Contents

1 Aeroshell

Default Values	Quantity	Units	Variable Name
Mass			`Mass`
Area			`Area`
Lift Coefficient			`LiftCoefficient`
Drag Coefficient			`DragCoefficient`

```
from openmdao.api import ExplicitComponent
import numpy as np
class AeroShell(ExplicitComponent):
    """Aeroshell of the pod"""
    <<AeroShell_initialize>>
    <<AeroShell_setup>>
    <<AeroShell_compute>>
    <<AeroShell_compute_partials>>
```

1.1 initialize **INITIALIZE**

```
def initialize(self):
   """Declare options"""

    # Need to convert this to an input at some point
    self.options.declare('Mass',
                         default=1.,
                         types=np.ScalarType,
```

```
                    desc='Mass of the HyperJackets Pod')

    # The following properties need to be made more accurate ASAP.
    # These should not exist like this

    self.options.declare('Area'
                         default=1.
                         types=np.ScalarType,
                         desc='Pod Lift Area')

    self.options.declare('LiftCoefficient'
                         default=1.
                         types=np.ScalarType,
                         desc='Lift Coefficient of Pod')

    self.options.declare('DragCoefficient'
                         default=1.
                         types=np.ScalarType,
                         desc='Drag Coