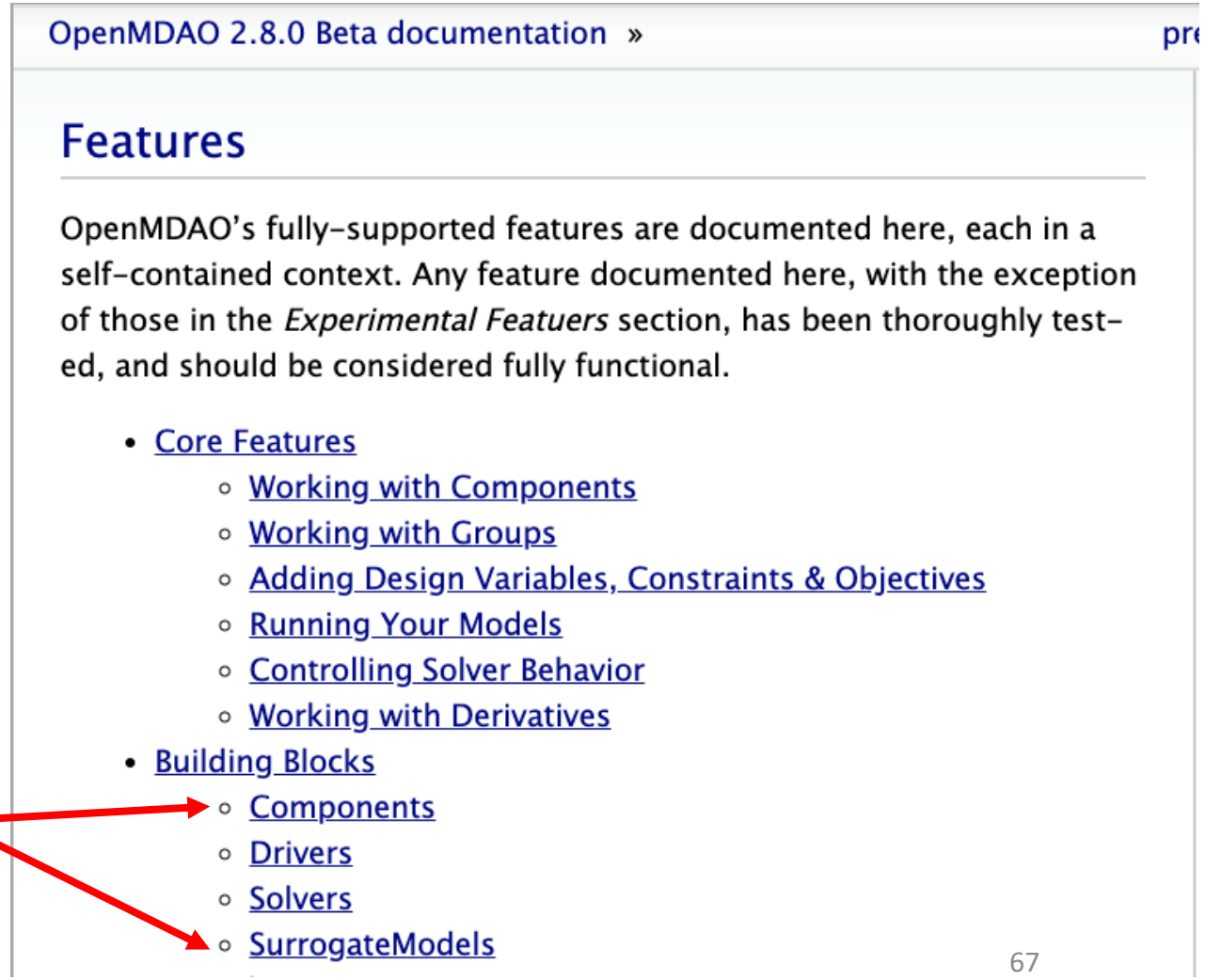


Check out the docs for more info!

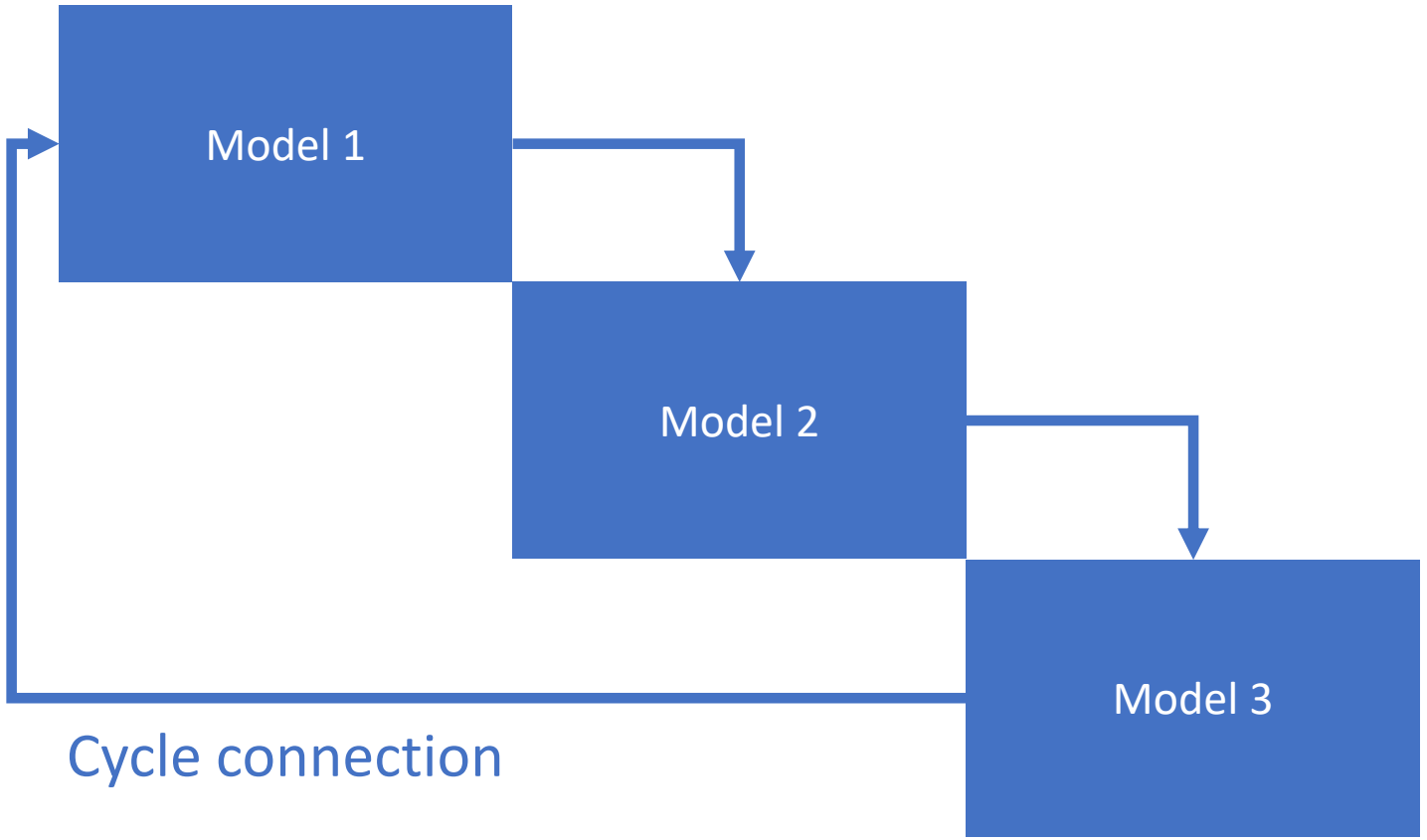
Look at the docs for OpenMDAO's standard library:

Lots of details on all the different solvers!

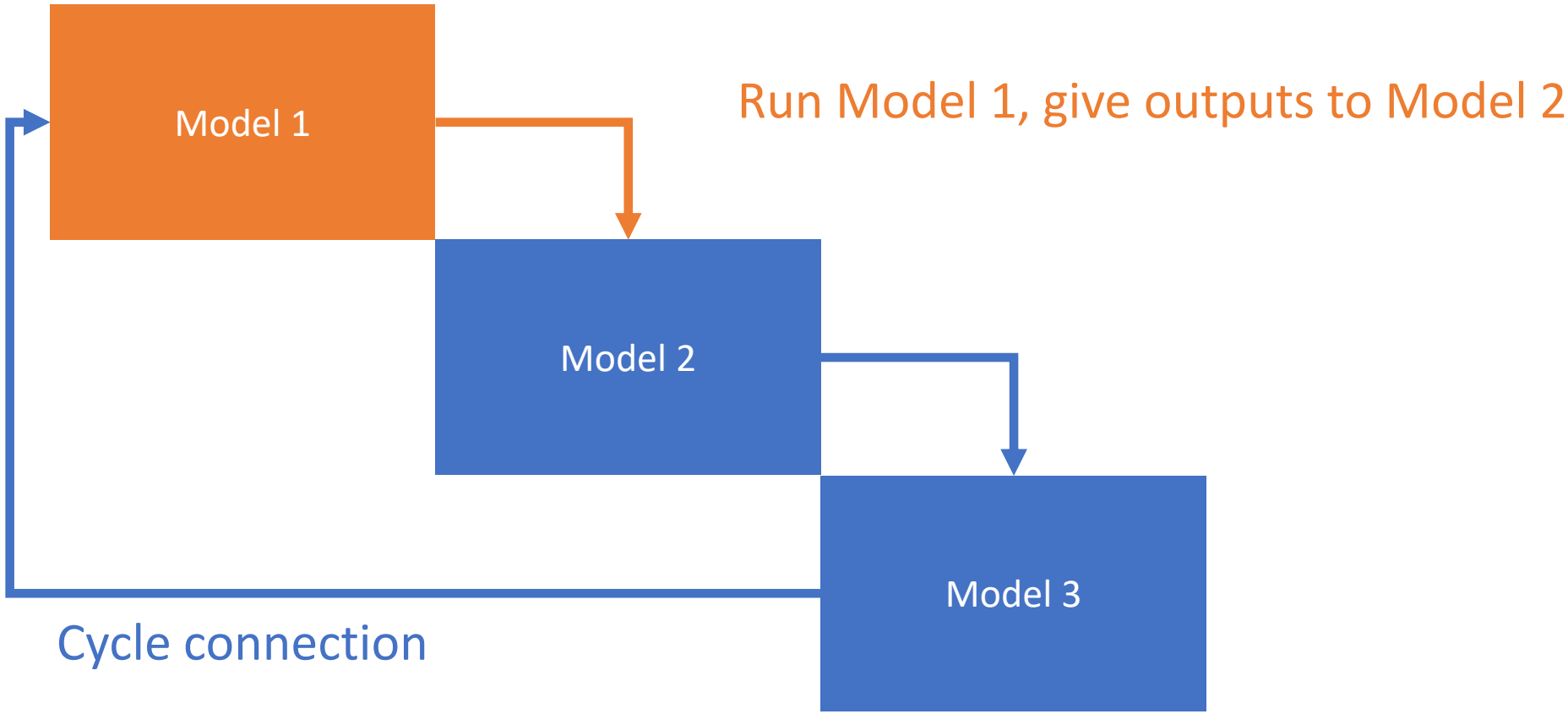
Also see sections for surrogates and helpful general purpose components!



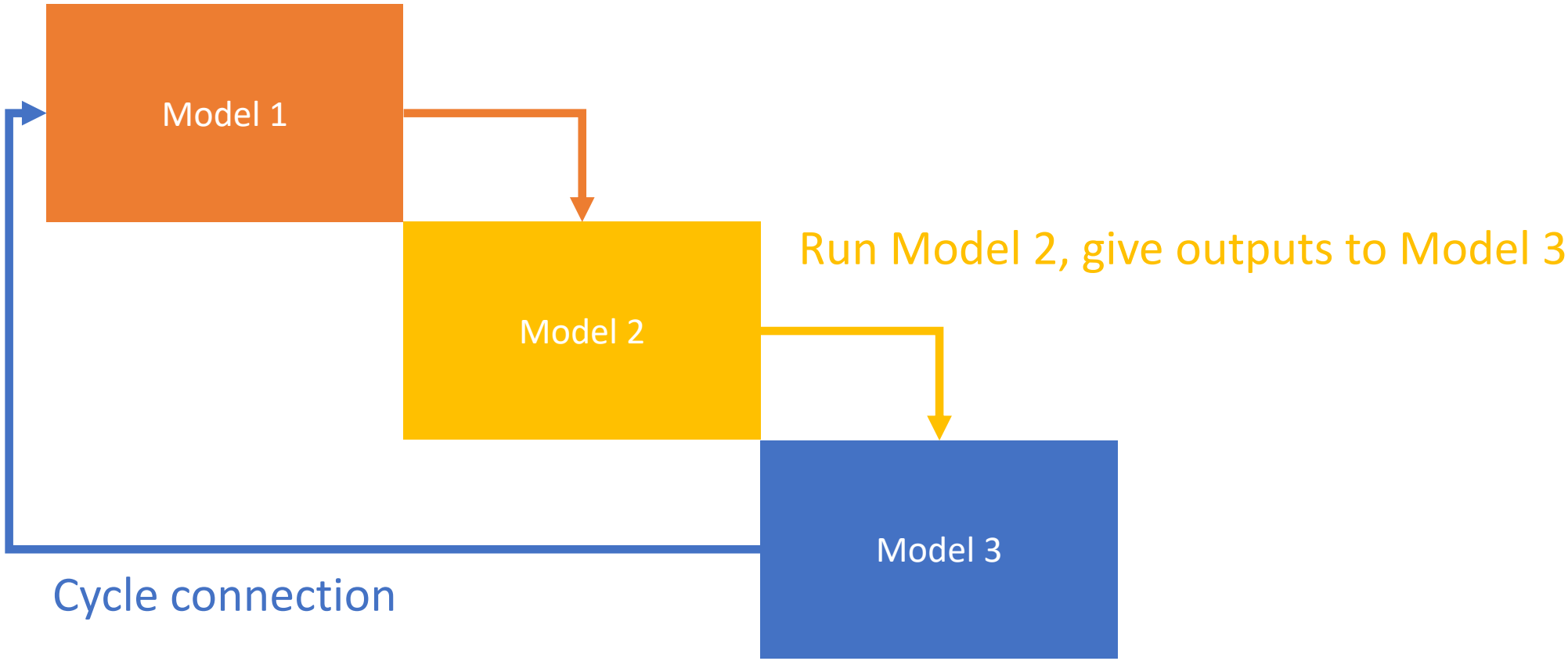
Nonlinear Block Gauss-Seidel (Deriv Free)



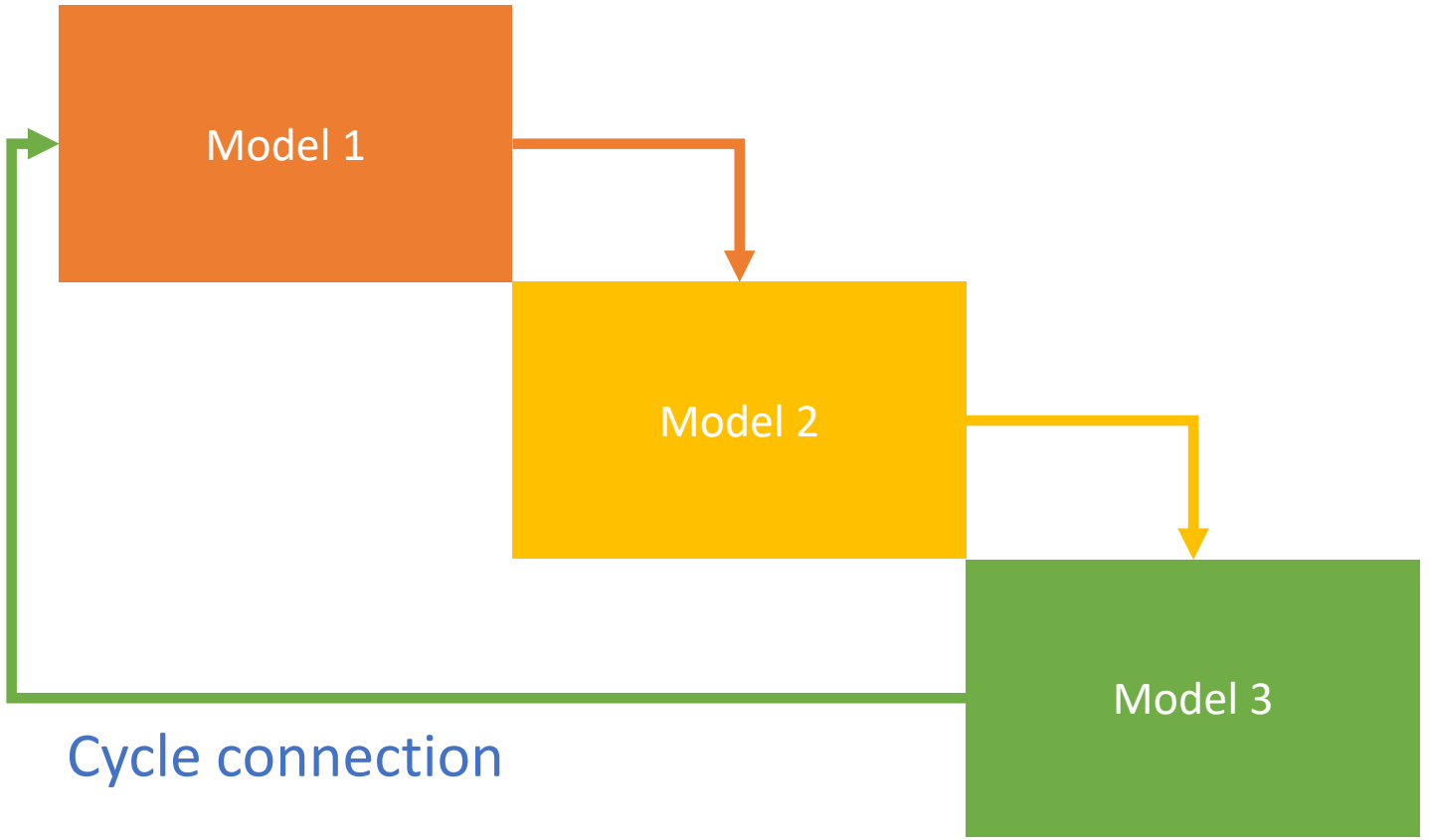
Nonlinear Block Gauss-Seidel



Nonlinear Block Gauss-Seidel

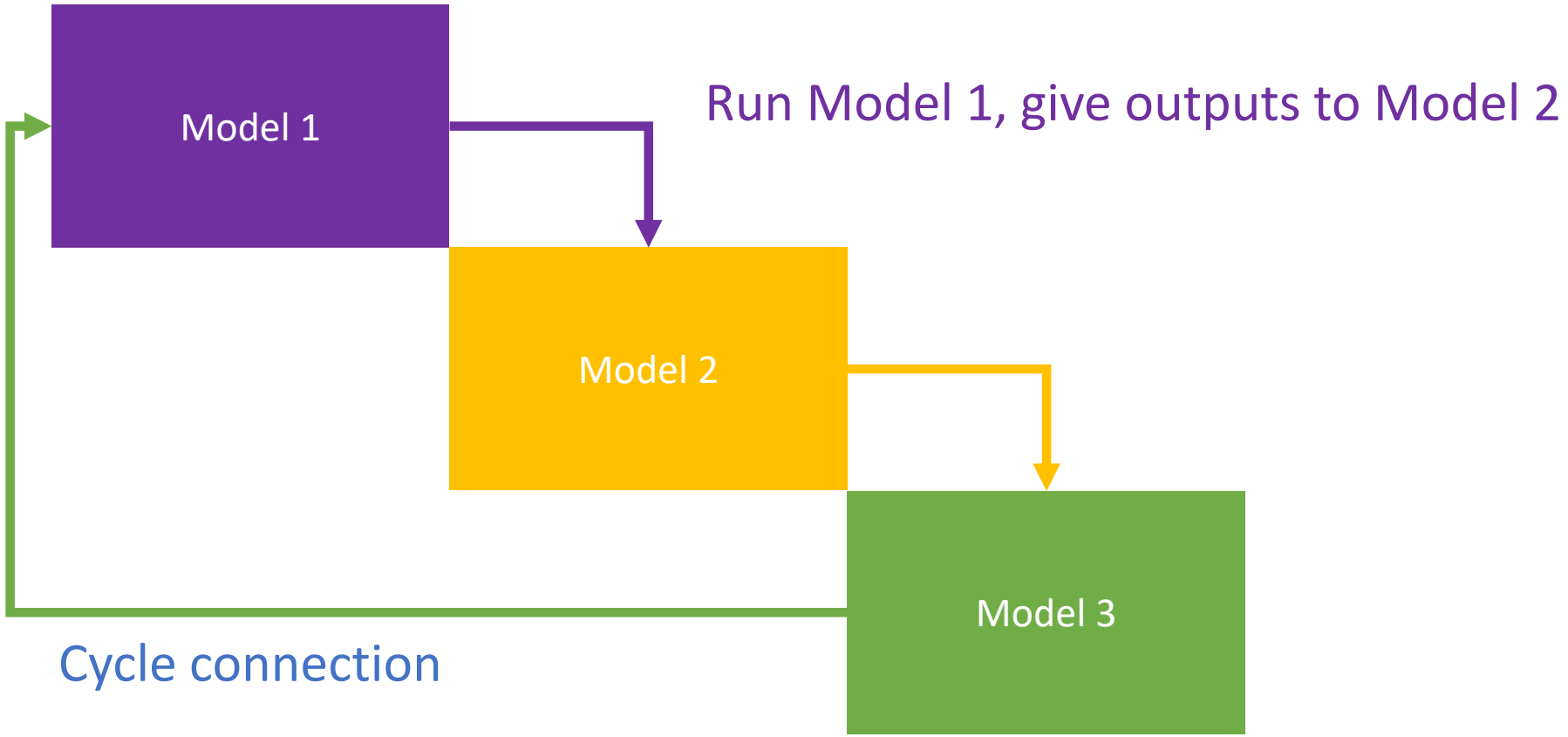


Nonlinear Block Gauss-Seidel

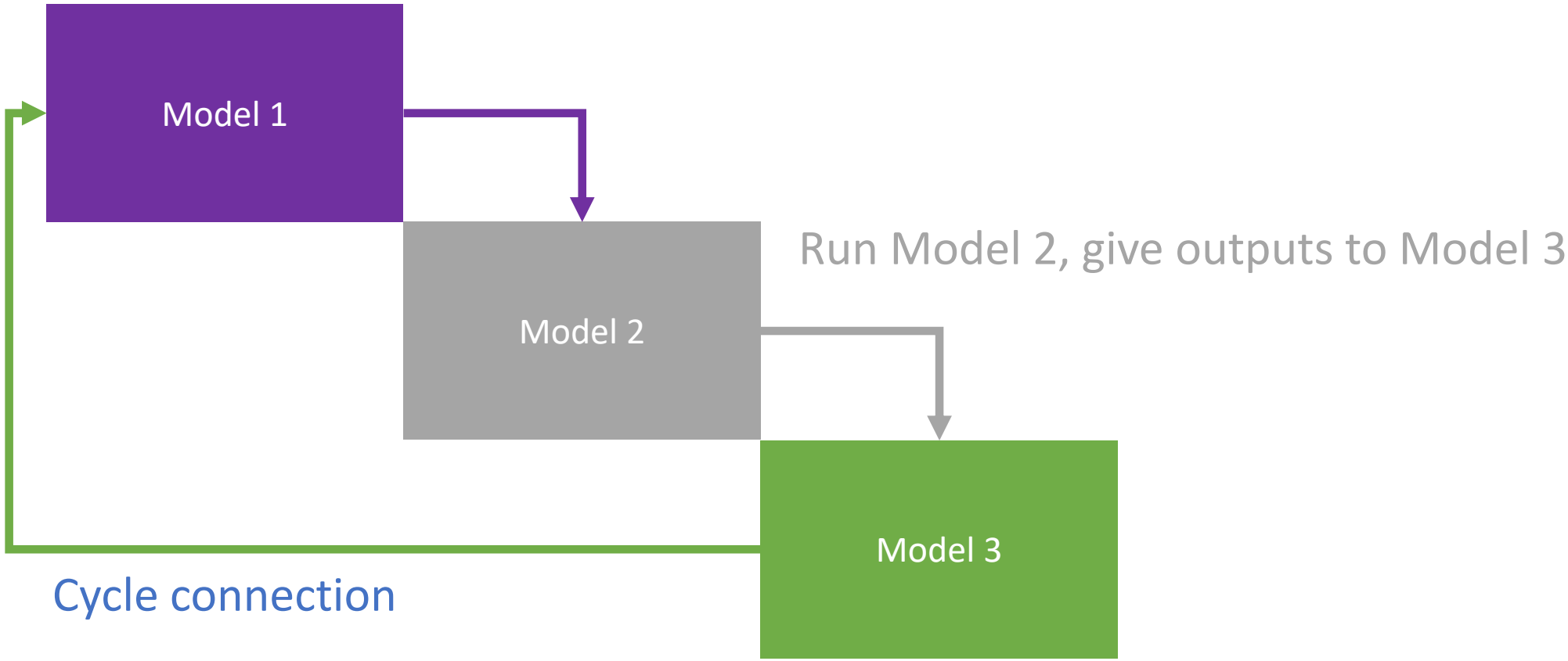


Run Model 3, give outputs to Model 1

Nonlinear Block Gauss-Seidel

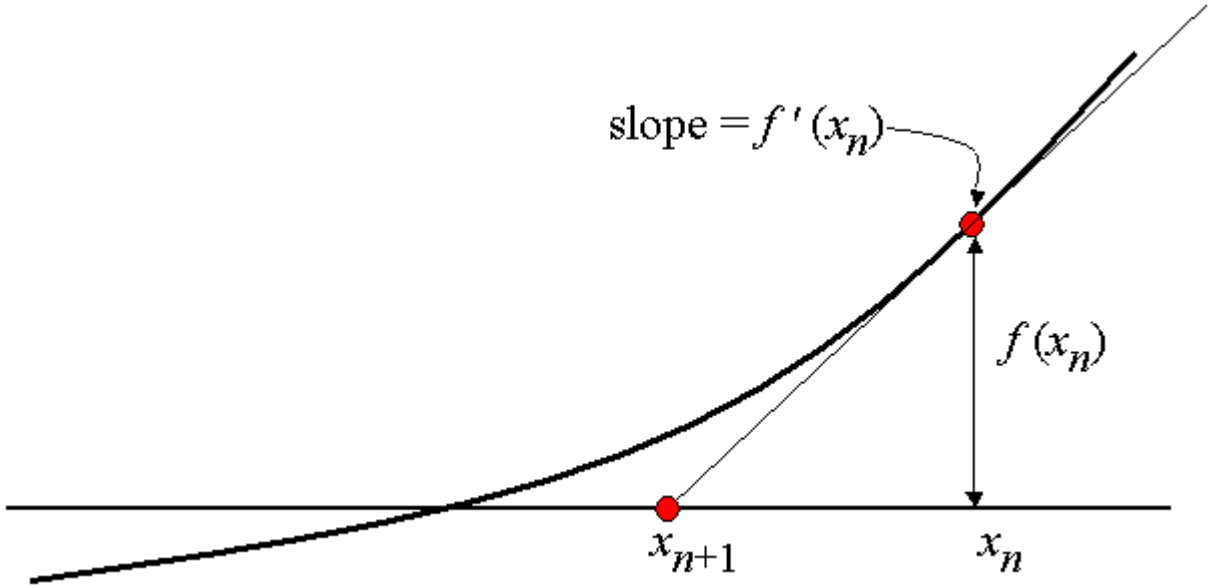


Nonlinear Block Gauss-Seidel



Newton's Method is simple in one dimension (Requires derivs)

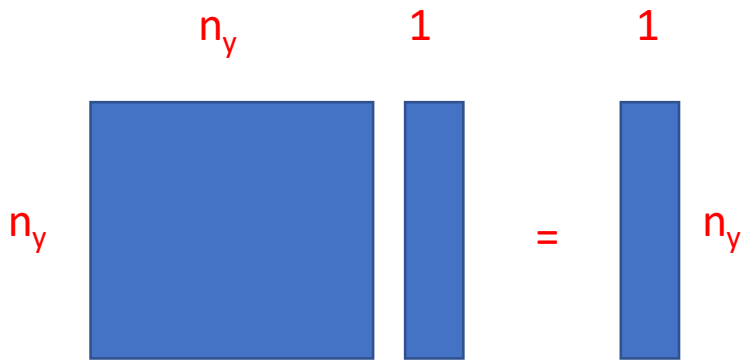
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$



Newton Solver for multiple dimensions is a bit more complex

$$R(x, y) = 0$$

We want to drive residuals to 0
x: design variables
y: implicit state variables



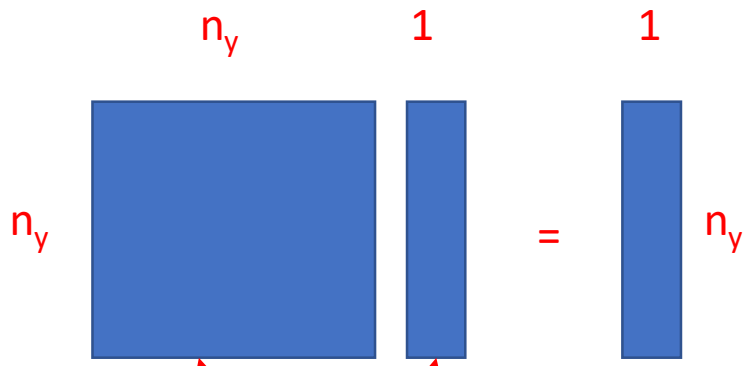
$$\frac{\partial R}{\partial y}\delta y = -R(x, y)$$

Solve this linear system

$$y_{n+1} = y_n + \delta y$$

Then take a step. Repeat until converged

Newton Solver needs some partial derivatives!



$$\frac{\partial R}{\partial y}\delta y = -R(x, y)$$

$$y_{n+1} = y_n + \delta y$$

Partial Derivatives in OpenMDAO

```
class WeightBuild(ExplicitComponent):
    """Compute TOW from component weights"""

    def setup(self):
        # define the following inputs: W_payload, W_empty, TOW
        self.add_input('W_payload', 800, units='lbm')
        self.add_input('W_empty', 5800, units='lbm')
        self.add_input('W_battery', 1500, units='lbm')

        # define the following outputs: W_battery
        self.add_output('TOW', val=6000, units='lbm')

        # declare generic finite difference partials
        self.declare_partials('TOW',['*'])

    def compute(self, inputs, outputs):
        # implement the calculation W_battery = TOW - W_payload - W_empty
        outputs['TOW'] = inputs['W_battery'] + inputs['W_payload'] + inputs['W_empty']

    def compute_partials(self, inputs, partials):
        partials['TOW','W_battery'] = 1
        partials['TOW','W_payload'] = 1
        partials['TOW','W_empty'] = 1
```

Define analytic derivatives: partials(<of>, <with respect to>)

This tells OpenMDAO which partials exist

This tells OpenMDAO the actual values of the partials

OpenMDAO assembles the Jacobian for you, from provided component partials

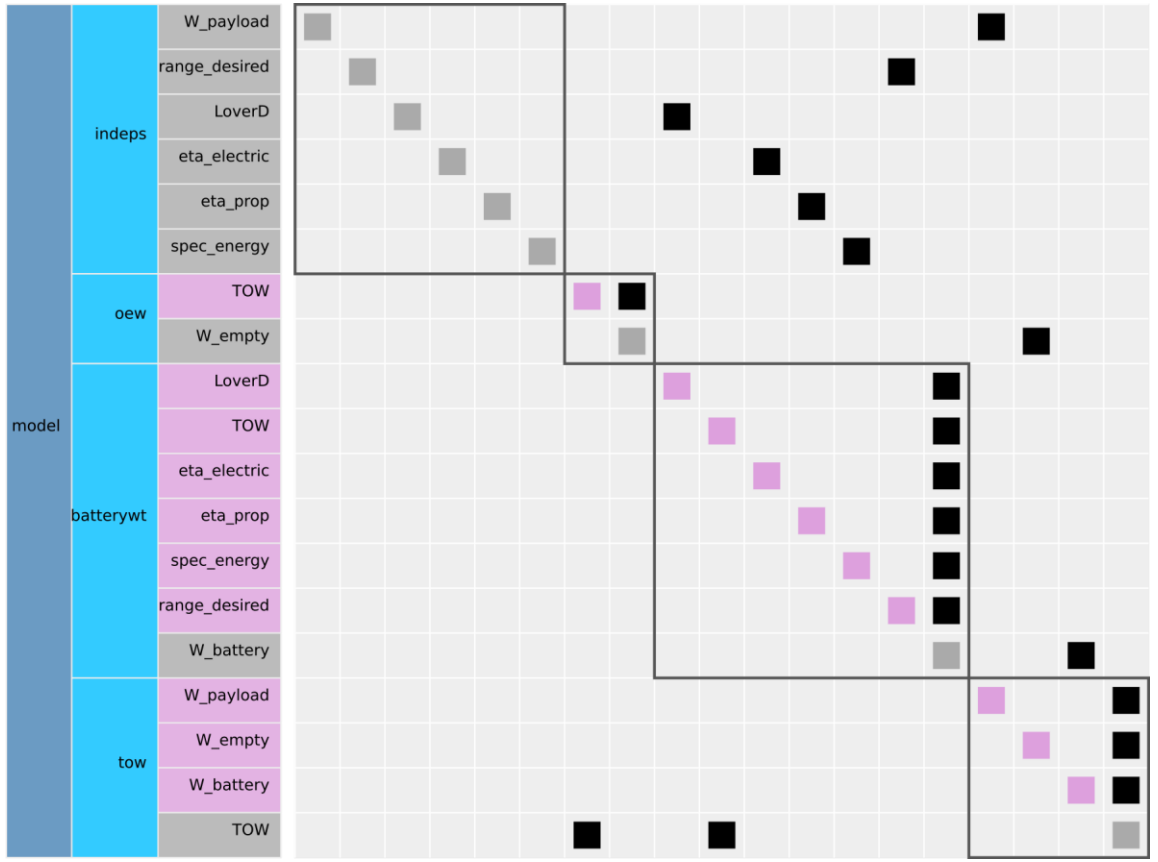
$$\frac{\partial R}{\partial y}\delta y = -R(x,y)$$

$$y_{n+1} = y_n + \delta y$$

Lab # 1 : Aircraft Sizing

Next, we will “size” an electric aircraft for desired range

- Open *lab_1_template.py* in a text editor or IDE and rename lab_1.py
- Check the model by building an N2 diagram
- This system is implicit (because it has a cycle) but has no implicit components



Lab # 1 : Aircraft Sizing

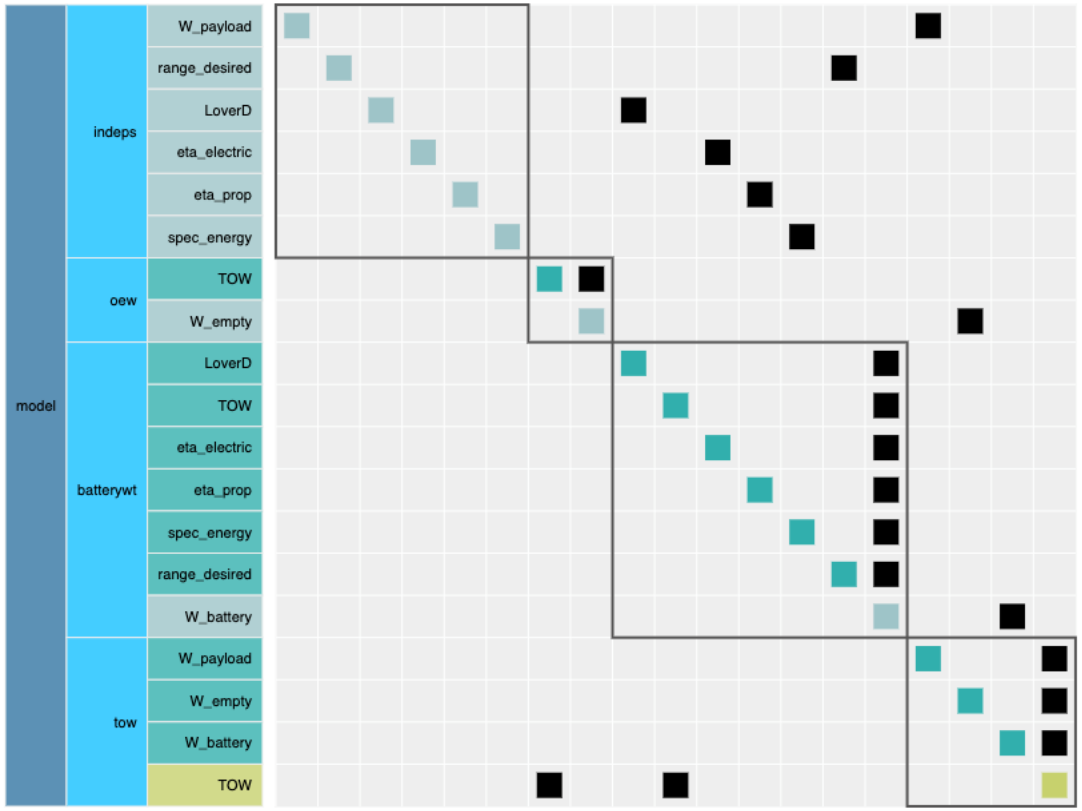
- Run the model as-is using NLBGS (*python lab_1.py*)
  - The numbers that print out are the absolute and relative residuals
- Change the empty weight fraction to 0.55
  - check out the `ExecComp`
- Try using the Newton solver (~ Line 124). What happens?
  - Check your partial derivatives (~ Line 173)
  - Fix the partial derivatives of the incorrect components
  - Which solver uses fewer iterations?
- Print all the inputs and outputs (~ Line 178, 179)

check_partials output

```
-------------------------
Component: ExecComp 'oew'
-------------------------
'<output>' wrt '<variable>' | fwd mag.   | check mag. | a(fwd-chk) | r(fwd-chk)
-------------------------------------------------------------------------------
'W_empty'  wrt 'TOW'        | 6.0000e-01 | 6.0000e-01 | 2.3842e-08 | 3.9736e-08
-----------------------------------
Component: BatteryWeight 'batterywt'
-----------------------------------
'<output>'  wrt '<variable>'    | fwd mag.   | check mag. | a(fwd-chk) | r(fwd-chk)
-------------------------------------------------------------------------------------
'W_battery' wrt 'LoverD'        | 0.0000e+00 | 9.6353e+01 | 9.6353e+01 | 1.0000e+00 >ABS_TOL >REL_TOL
'W_battery' wrt 'TOW'           | 0.0000e+00 | 3.2118e-01 | 3.2118e-01 | 1.0000e+00 >ABS_TOL >REL_TOL
'W_battery' wrt 'eta_electric'  | 0.0000e+00 | 2.0946e+03 | 2.0946e+03 | 1.0000e+00 >ABS_TOL >REL_TOL
'W_battery' wrt 'eta_prop'      | 0.0000e+00 | 2.3218e+03 | 2.3218e+03 | 1.0000e+00 >ABS_TOL >REL_TOL
'W_battery' wrt 'range_desired' | 0.0000e+00 | 1.2847e+01 | 1.2847e+01 | 1.0000e+00 >ABS_TOL >REL_TOL
'W_battery' wrt 'spec_energy'   | 0.0000e+00 | 6.4235e+00 | 6.4235e+00 | 1.0000e+00 >ABS_TOL >REL_TOL
---------------------------
Component: WeightBuild 'tow'
---------------------------
'<output>' wrt '<variable>' | fwd mag.   | check mag. | a(fwd-chk) | r(fwd-chk)
-------------------------------------------------------------------------------
'TOW'      wrt 'W_battery'  | 7.5000e+01 | 1.0000e+00 | 7.6000e+01 | 7.6000e+01 >ABS_TOL >REL_TOL
'TOW'      wrt 'W_empty'    | 7.5000e+01 | 1.0000e+00 | 7.6000e+01 | 7.6000e+01 >ABS_TOL >REL_TOL
'TOW'      wrt 'W_payload'  | 7.5000e+01 | 1.0000e+00 | 7.6000e+01 | 7.6000e+01 >ABS_TOL >REL_TOL
############################################################
Sub Jacobian with Largest Relative Error: WeightBuild 'tow'
############################################################
'<output>' wrt '<variable>' | fwd mag.   | check mag. | a(fwd-chk) | r(fwd-chk)
-------------------------------------------------------------------------------
'TOW'      wrt 'W_battery'  | 7.5000e+01 | 1.0000e+00 | 7.6000e+01 | 7.6000e+01
```

Lab # 1 : Aircraft Sizing continued

- Switch back to the Gauss-Seidel solver
- Switch to the `WeightBuildImplicit` component
  - solves for TOW by forcing design range equal to analyzed range
  - What happens when you run the model? Why?
  - Try the Newton solver



Optimization with and without analytic derivatives

- High level introduction to different methods of computing derivatives

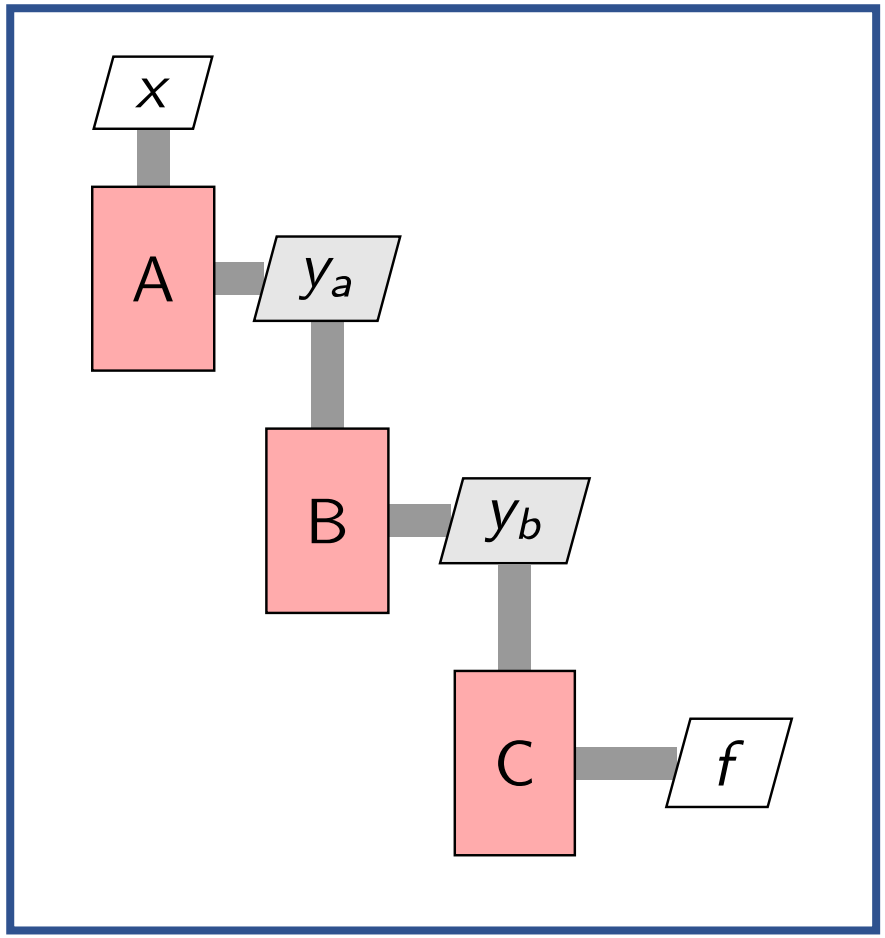
- Lab 2: Optimization of a FEM for a cantilever beam

Optimization

OpenMDAO is capable of evaluating DOEs just like ModelCenter, but…

Gradient based optimization is A LOT faster!

So we need **total derivatives** across the whole model:



$$f(\mathbf{x})$$

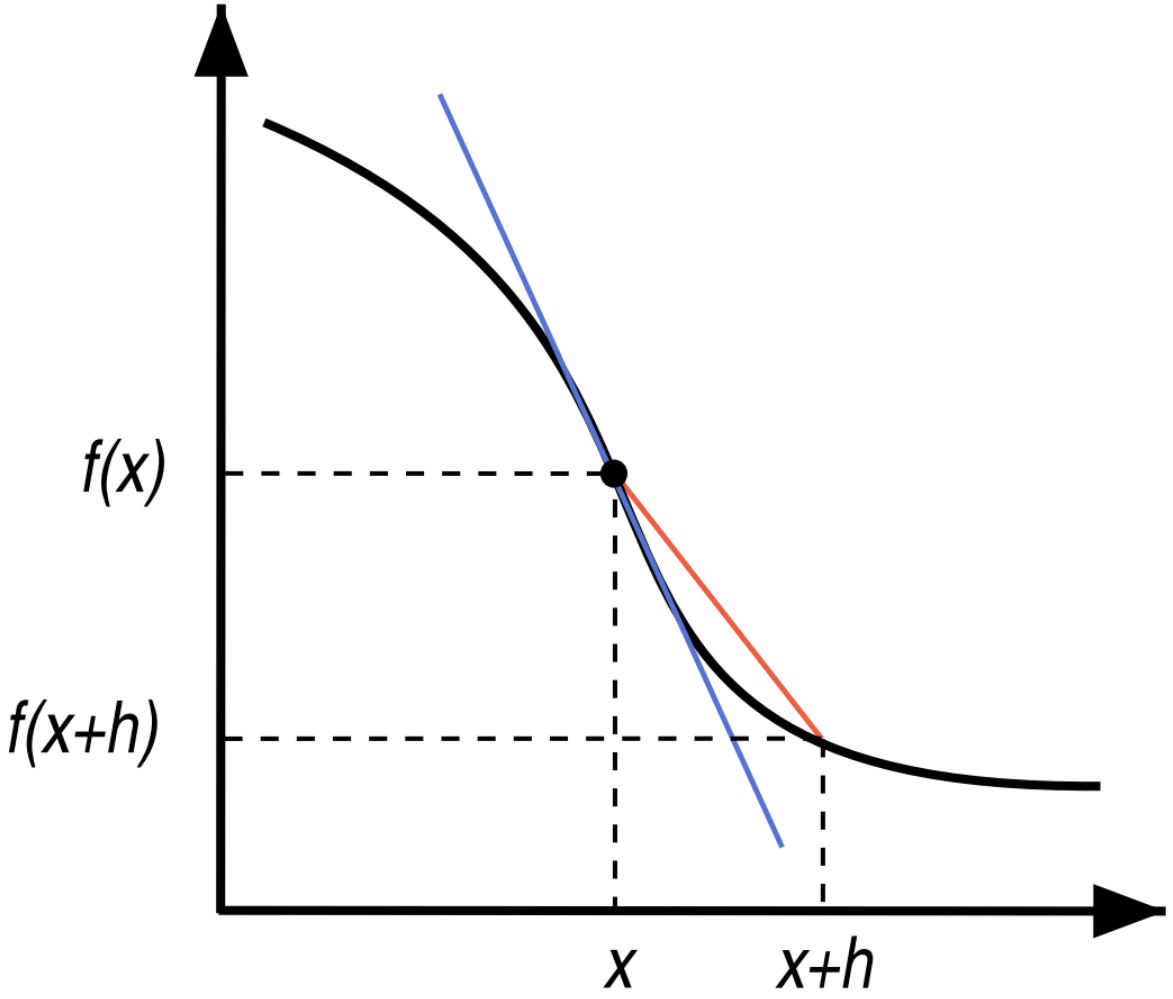
$$\frac{df(\mathbf{x})}{d\mathbf{x}}$$

??

Finite difference is easy

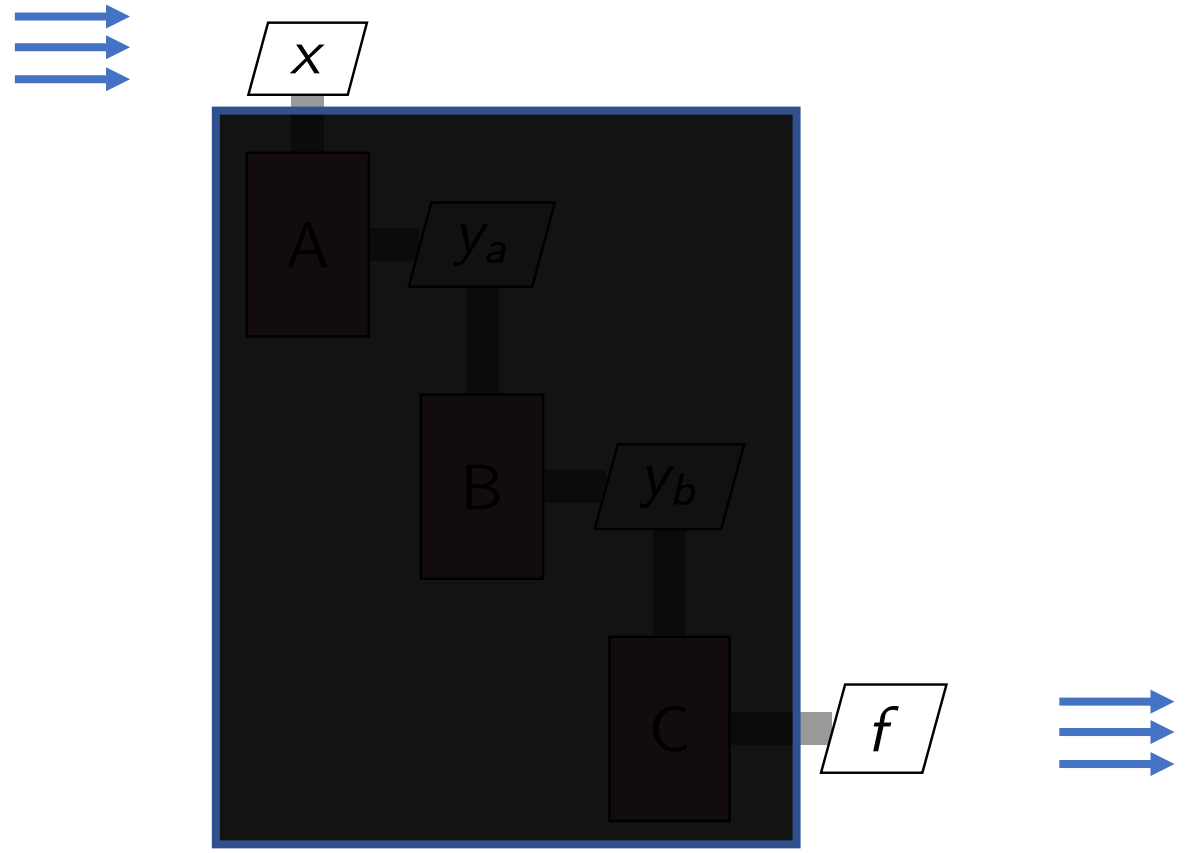
$$\frac{\partial \boldsymbol{F}}{\partial x_j} = \frac{\boldsymbol{F}(\boldsymbol{x} + \boldsymbol{e}_j h) - \boldsymbol{F}(\boldsymbol{x})}{h} + \mathcal{O}(h)$$

$f(x+h)$	$+1.2345678901234\color{red}{31}$
$f(x)$	$+1.234567890123456$
Δ f	-0.000000000000025



Finite difference is easy, but…

- Treats your model as a black-box
- Expensive
- Accuracy can be bad, especially close to the optimal point



$$\frac{\partial \boldsymbol{F}}{\partial x_j} = \frac{\boldsymbol{F}(\boldsymbol{x} + \boldsymbol{e}_j h) - \boldsymbol{F}(\boldsymbol{x})}{h} + \mathcal{O}(h)$$

It's not a good idea to finite difference across solvers

Seriously… don't do this unless you really have to.

- It's expensive (make your solver tolerances really tight)
- It's inaccurate
- It will probably make your optimization converge slowly!

Lots of papers on this topic:

E. S. Hendricks and J. S. Gray, "Pycycle: a tool for efficient optimization of gas turbine engine cycles," *Aerospace*, vol. 6, iss. 87, 2019.

E. S. Hendricks, "A multi-level multi-design point approach for gas turbine cycle and turbine conceptual design," PhD Thesis, 2017.

J. S. Gray, T. A. Hearn, K. T. Moore, J. Hwang, J. Martins, and A. Ning, "Automatic Evaluation of Multidisciplinary Derivatives Using a Graph-Based Problem Formulation in OpenMDAO," in *15th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, 2014.

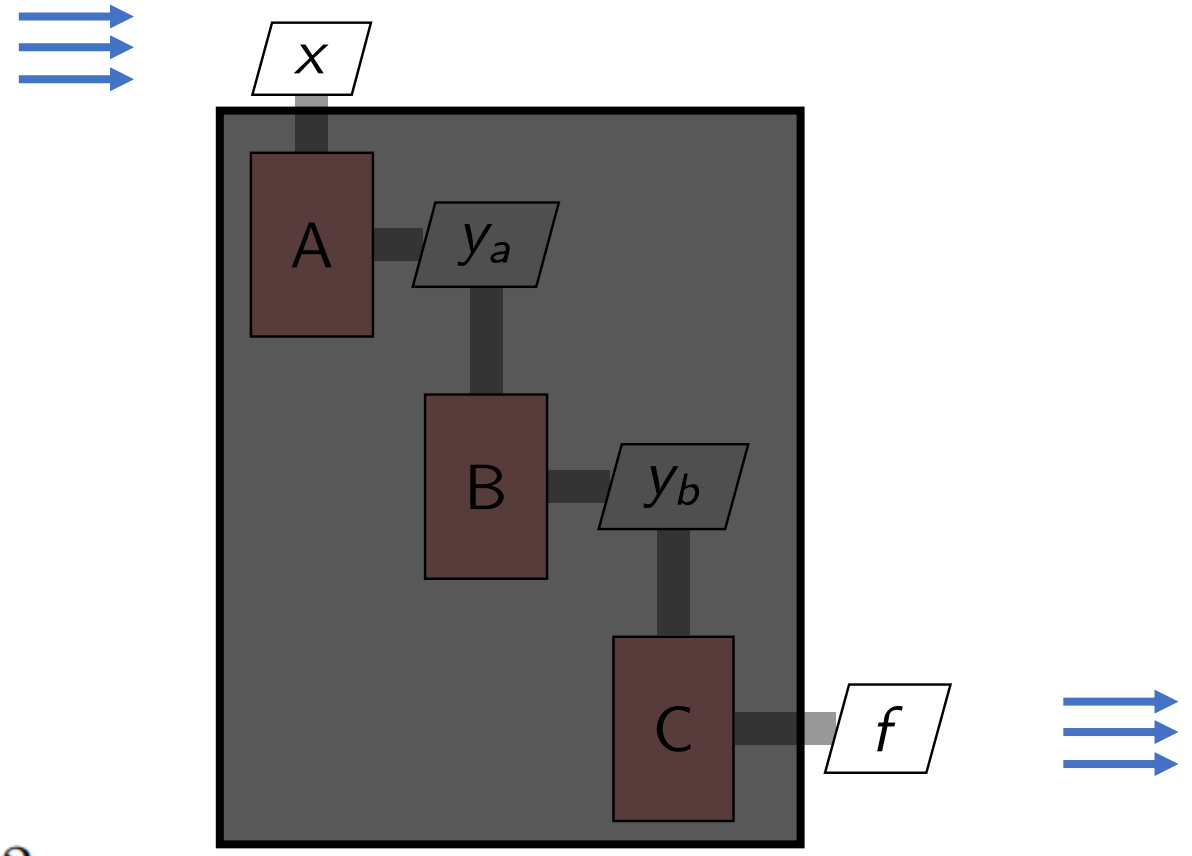
J. S. Gray, K. T. Moore, T. A. Hearn, and B. A. Naylor, "Standard Platform for Benchmarking Multidisciplinary Design Analysis and Optimization Architectures," *AIAA Journal*, vol. 51, iss. 10, p. 2380–2394, 2013.

C. Marriage and Martins, J. R. R. A. "Reconfigurable Semi-Analytic Sensitivity Methods and MDO Architectures Within the πMDO Framework", in *Proceedings of the 12th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Victoria, BC, 2008.

Complex step is more accurate than FD, but…

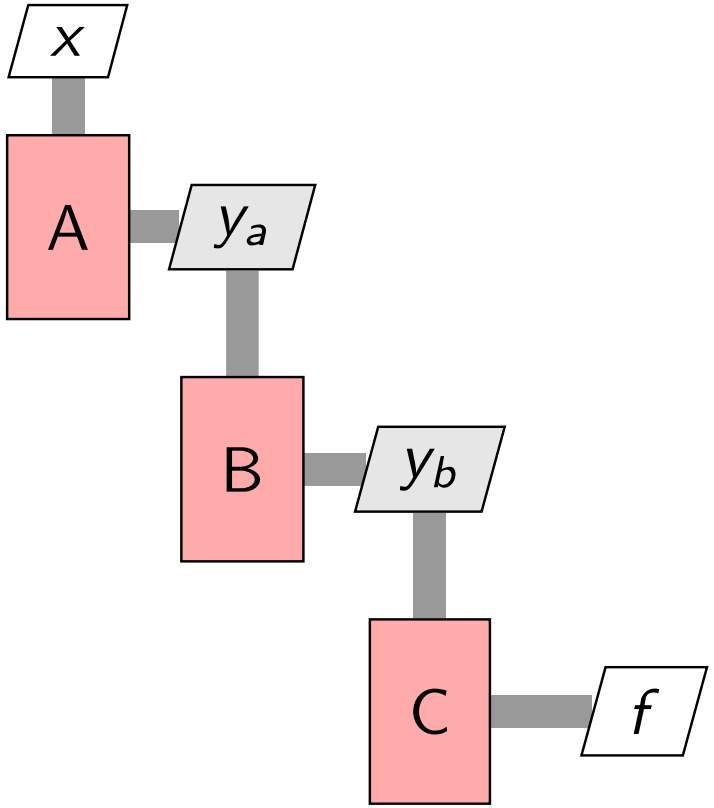
- Treats your models as a gray-box (might require code modification)
- Even more expensive
- Some functions are tricky…
  - min(), max(), abs() can cause issues

$$\frac{\partial \boldsymbol{F}}{\partial x_j} = \frac{\mathrm{Im}\left[\boldsymbol{F}(\boldsymbol{x} + ih\boldsymbol{e}_j)\right]}{h} + \mathcal{O}(h^2)$$



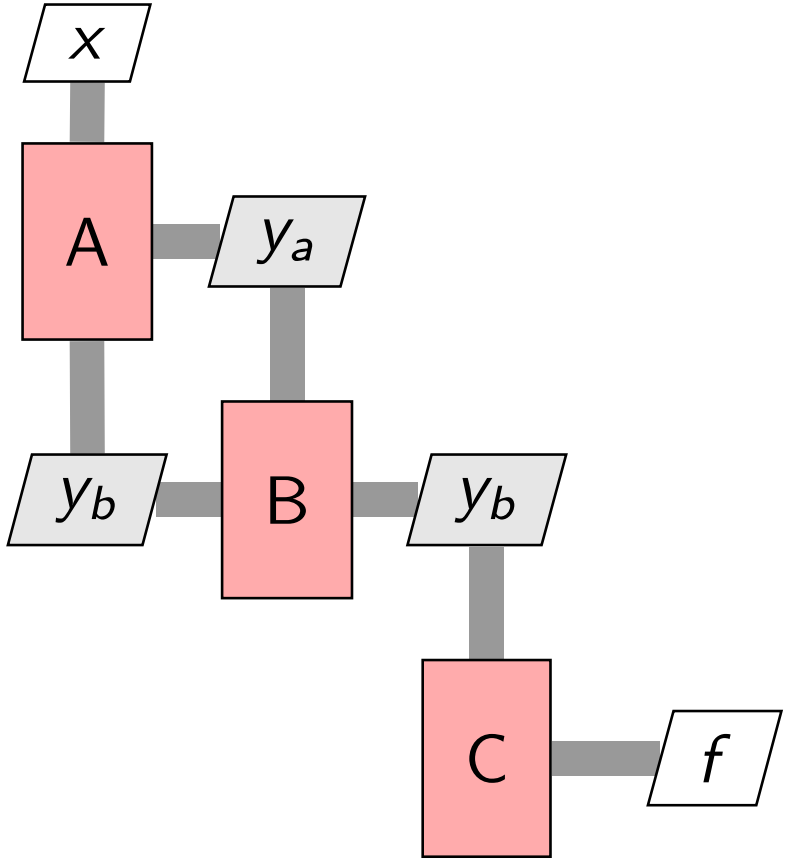
[Martins et al., *ACM TOMS,* 2003]

Analytic derivatives are both fast and accurate!

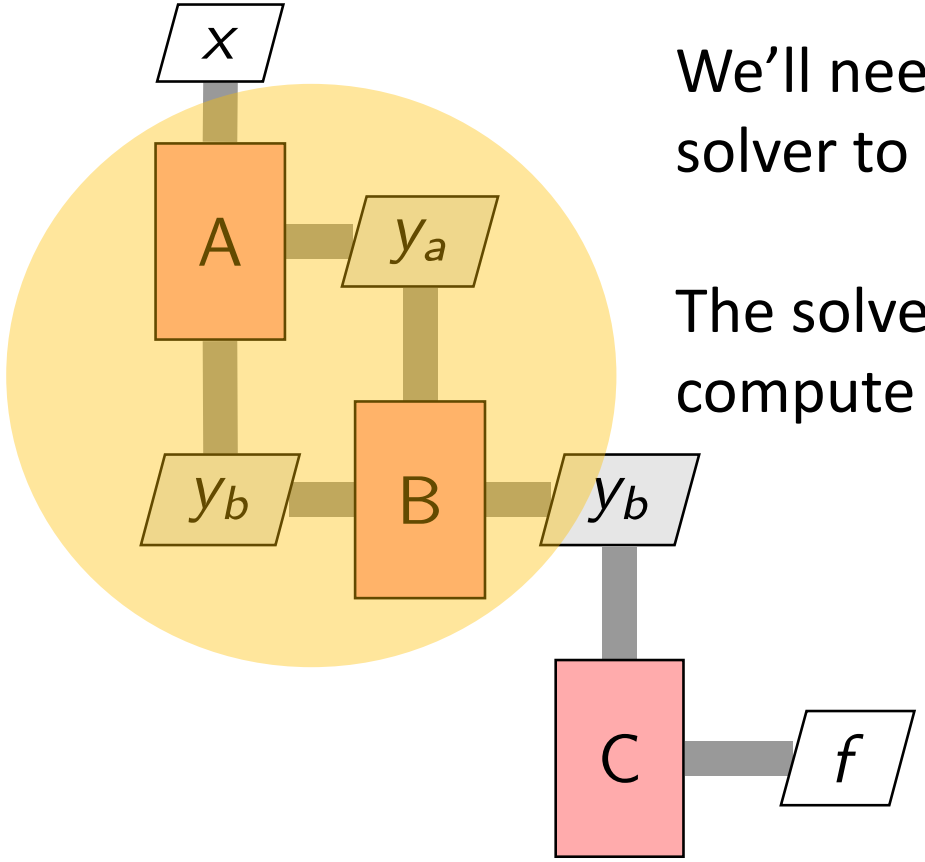


$$\frac{df}{dx} = \frac{\partial f}{\partial y_b}\frac{\partial y_b}{\partial y_a}\frac{\partial y_a}{\partial x}$$

What about models with coupling?



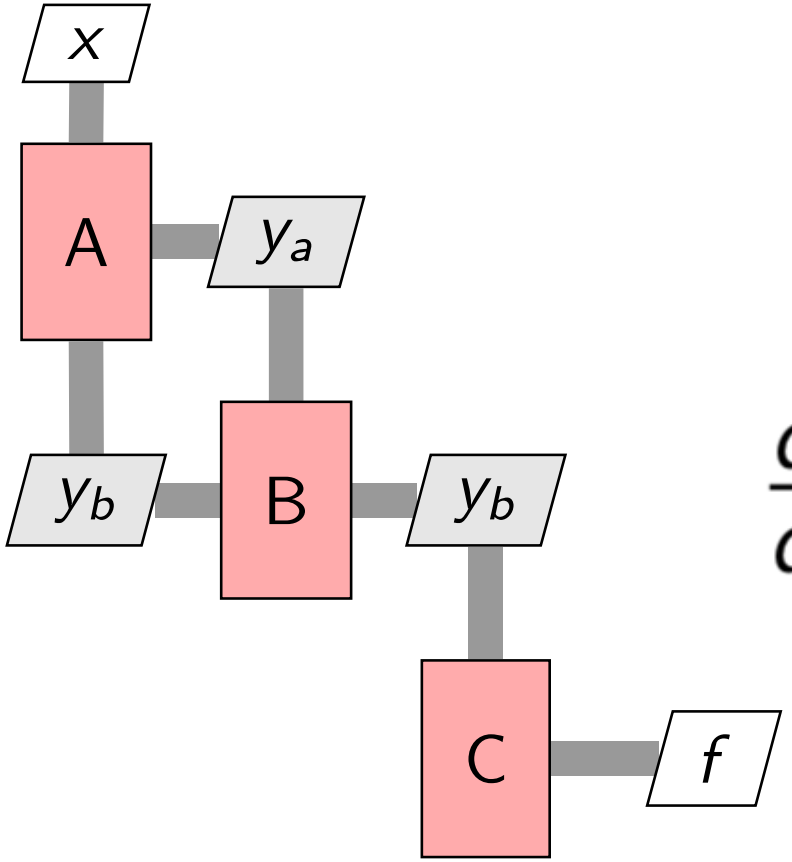
How do you differentiate through this convergence loop?



We'll need a nonlinear solver to converge this coupling.

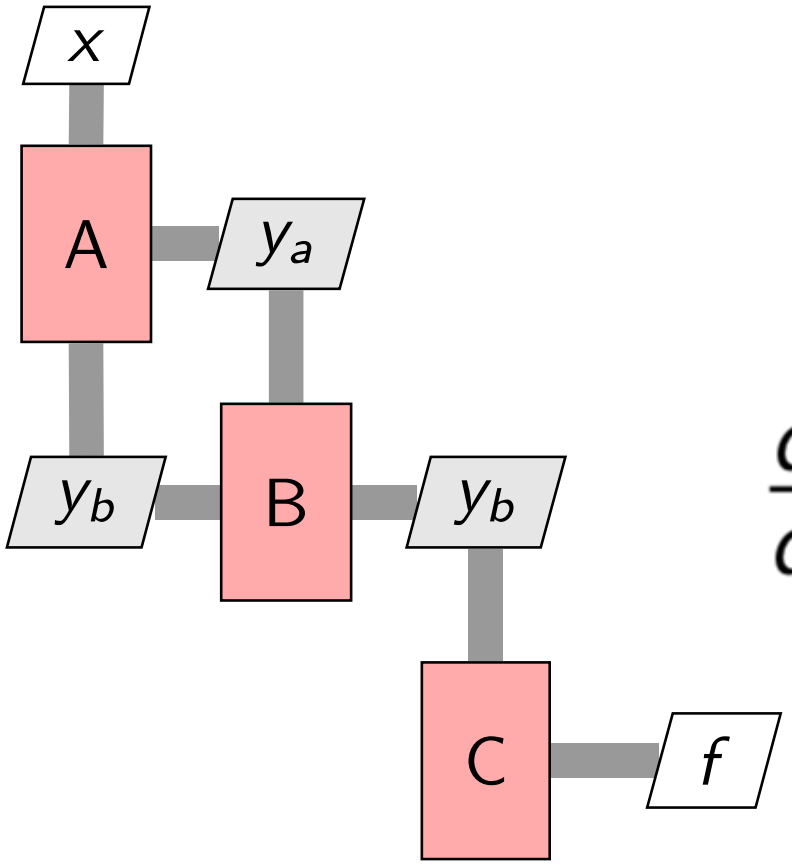
The solver loop changes the way you compute analytic derivatives

Manually computing analytic derivatives for coupled models takes a lot of work



$$\frac{df}{dx} = \frac{\partial f}{\partial y_b}\frac{\partial y_b}{\partial y_a}\frac{\partial y_a}{\partial x} + \frac{\partial f}{\partial y_b}\frac{dy_b}{dx} + \frac{\partial f}{\partial y_a}\frac{dy_a}{dx}$$

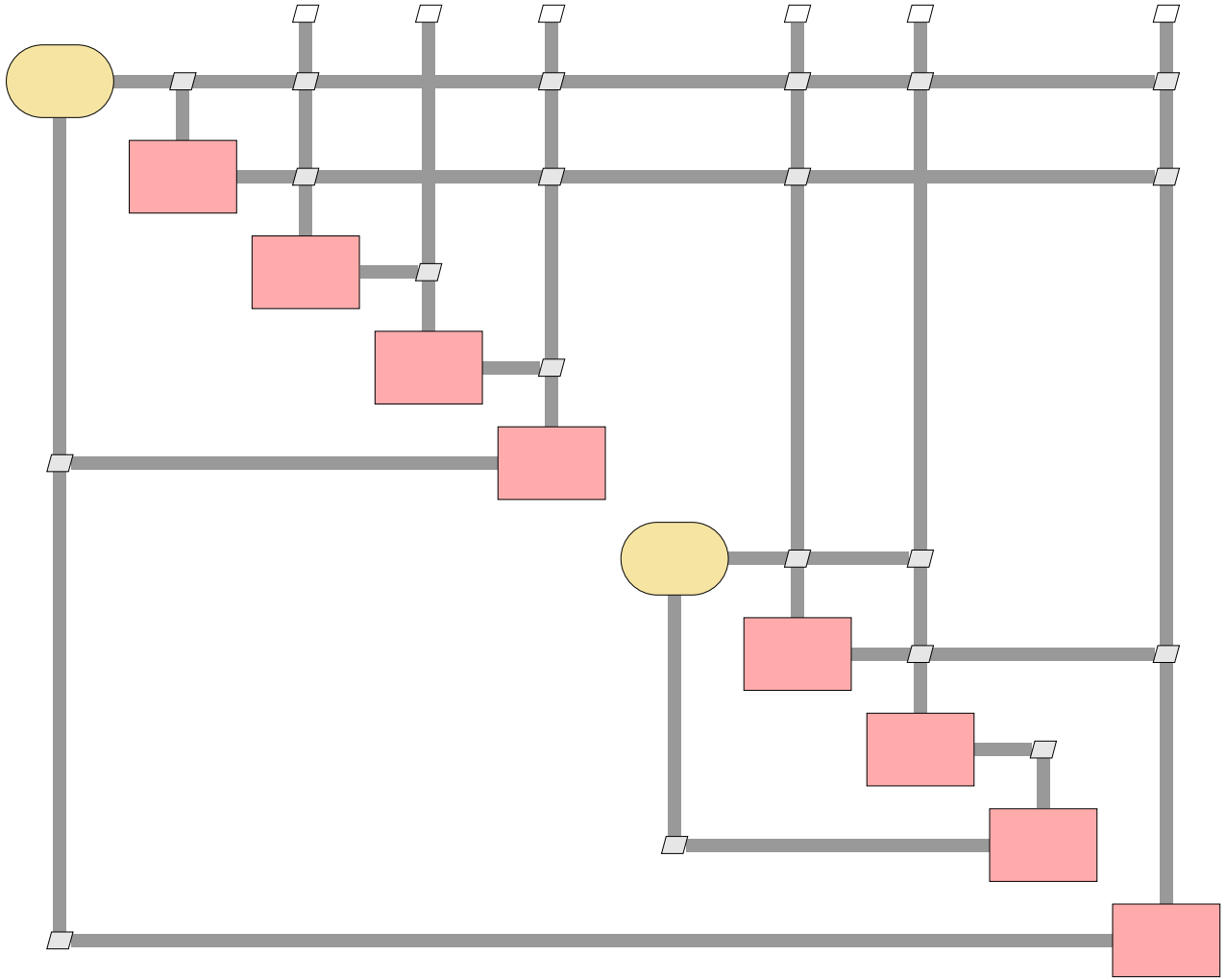
How do compute these extra terms?



$$\frac{df}{dx} = \frac{\partial f}{\partial y_b}\frac{\partial y_b}{\partial y_a}\frac{\partial y_a}{\partial x} + \frac{\partial f}{\partial y_b}\frac{dy_b}{dx} + \frac{\partial f}{\partial y_a}\frac{dy_a}{dx}$$

These terms will be computed using adjoints!

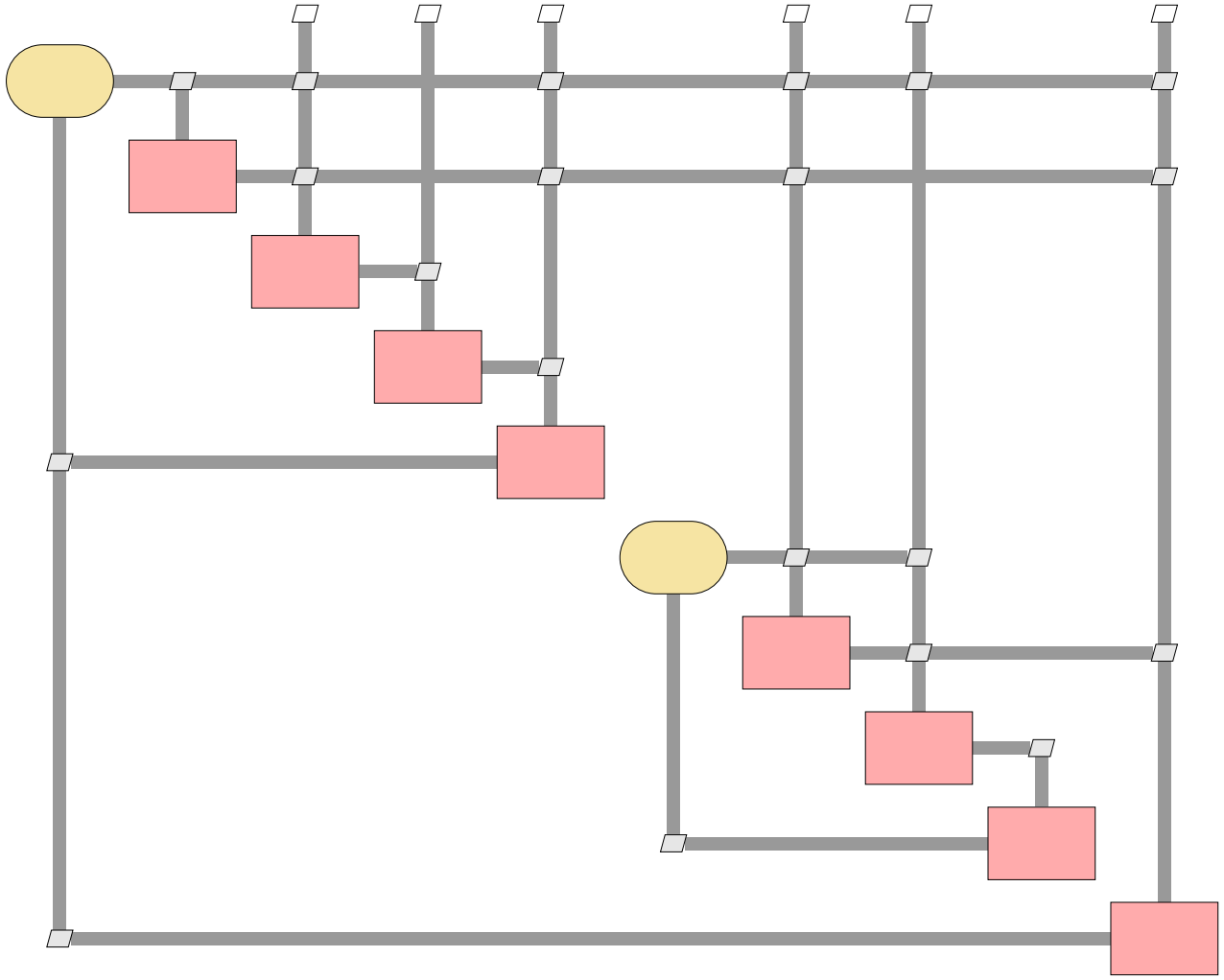
When models get really big … have fun!



$$\frac{df}{dx} = ???$$

Lets be honest, this just isn't going to happen!

OpenMDAO can compute these total derivatives for you, automatically!

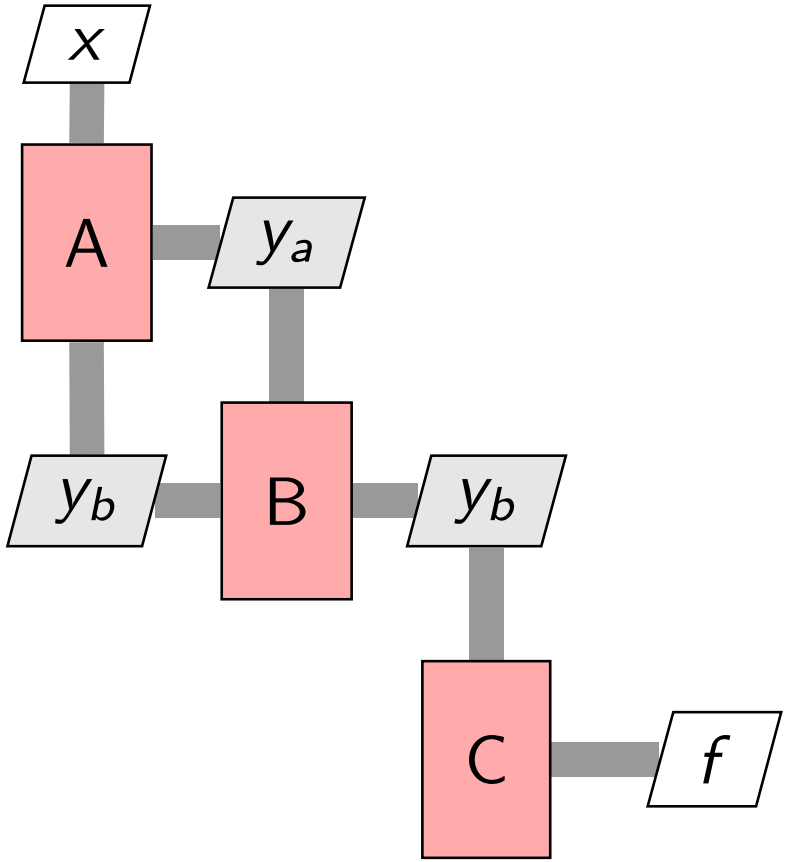


$$\frac{df}{dx} = \text{[ask OpenMDAO]}$$

For a deep dive on the math:

[Martins and Hwang 2013]
[Hwang and Martins 2018]

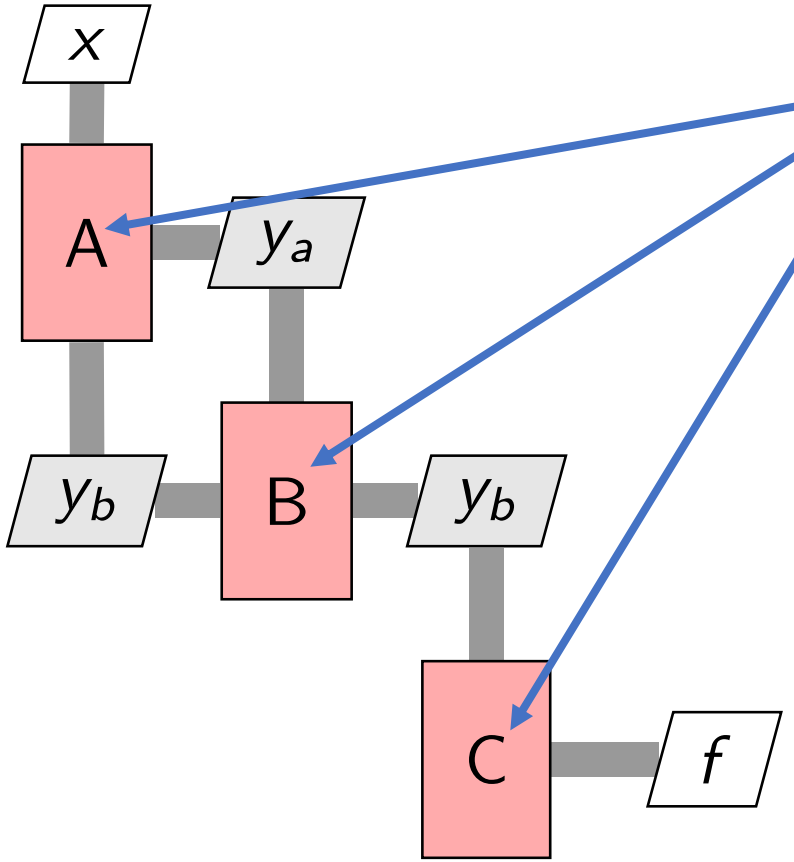
OpenMDAO splits total derivative computation into two steps



1) Computing the partial derivatives for each component

2) Solving a linear system for total derivatives

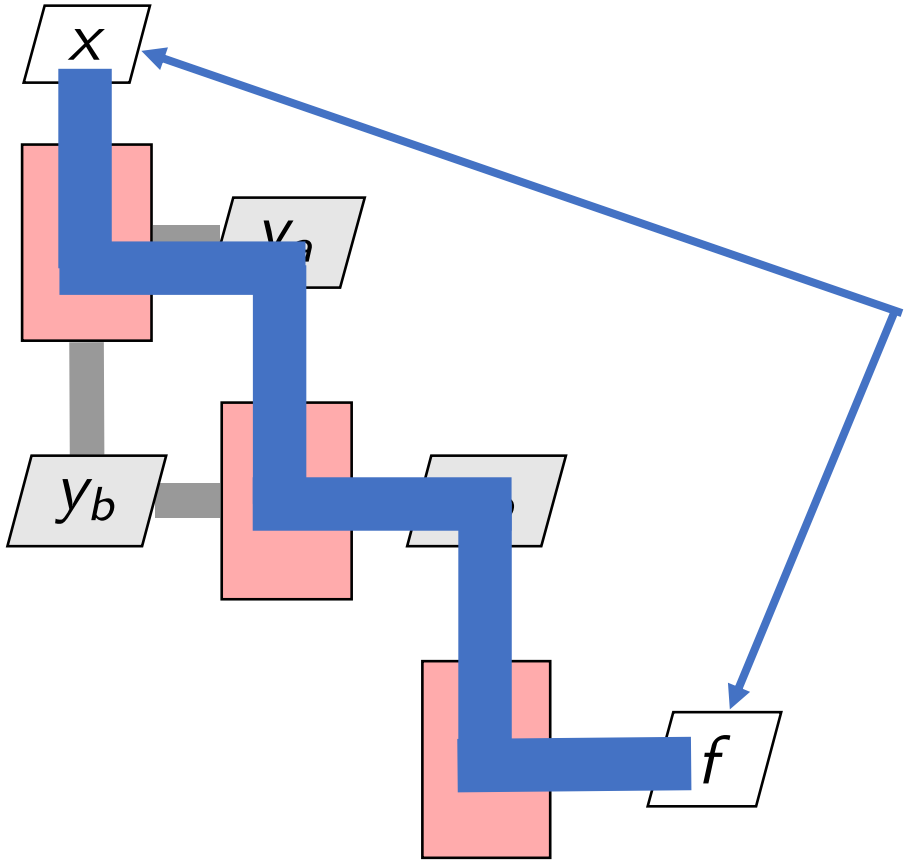
OpenMDAO splits total derivative computation into two steps



1) **Computing the partial derivatives for each component**
   - Numerical approach: FD/CS
   - Analytic approach: hand derivation or algorithmic differentiation
   - Mix and match numerical and analytic

You are responsible for this step, but OpenMDAO can help you a bit

OpenMDAO splits total derivative computation into two steps



1) Computing the partial derivatives for each component

2) Solving a linear system for total derivatives

OpenMDAO does this automatically using a library of different linear solvers for different problems

By using OpenMDAO's analytic derivative functionality you can

- Efficiently solve optimization problems with 1000's of design variables or thousands of constraints
- Get total derivatives across a complicated model, by providing only partial derivatives of each component
- Mix and match different techniques for computing partial derivatives

Rules of thumb for computing partials

- Fine to start out with FD partials
  while getting your model set up!

- For cheap black-box codes:
  - FD *might* be ok for production runs
  - CS is better if you can do it!

- For “expensive” codes (anything with an internal solver):
  - Implement as ImplicitComponent
    (expose the states/residuals to OpenMDAO)
  - Can still FD the residual if you need to
  - Analytic derivatives typically cost a lot for these components

Optimization

Optimization is handled by a `Driver`

```
# build the model
prob = Problem()
indeps = prob.model.add_subsystem('indeps', IndepVarComp(), promotes=['*'])
indeps.add_output('a', .5)
indeps.add_output('Area', 10.0, units='m**2')
indeps.add_output('rho', 1.225, units='kg/m**3')
indeps.add_output('Vu', 10.0, units='m/s')

prob.model.add_subsystem('a_disk', ActuatorDisc(),
                         promotes_inputs=['a', 'Area', 'rho', 'Vu'])

# setup the optimization
prob.driver = ScipyOptimizeDriver()
prob.driver.options['optimizer'] = 'SLSQP'

prob.model.add_design_var('a', lower=0., upper=1.)
prob.model.add_design_var('Area', lower=0., upper=1.)
prob.model.add_constraint('a_disk.Ct',lower=0.,upper=0.8)
# negative one so we maximize the objective
prob.model.add_objective('a_disk.Cp', scaler=-1)
prob.model.approx_totals(method='fd')
prob.setup(mode='fwd')
prob.run_driver()
```

ScipyOptimizeDriver is what most people will use on Windows

add_design_var(<var_name>, lower=…, upper=…)

add_constraint(<var_name>, lower=…, upper=…)

add_objective(<var_name>)

run_driver instead of run_model

Optimization

Available drivers:

- `ScipyOptimizeDriver` will work in Anaconda
  - Works on Windows machines
  - SLSQP method is the standard constrained opt in OpenMDAO

- `PyoptSparseDriver` is much better (SNOPT algorithm) but really only builds on Linux/MacOSX
  (hard to build on windows, but technically is possible)

Recording the optimization history

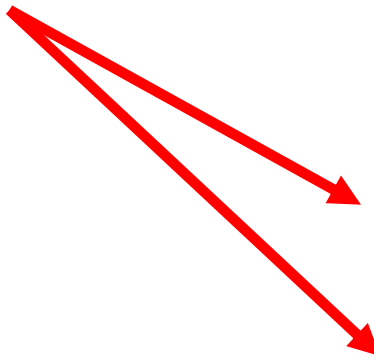
- Often you will want to record variable values changing over the course of an optimization.
- This is achieved using a “recorder” object
- This will record all the objective, all constraints, and all DVs for each optimizer iteration (but no “intermediate variables”)
- Can then read in the data to postprocess

```python
import openmdao.api as om
from openmdao.test_suite.components.sellar import SellarDerivatives

import numpy as np

prob = om.Problem(model=SellarDerivatives())

model = prob.model
model.add_design_var('z', lower=np.array([-10.0, 0.0]),
                          upper=np.array([10.0, 10.0]))
model.add_design_var('x', lower=0.0, upper=10.0)
model.add_objective('obj')
model.add_constraint('con1', upper=0.0)
model.add_constraint('con2', upper=0.0)

prob.driver = om.ScipyOptimizeDriver(optimizer='SLSQP', tol=1e-9)

recorder = om.SqliteRecorder("cases.sql")

prob.driver.add_recorder(recorder)
prob.driver.recording_options['includes'] = []
prob.driver.recording_options['record_objectives'] = True
prob.driver.recording_options['record_constraints'] = True
prob.driver.recording_options['record_desvars'] = True


prob.add_recorder(recorder)
prob.recording_options['includes'] = ['*']

prob.setup()
prob.run_driver()
prob.record_iteration('final')
```

Saving all the variables for the final optimized point

- We don't normally save all the variables for every iteration to avoid making huge database files

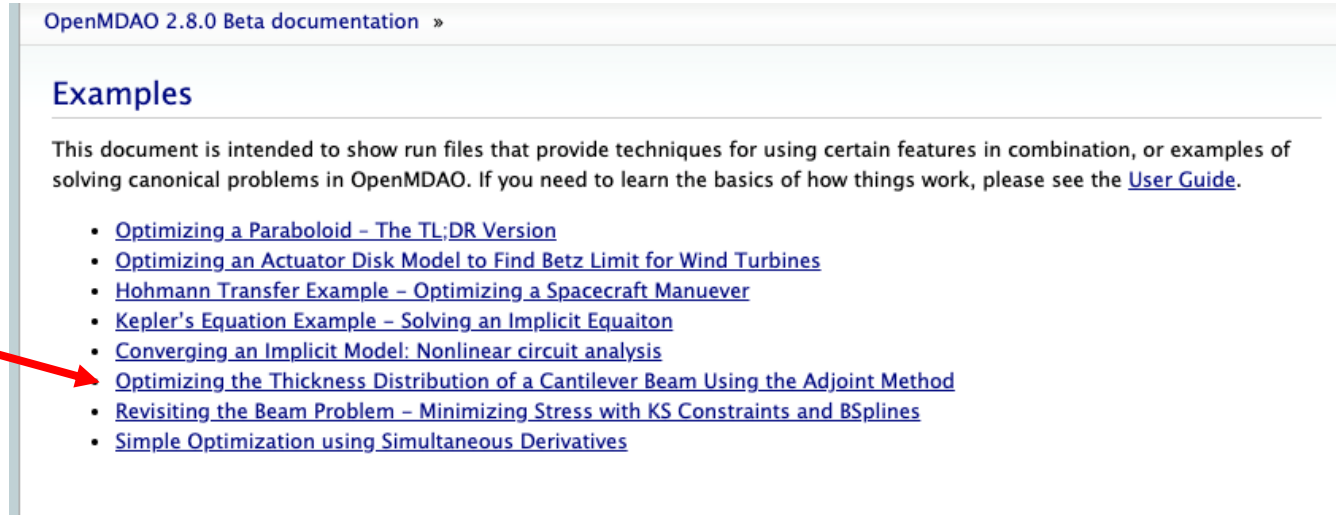
- This configures a secondary case type to record specific runs with all the variables



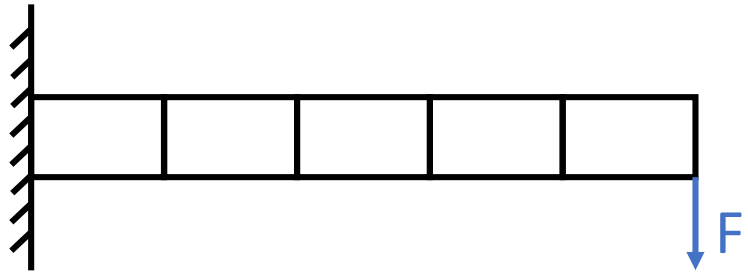
```
import openmdao.api as om
from openmdao.test_suite.components.sellar import SellarDerivatives

import numpy as np

prob = om.Problem(model=SellarDerivatives())

model = prob.model
model.add_design_var('z', lower=np.array([-10.0, 0.0]),
                          upper=np.array([10.0, 10.0]))
model.add_design_var('x', lower=0.0, upper=10.0)
model.add_objective('obj')
model.add_constraint('con1', upper=0.0)
model.add_constraint('con2', upper=0.0)

prob.driver = om.ScipyOptimizeDriver(optimizer='SLSQP', tol=1e-9)

recorder = om.SqliteRecorder("cases.sql")

prob.driver.add_recorder(recorder)
prob.driver.recording_options['includes'] = []
prob.driver.recording_options['record_objectives'] = True
prob.driver.recording_options['record_constraints'] = True
prob.driver.recording_options['record_desvars'] = True


prob.add_recorder(recorder)
prob.recording_options['includes'] = ['*']

prob.setup()
prob.run_driver()
prob.record_iteration('final')
```

Lab #2: Optimization of a cantilever beam

- Find the thickness distribution for a cantilever beam that minimizes compliance subject to a volume constraint



- Check out the other examples on your own!



Lab #2: Simple FEM in 5 steps!

0) Break the beam into segments
(0 doesn't count as a real step!)

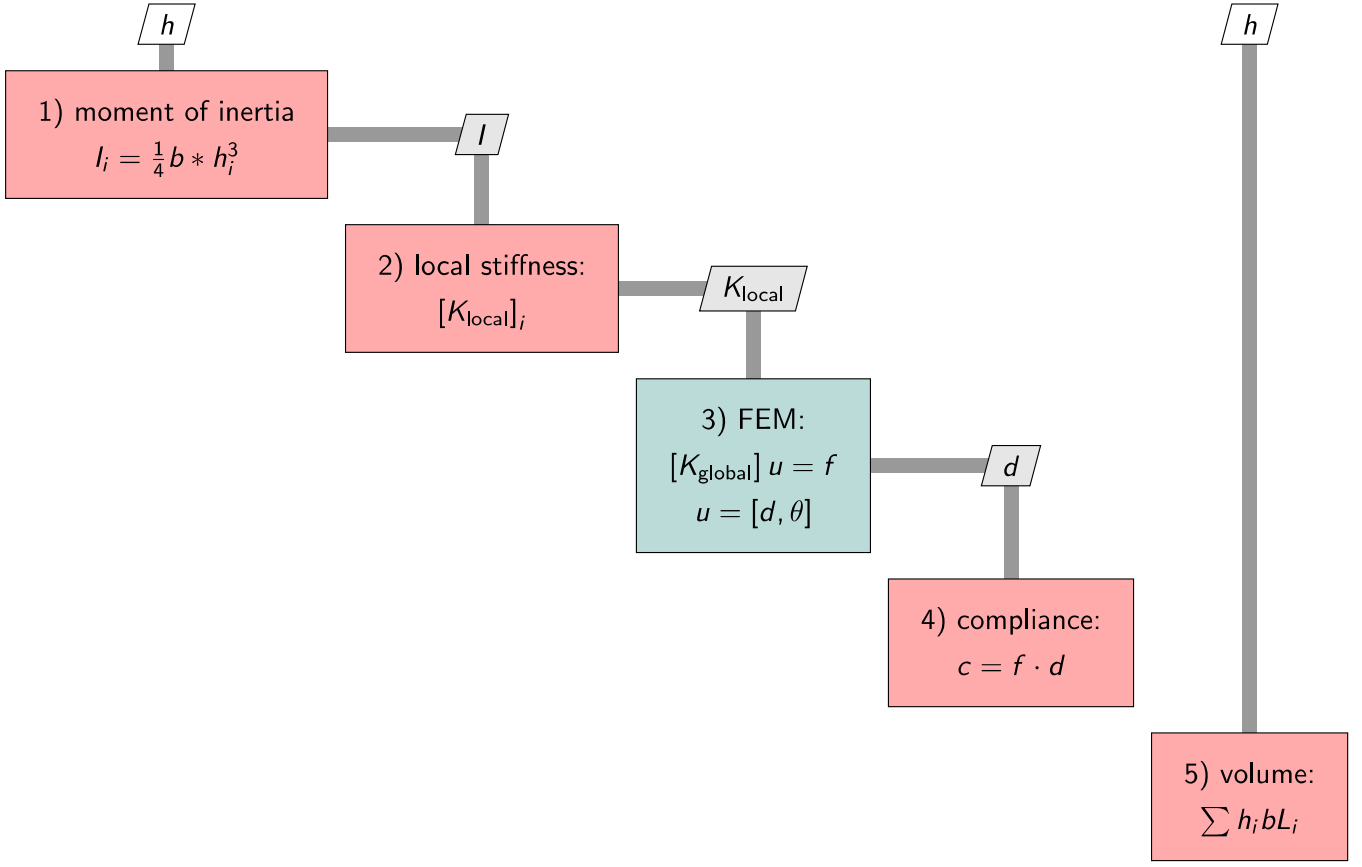
1) Compute the moment of inertia for each segment

2) Compute the local stiffness matrix for each segment

3) Combine all the local stiffness matrices into a global K matrix and solve the FEM

4) Pull displacements from the state vector and compute compliance

5) Compute beam volume



Lab #2: Pay close attention to step 4!

0) Break the beam into segments (0 doesn't count as a real step!)

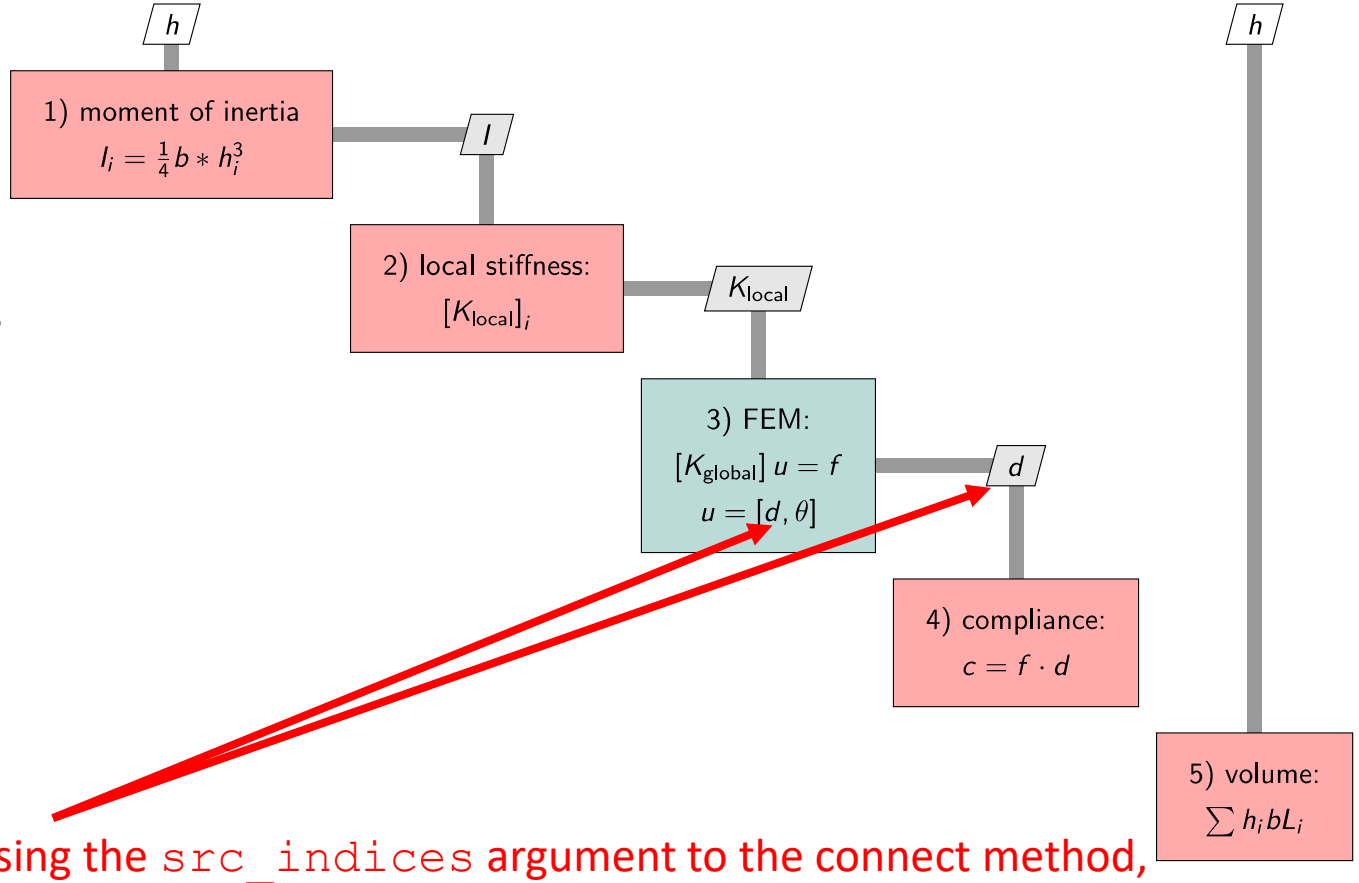
1) Compute the moment of inertia for each segment

2) Compute the local stiffness matrix for each segment

3) Combine all the local stiffness matrices into a global K matrix and solve the FEM

4) Pull displacements from the state vector and compute compliance

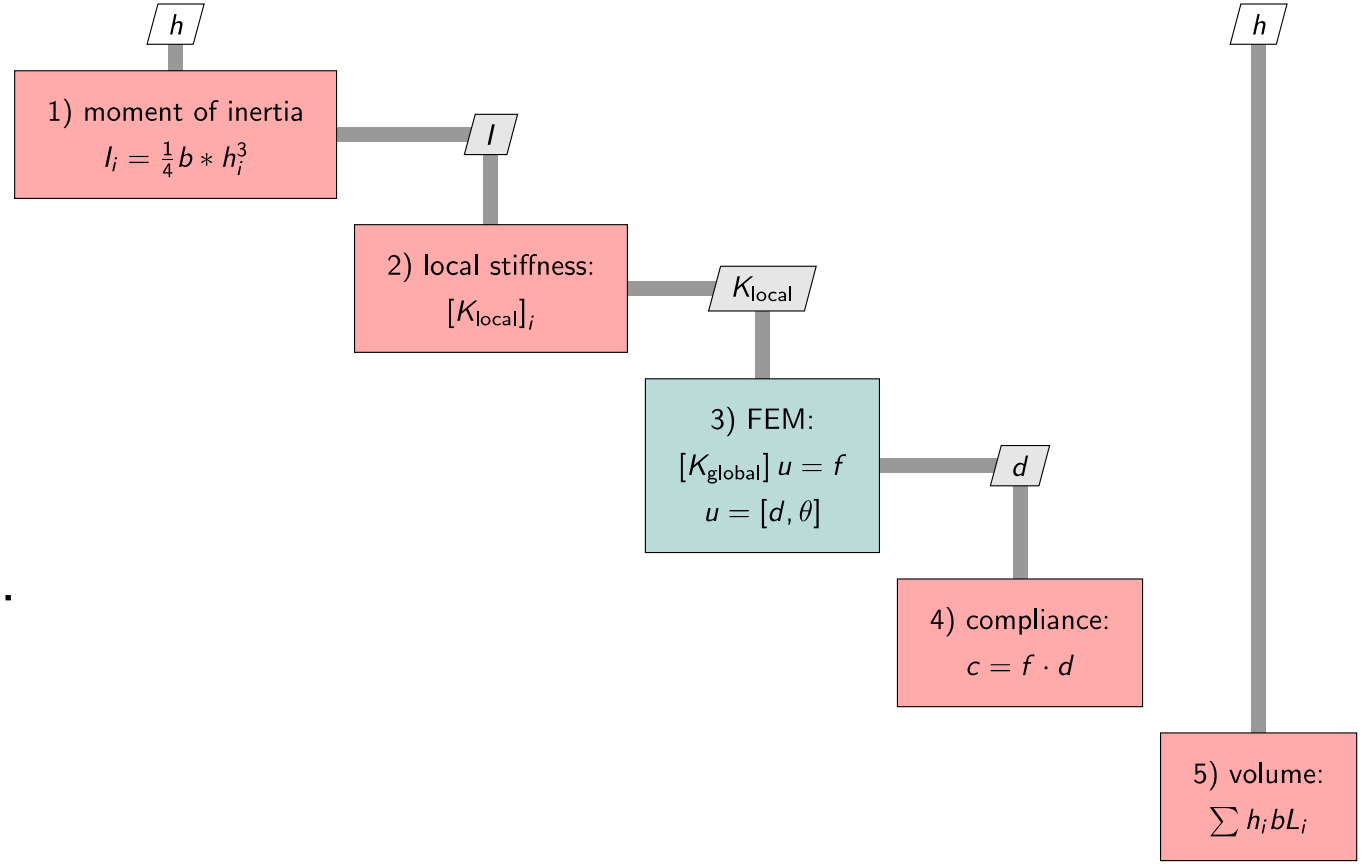
5) Compute beam volume



using the `src_indices` argument to the connect method, you can extract just the relevant portion of the state vector for the compliance calculation

Lab #2: Get to it!

- Copy the "`lab_2_template.py`" file to a new file called "`lab_2.py`"
- Add components in the correct order (~ line 40)
- Connect the components together (~line 48)
- Use the N2 diagram to make sure you have everything connected (no red inputs!)
- What happens if you switch to full model FD? Does it work? Try changing the FD step size…
- What about complex step?
- How does compute cost scale with `num_elements` for different methods of computing derivatives?
- What happens if you add the components in the incorrect order?



Wrapping external codes

- How to handle file wrappers with OpenMDAO's built in helper components

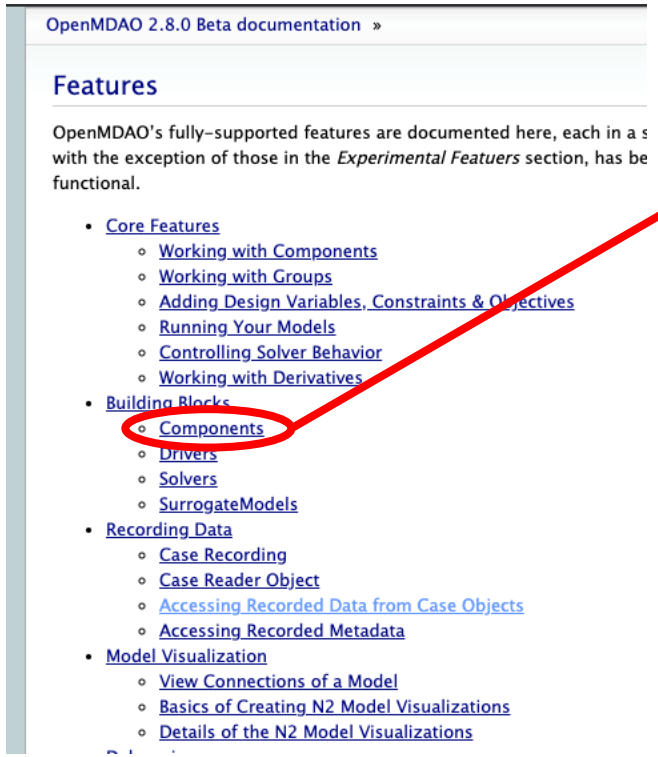
- Lab 3: Explicit and Implicit file wrappers

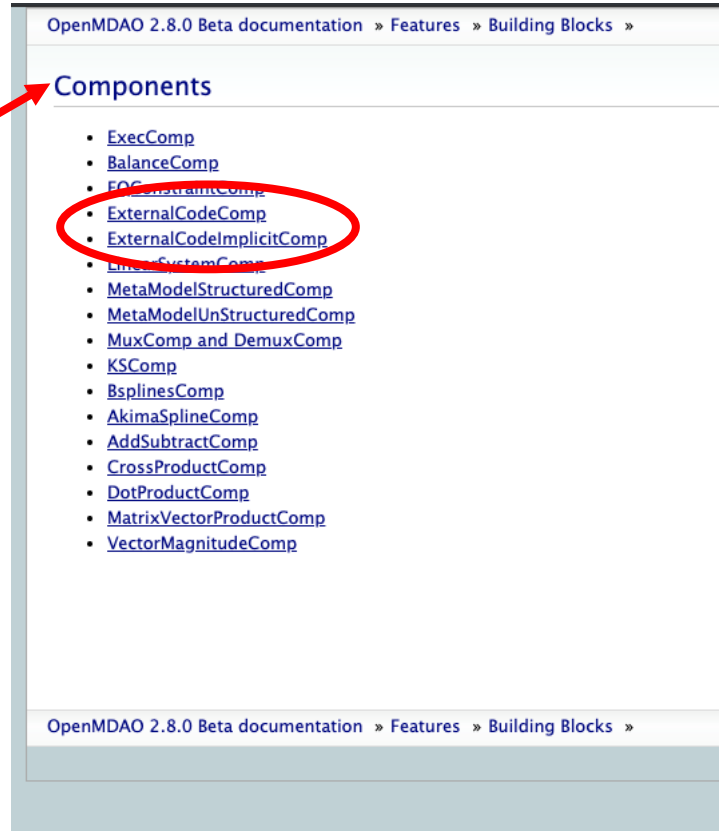
OpenMDAO has two file wrapper components in the standard library

There are lots of useful components
In the standard library!

spines, constraint aggregation,
file wrappers, Ax=b, Ax, etc…

Have a look through these comps,
So you know whats there





ExternalCodeComp

Inherit from OpenMDAO provided class

Define the I/O like normal

Wrapper specific options

```python
class ParaboloidExternalCodeComp(om.ExternalCodeComp):
    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)

        self.input_file = 'paraboloid_input.dat'
        self.output_file = 'paraboloid_output.dat'

        # providing these is optional; the component will verify that any input
        # files exist before execution and that the output files exist after.
        self.options['external_input_files'] = [self.input_file]
        self.options['external_output_files'] = [self.output_file]

        self.options['command'] = [
            'python', 'extcode_paraboloid.py', self.input_file, self.output_file
        ]

    def compute(self, inputs, outputs):
        x = inputs['x']
        y = inputs['y']

        # generate the input file for the paraboloid external code
        with open(self.input_file, 'w') as input_file:
            input_file.write('%.16f\n%.16f\n' % (x, y))

        # the parent compute function actually runs the external code
        super(ParaboloidExternalCodeComp, self).compute(inputs, outputs)

        # parse the output file from the external code and set the value of f_xy
        with open(self.output_file, 'r') as output_file:
            f_xy = float(output_file.read())

        outputs['f_xy'] = f_xy
```

Write the input file

Create a sub-process to run the code

Parse the output file

Push the values back into OpenMDAO's output vector

ExternalCodeImplicitComp

```python
class MachExternalCodeComp(om.ExternalCodeImplicitComp):

    def initialize(self):
        self.options.declare('super_sonic', types=bool)

    def setup(self):
        self.add_input('area_ratio', val=1.0, units=None)
        self.add_output('mach', val=1., units=None)
        self.declare_partials(of='mach', wrt='area_ratio', method='fd')

        self.input_file = 'mach_input.dat'
        self.output_file = 'mach_output.dat'

        # providing these are optional; the component will verify that any input
        # files exist before execution and that the output files exist after.
        self.options['external_input_files'] = [self.input_file]
        self.options['external_output_files'] = [self.output_file]

        self.options['command_apply'] = [
            'python', 'extcode_mach.py', self.input_file, self.output_file,
        ]
        self.options['command_solve'] = [
            'python', 'extcode_mach.py', self.input_file, self.output_file,
        ]

    def apply_nonlinear(self, inputs, outputs, residuals):
        with open(self.input_file, 'w') as input_file:
            input_file.write('residuals\n')
            input_file.write('{}\n'.format(inputs['area_ratio'][0]))
            input_file.write('{}\n'.format(outputs['mach'][0]))

        # the parent apply_nonlinear function actually runs the external code
        super(MachExternalCodeComp, self).apply_nonlinear(inputs, outputs, residuals)

        # parse the output file from the external code and set the value of mach
        with open(self.output_file, 'r') as output_file:
            mach = float(output_file.read())
        residuals['mach'] = mach

    def solve_nonlinear(self, inputs, outputs):
        with open(self.input_file, 'w') as input_file:
            input_file.write('outputs\n')
            input_file.write('{}\n'.format(inputs['area_ratio'][0]))
            input_file.write('{}\n'.format(self.options['super_sonic']))
        # the parent apply_nonlinear function actually runs the external code
        super(MachExternalCodeComp, self).solve_nonlinear(inputs, outputs)

        # parse the output file from the external code and set the value of mach
        with open(self.output_file, 'r') as output_file:
            mach = float(output_file.read())
        outputs['mach'] = mach
```

Inherit from OpenMDAO provided class (different base class then for the explicit case!)

Define the I/O like normal

Wrapper specific options

Write the input file

apply_nonlinear is the Residual evaluation

Create a sub-process to run the code

Parse the output file

solve_nonlinear asks the code to converge itself using its own internal solvers

Lab #3: Beam optimization with file wrapped external codes

1. cd into the `lab_3` folder
2. Have a quick look at standalone_beam.py
   this is a pure python version of the same FEM you built in OpenMDAO for Lab #2 (but without derivatives)
3. Call `` `python standalone_beam.py opt` `` from the command line
4. Call `` `python lab_3_explicit_wrapper.py` ``
5. Call `` `python lab_3_implicit_wrapper.py` ``
6. Why does the implicit wrapper call both solve_nonlinear and apply_nonlinear? (hint: it has to do with derivatives)
7. Compare the compute cost of running the optimization using pure OpenMDAO, the standalone optimization, and the file wrapped version.

Advanced Topics

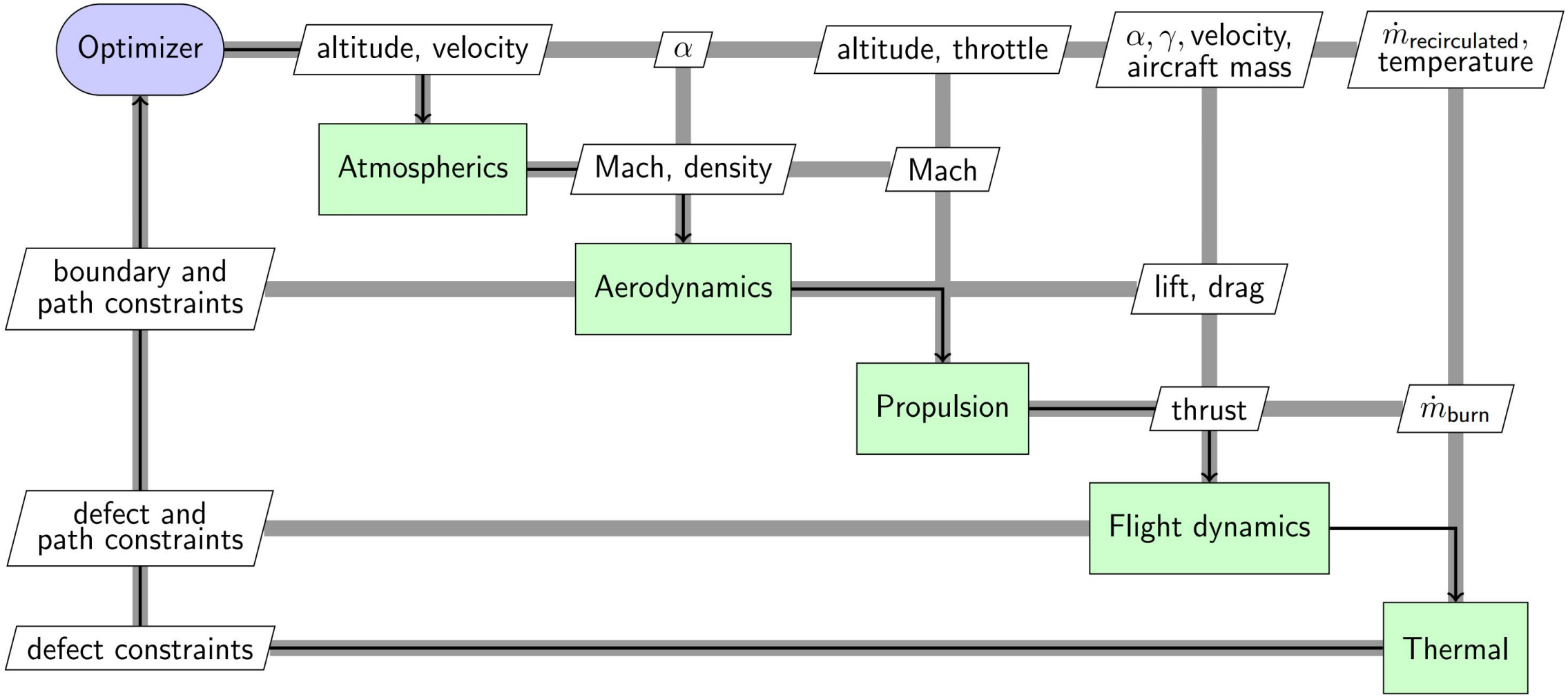
- Model architecting
- Vectorization
- Implicit components
- Getting system derivatives efficiently
- High-fidelity and expensive tools
- Debugging tools and processes
- Surrogate modeling
- Trajectory optimization
- Propulsion analysis

Model architecting

- Developing good OpenMDAO models is like object-oriented software engineering: **you need to pick the right level to draw the component boundary**
- In my opinion, a good component has:
  - on the order of 5 – 20 inputs
  - 1 – 20 outputs
  - a few computations that are logically related
- Python's package import system makes this easy

- Real-world examples:
- https://github.com/mdolab/OpenConcept
- https://github.com/mdolab/OpenAeroStruct

Model architecting example



Vectorization

- Where possible, avoid instantiating models over and over for multi-point analyses (e.g. trajectory optimization)
- Instead, do vectorized computations using numpy
- OpenMDAO needs to know the dimension of all component inputs / outputs *at setup() time*
  - Best practice: pass in vector length as a `self.option['vec_size']`
  - If you mess this up you'll get dimension mismatch errors at run time

Vectorization

User sets vector dims when instantiating the model

```
def initialize(self):
    self.options.declare('num_nodes', default=1, desc="Number of nodes to compute")

def setup(self):
    nn = self.options['num_nodes']
    arange = np.arange(0, nn)
    self.add_input('fltcond|CL', shape=(nn,))
    self.add_input('fltcond|q', units='N * m**-2', shape=(nn,))
    self.add_input('ac|geom|wing|S_ref', units='m **2')
    self.add_input('CD0')
    self.add_input('e')
    self.add_input('ac|geom|wing|AR')
    self.add_output('drag', units='N', shape=(nn,))

    self.declare_partials(['drag'], ['fltcond|CL', 'fltcond|q'], rows=arange, cols=arange)
    self.declare_partials(['drag'],
                          ['ac|geom|wing|S_ref', 'ac|geom|wing|AR', 'CD0', 'e'],
                          rows=arange, cols=np.zeros(nn))
```

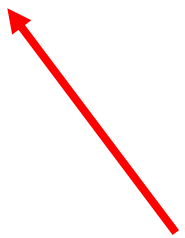
shape=(dim1, dim2,…)

Sparse Jacobian indices for vector input and vector output

Sparse Jacobian indices for scalar input and vector output

Vectorization

```
    self.declare_partials(['drag'], ['fltcond|CL', 'fltcond|q'], rows=arange, cols=arange)
    self.declare_partials(['drag'],
                          ['ac|geom|wing|S_ref', 'ac|geom|wing|AR', 'CD0', 'e'],
                          rows=arange, cols=np.zeros(nn))

def compute(self, inputs, outputs):
    outputs['drag'] = (inputs['fltcond|q'] * inputs['ac|geom|wing|S_ref'] *
                       (inputs['CD0'] + inputs['fltcond|CL']**2 / np.pi / inputs['e'] /
                        inputs['ac|geom|wing|AR']))
```



Many flight conditions computed at once using vectorized computation

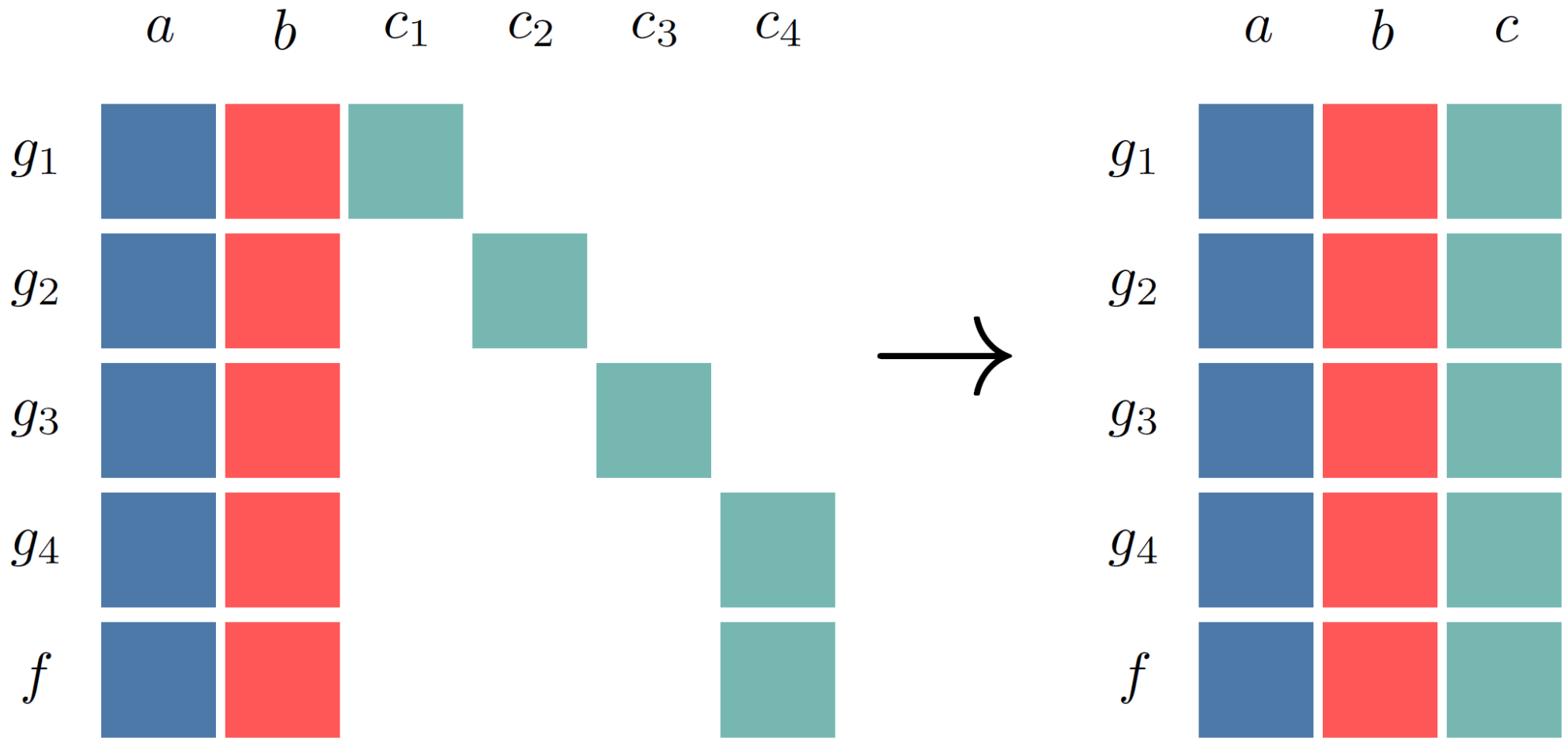
Implicit Components

- Useful for computations where there isn't an easily written explicit expression
  - $\cos(x \cdot y) - z \cdot y = 0$ y is a state and x,z are inputs
  - CFD or FEA analyses

- You compute the residuals for the states instead of the outputs directly
- Implicit components always require a solver

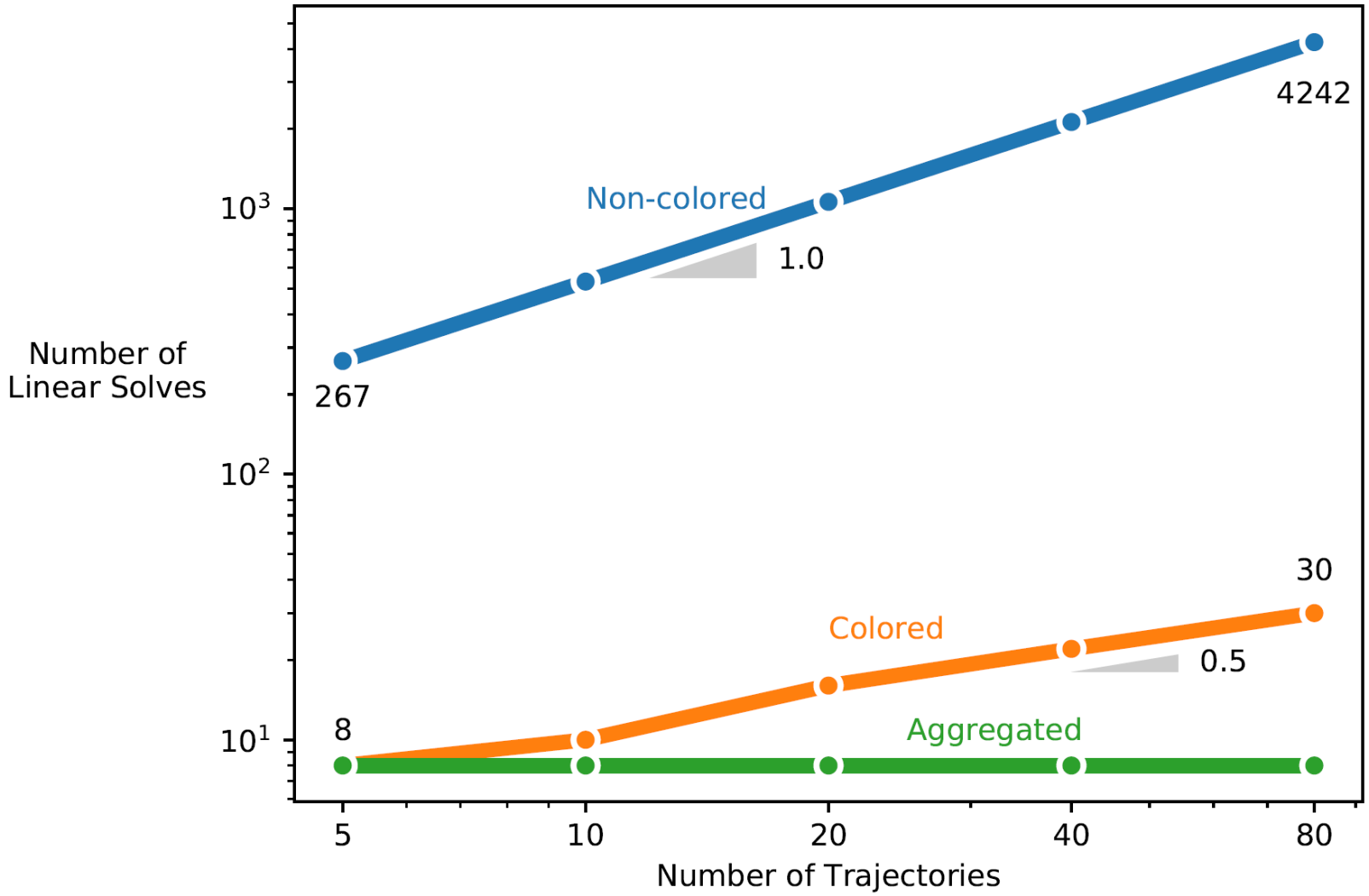
http://openmdao.org/twodocs/versions/latest/advanced_guide/implicit_comps/defining_icomps.html

Getting system derivatives efficiently: automatic Jacobian coloring



http://openmdao.org/twodocs/versions/latest/features/core_features/working_with_derivatives/simul_derivs.html

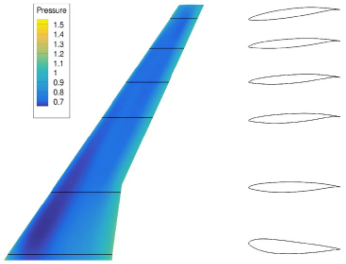
Getting system derivatives efficiently: automatic Jacobian coloring



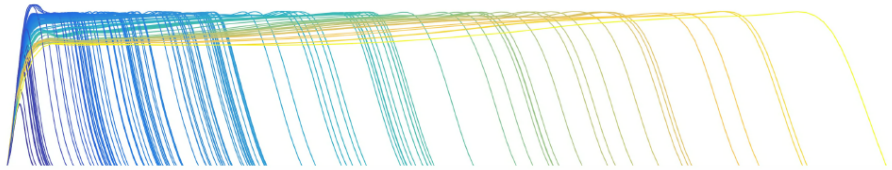
High-fidelity and expensive tools

- OpenMDAO can handle a mix of fidelity levels easily
- Can parallelize at the group or subgroup level
- Only requiring the partial derivatives becomes quite important here

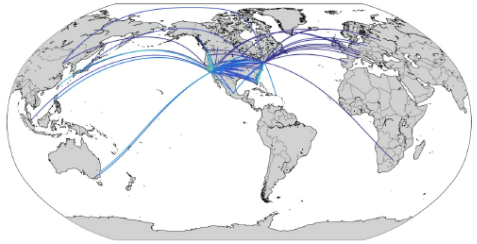
High-fidelity and expensive tools



Design



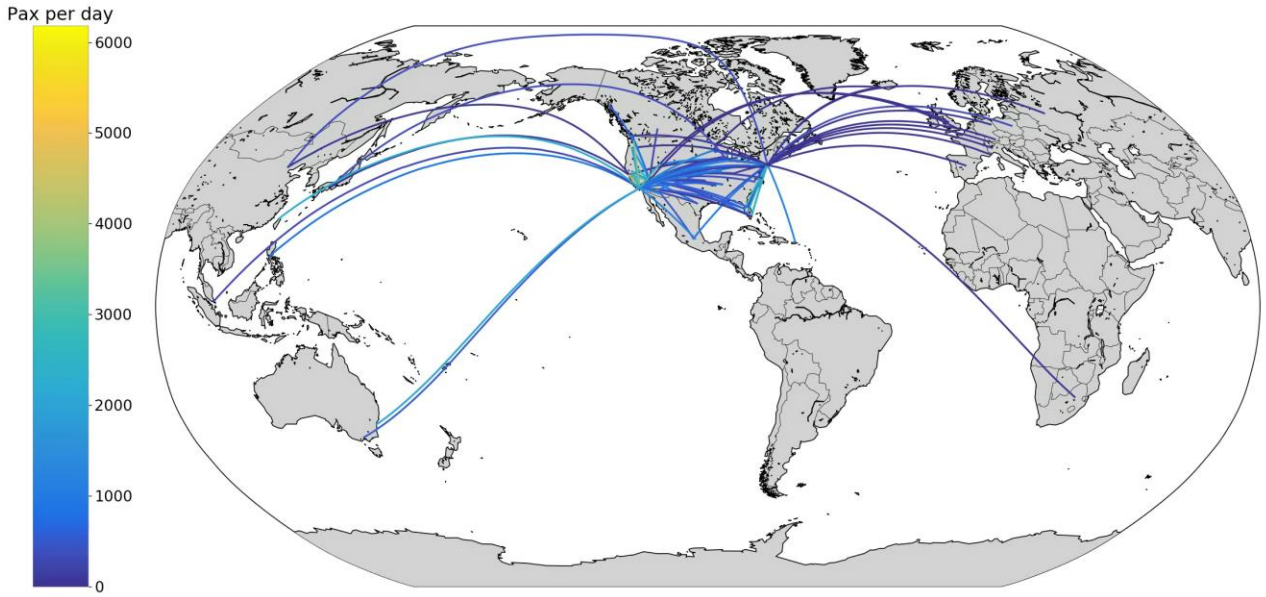
Mission (128 routes)

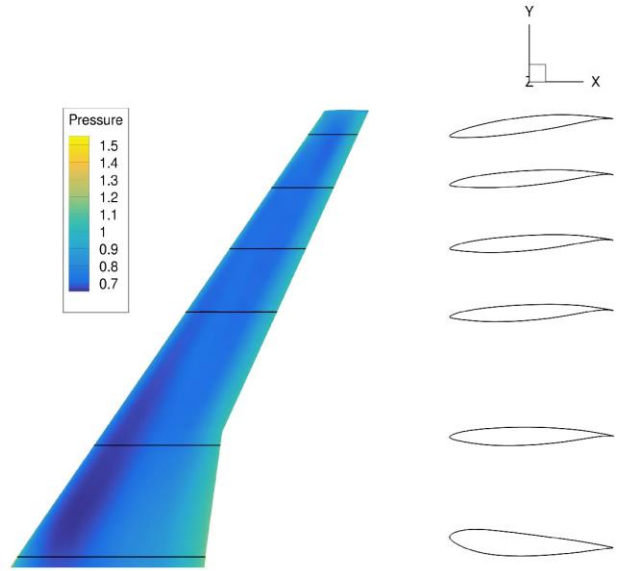


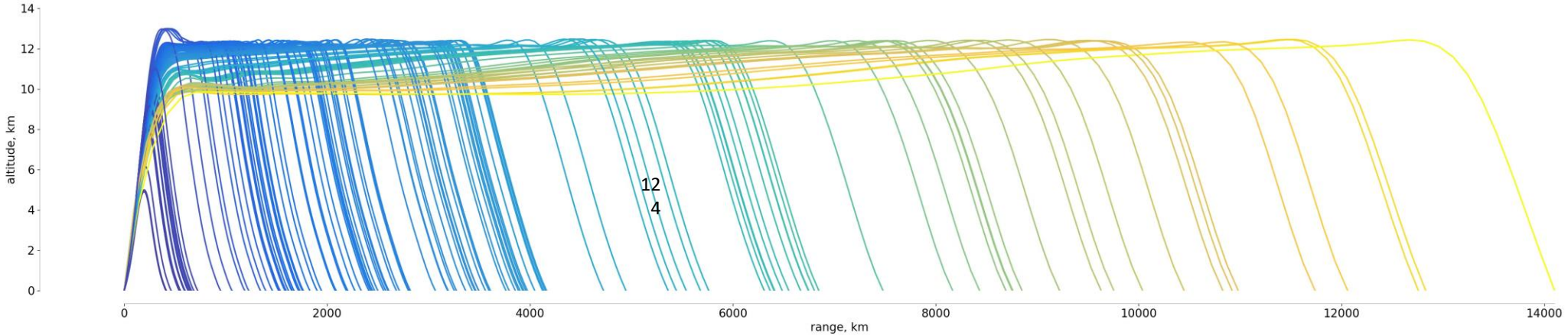
Allocation

Objective		Profit	1	
Design variables	(design)	Shape	192	4169 design variables
		Twist	7	
		Area	1	
		Sweep	1	
	(mission)	Altitude	2560	
		Cruise Mach	128	
	(allocation)	Pax / flight	640	
		Flight / day	640	
Constraints	(design)	LE / TE	16	13956 constraints
		Thickness	750	
		Volume	1	
	(mission)	Climb angle	12800	
		Thrust limits	256	
	(allocation)	Demand	128	
		Aircraft fleet	5	

High-fidelity and expensive tools

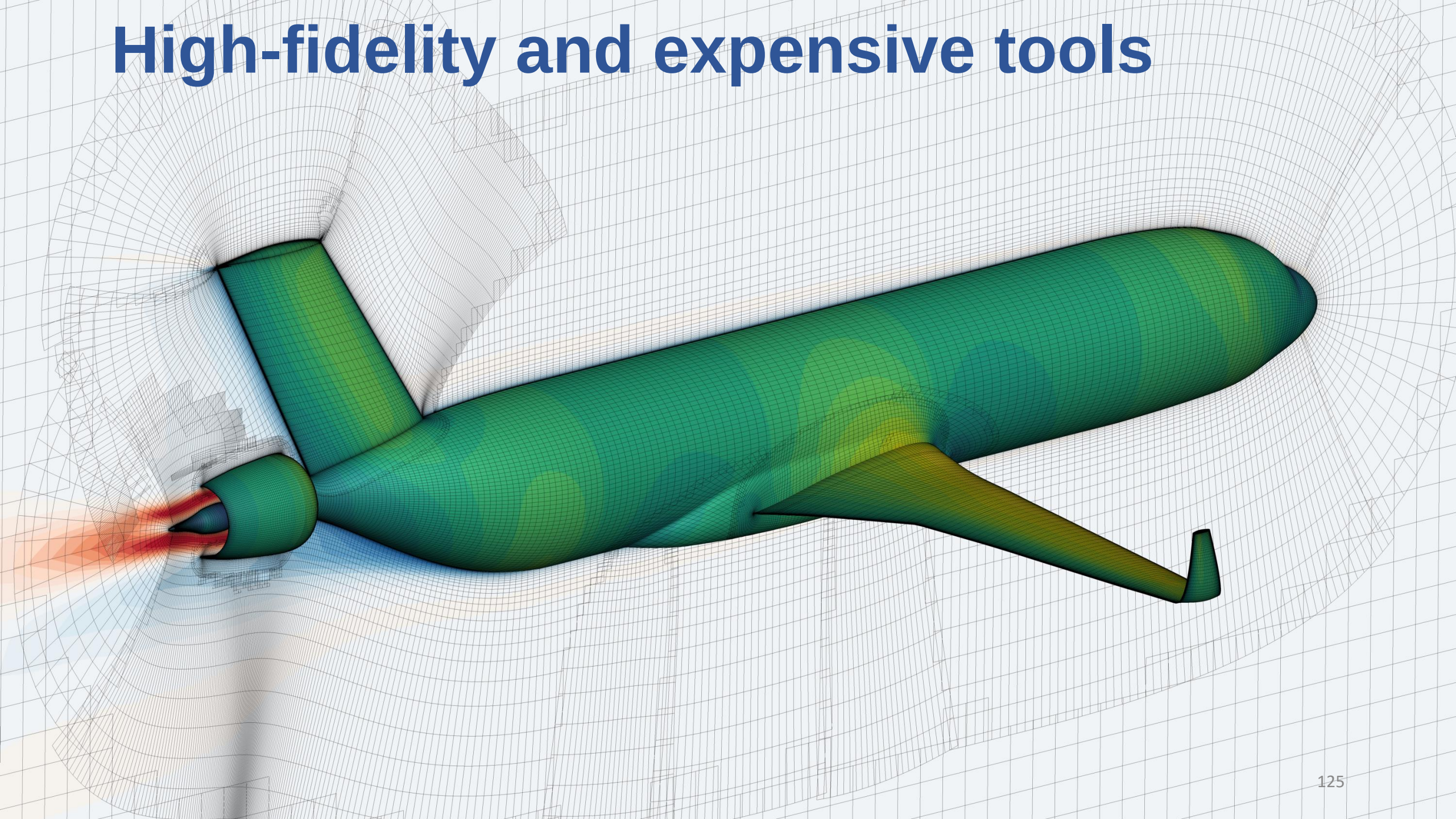






[Hwang et al., *AIAA 2019-1.C035082*]

High-fidelity and expensive tools



Debugging tools

- Many tools come pre-packaged in OpenMDAO to help you develop, debug, and run your models
  - N2
  - Connection viewer
  - Speed and memory profiling
  - Call tracing

http://openmdao.org/twodocs/versions/latest/other/om_command.html

Connection viewer: openmdao view_connections <file.py>

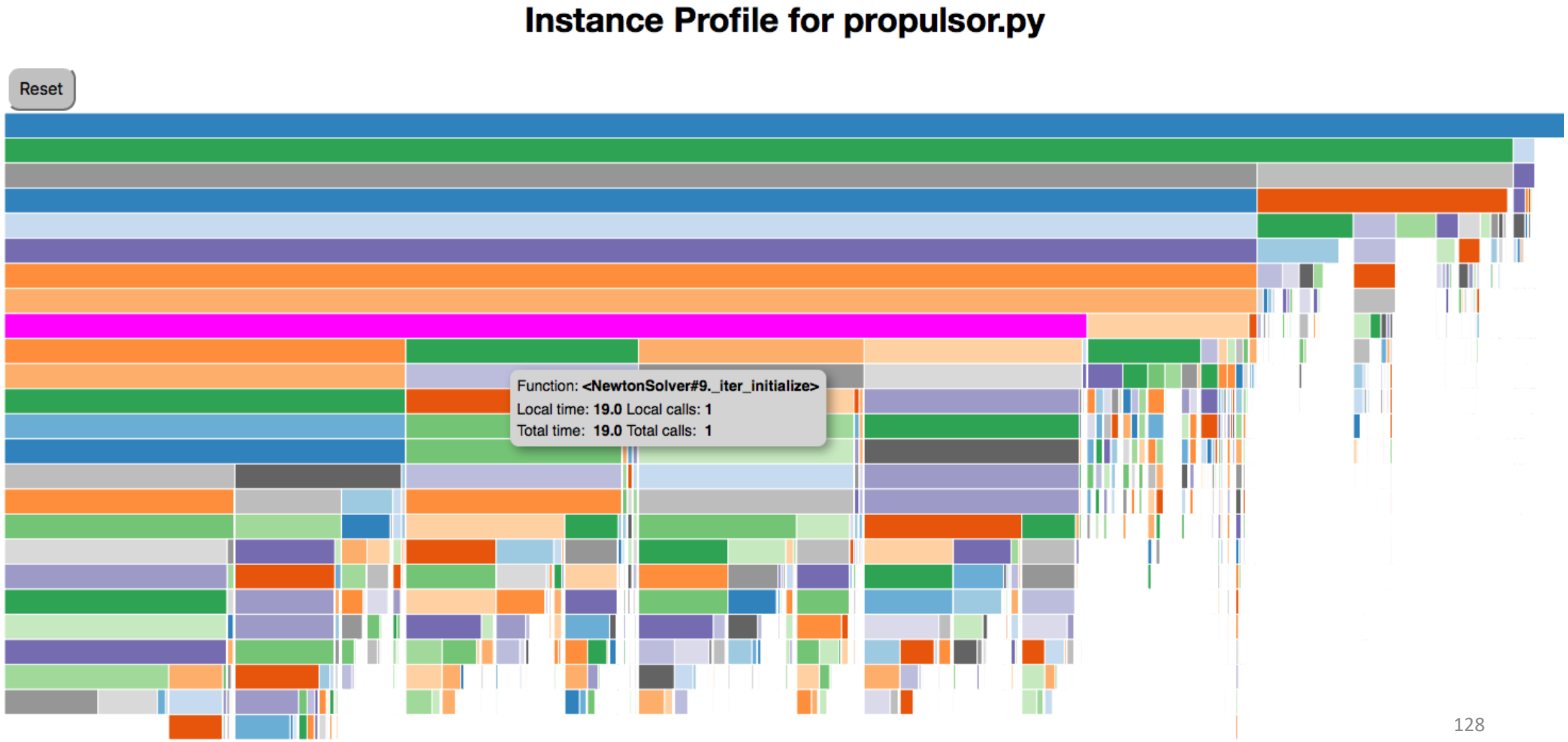
Filter by Source System

Filter by Target System

design.fan

Source	Units	Value	Units	Target
NO CONNECTION		[0.5]		design.fan.out_stat.statics.ps_resid.MN
NO CONNECTION		[0.5]		design.fan.out_stat.flow_static.MN
NO CONNECTION		[0.96]		design.fan.map.scalars.effDes
NO CONNECTION		[0.]		design.fan.map.readMap.alphaMap
NO CONNECTION		[2.]		design.fan.map.readMap.RlineMap
NO CONNECTION		[0.]		design.fan.map.desMap.alphaMap
NO CONNECTION		[2.]		design.fan.map.desMap.RlineMap
NO CONNECTION		[1.]	rpm	design.fan.map.desMap.NcMap
NO CONNECTION		[1.]	lbm/s	design.fan.flow_in.Fl_I:stat:Wc
NO CONNECTION		[1.]		design.fan.flow_in.Fl_I:FAR
NO CONNECTION		[0.]		design.fan.FAR_passthru.Fl_I:FAR
des_vars.FPR		[1.2]		**design.fan.map.scalars.PRdes**
design.fan.blds_pwr.W_out	lbm/s	[0.453592]	kg/s	design.fan.out_stat.statics.ps_resid.W
design.fan.blds_pwr.W_out	lbm/s	[1.]	lbm/s	design.fan.out_stat.flow_static.W
design.fan.corrinputs.Nc	rpm	[100.]	rpm	design.fan.map.scalars.Nc
design.fan.corrinputs.Nc	rpm	[100.]	rpm	design.fan.map.shaftNc.Nc
design.fan.corrinputs.Wc	lbm/s	[30.]	lbm/s	design.fan.map.scalars.Wc
design.fan.enth_rise.ht_out	Btu/lbm	[0.555927]	cal/g	design.fan.real_flow.chem_eq.h
design.fan.enth_rise.ht_out	Btu/lbm	[1.]	Btu/lbm	design.fan.real_flow.flow.h
design.fan.enth_rise.ht_out	Btu/lbm	[1.]	Btu/lbm	design.fan.blds_pwr.ht_out
design.fan.ideal_flow.chem_eq.T	degK	[400.]	degK	design.fan.ideal_flow.props.TP2ls.T
design.fan.ideal_flow.chem_eq.T	degK	[400.]	degK	design.fan.ideal_flow.props.tp2props.T
design.fan.ideal_flow.chem_eq.T	degK	[720.]	degR	design.fan.ideal_flow.flow.T
design.fan.ideal_flow.chem_eq.b0		[1. 1. 1. 1.]		design.fan.ideal_flow.props.TP2ls.b0
design.fan.ideal_flow.chem_eq.n		View		design.fan.ideal_flow.props.TP2ls.n
design.fan.ideal_flow.chem_eq.n		View		design.fan.ideal_flow.props.tp2props.n
design.fan.ideal_flow.chem_eq.n		View		design.fan.ideal_flow.flow.n
design.fan.ideal_flow.chem_eq.n_moles		[0.034]		design.fan.ideal_flow.props.TP2ls.n_moles
design.fan.ideal_flow.chem_eq.n_moles		[0.034]		design.fan.ideal_flow.props.tp2props.n_moles
design.fan.ideal_flow.chem_eq.n_moles		[0.034]		design.fan.ideal_flow.flow.n_moles
design.fan.ideal_flow.props.TP2ls.lhs_TP		View		design.fan.ideal_flow.props.ls2t.A
design.fan.ideal_flow.props.TP2ls.lhs_TP		View		design.fan.ideal_flow.props.ls2p.A
design.fan.ideal_flow.props.TP2ls.rhs_P		View		design.fan.ideal_flow.props.ls2p.b
design.fan.ideal_flow.props.TP2ls.rhs_T		View		design.fan.ideal_flow.props.ls2t.b
design.fan.ideal_flow.props.ls2p.x		View		design.fan.ideal_flow.props.tp2props.result_P
design.fan.ideal_flow.props.ls2t.x		View		design.fan.ideal_flow.props.tp2props.result_T
design.fan.ideal_flow.props.tp2props.Cp	cal/(g*degK)	[0.999331]	Btu/(lbm*degR)	design.fan.ideal_flow.flow.Cp
design.fan.ideal_flow.props.tp2props.Cv	cal/(g*degK)	[0.999331]	Btu/(lbm*degR)	design.fan.ideal_flow.flow.Cv

Speed and memory profiling: openmdao iprof <file.py>



Call tracing:
openmdao call_tree <method>

```
openmdao call_tree openmdao.api.LinearBlockGS.solve
```

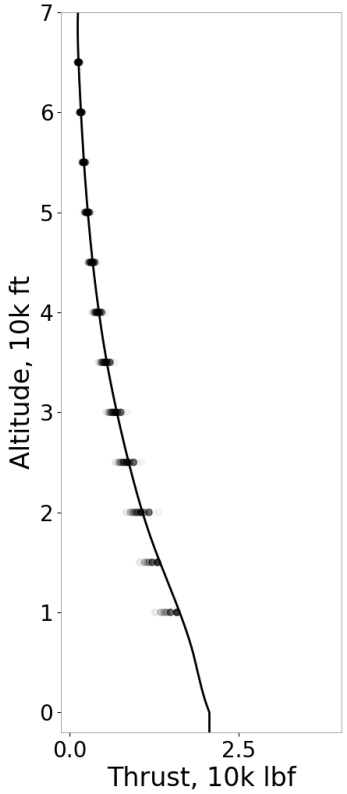
```
BlockLinearSolver.solve
  Solver._solve
    Solver._mpi_print_header
    BlockLinearSolver._iter_initialize
      BlockLinearSolver._update_rhs_vecs
      LinearSolver._run_apply
      BlockLinearSolver._iter_get_norm
    Solver._mpi_print
    LinearBlockGS._single_iteration
    LinearSolver._run_apply
    BlockLinearSolver._iter_get_norm
```

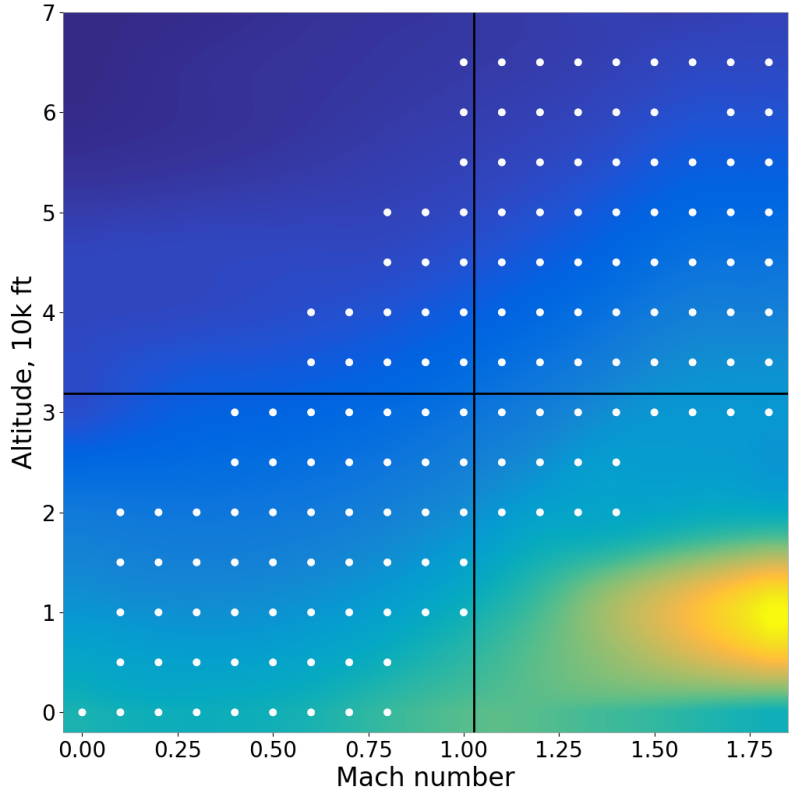
Surrogate modeling

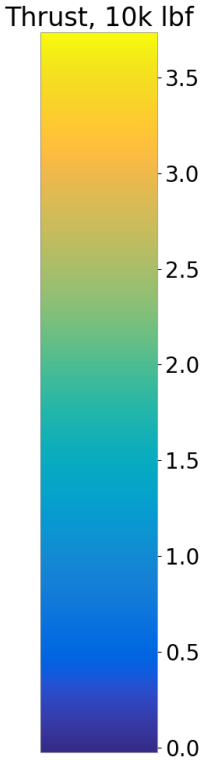
- Built-in methods for structured and unstructured data interpolation
  - Kriging surrogate
  - Nearest neighbor
  - Response surface

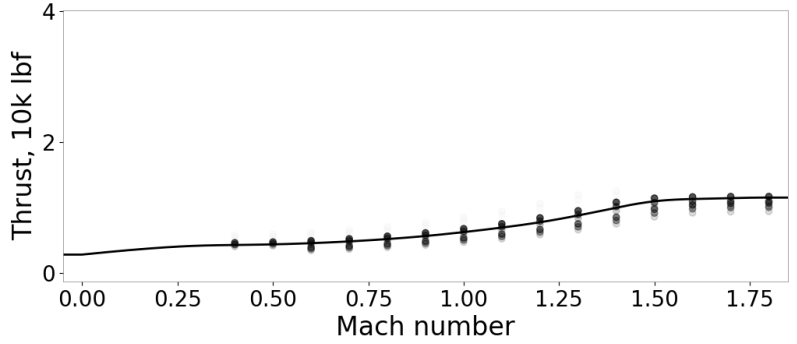
http://openmdao.org/twodocs/versions/latest/features/building_blocks/components/metamodelunstructured_comp.html

Surrogate modeling





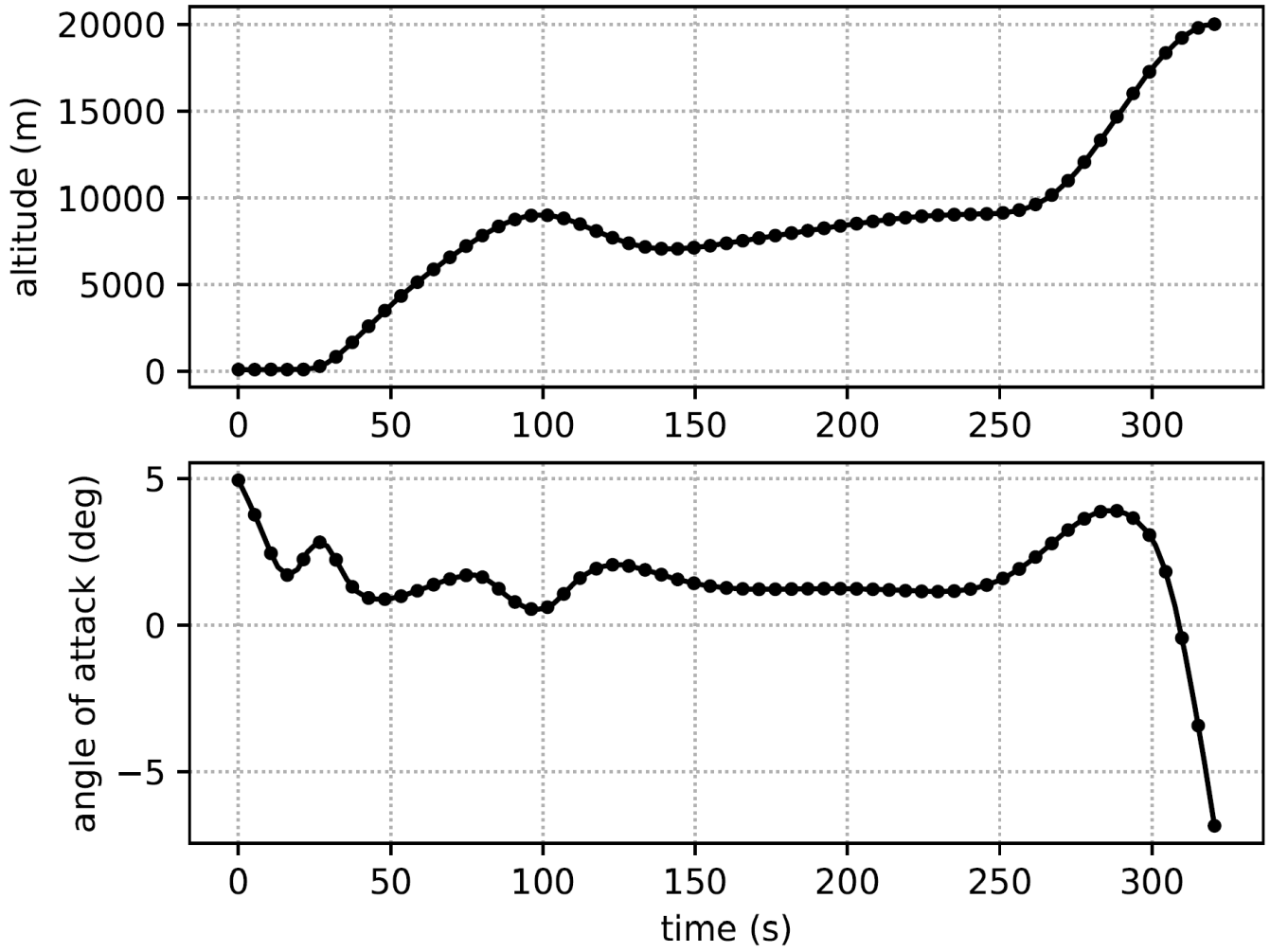




Trajectory optimization: Dymos

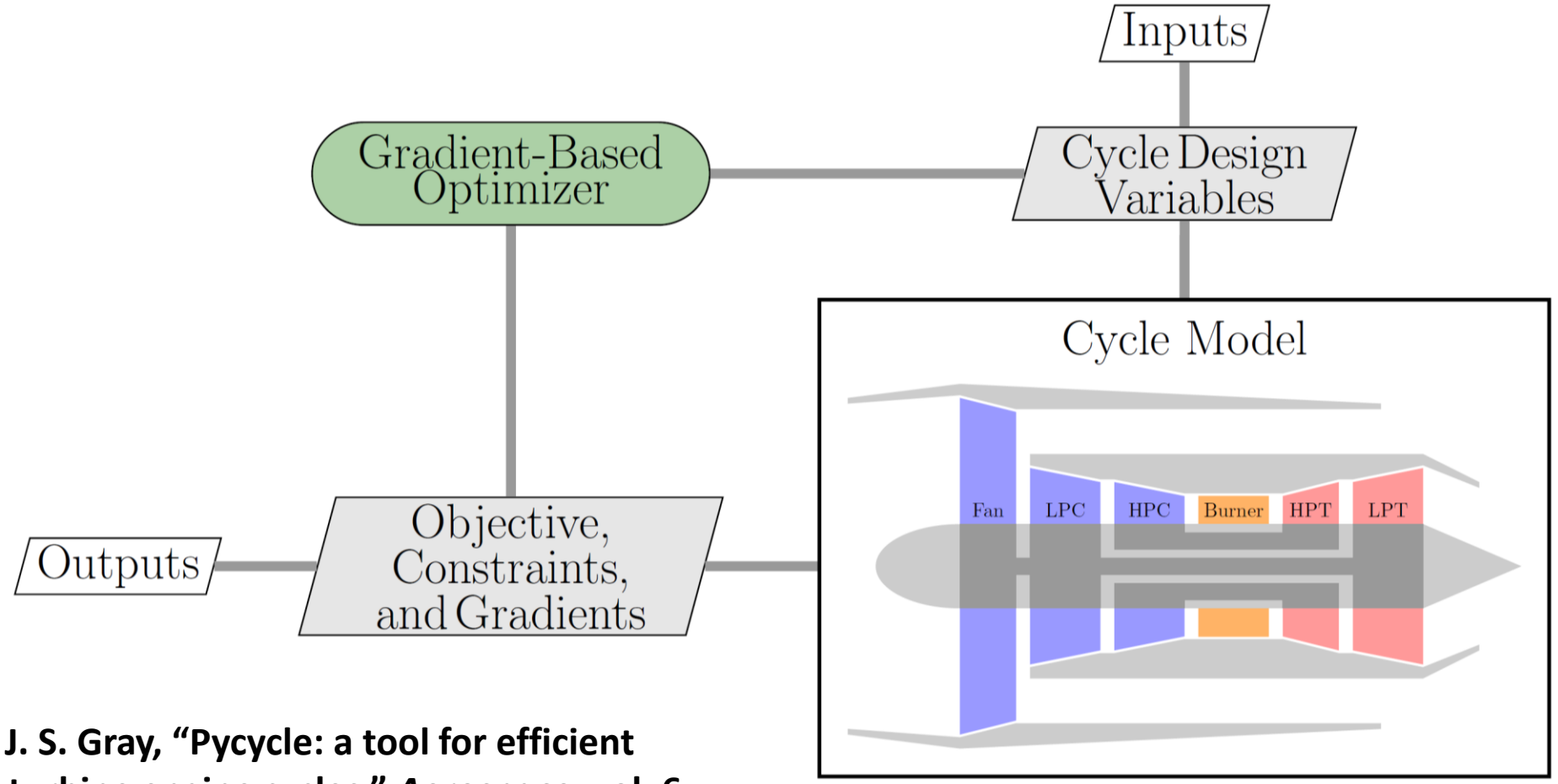
- *Dymos* is a tool built using OpenMDAO that solves optimal control problems involving multidisciplinary system

- Example optimal trajectory for an aircraft climb shown on right



http://github.com/openmdao/dymos

Propulsion optimization: PyCycle



E. S. Hendricks and J. S. Gray, "Pycycle: a tool for efficient optimization of gas turbine engine cycles," *Aerospace*, vol. 6, iss. 87, 2019.

https://github.com/openmdao/pycycle

Recap of today's activities

- OpenMDAO intro and basics
  - Lab 0: Explicit components and connections
- Using solvers with implicit models
  - Lab 1: Comparing gradient-free and gradient-based solvers
- Optimization with and without analytic derivatives
  - Lab 2: Optimizing the thickness distribution of a simple beam
- Wrapping external codes
  - Lab 3: Wrapping external codes as explicit and implicit components
- Advanced topics

Tusen takk!

AEROSPACE ENGINEERING
UNIVERSITY of MICHIGAN

NTNU

Tusen takk!

Subscribe to MDO Lab e-mail updates by sending a blank e-mail to:

engopt-join@mdolab.engin.umich.edu

http://mdolab.engin.umich.edu/

AEROSPACE ENGINEERING
UNIVERSITY of MICHIGAN

NTNU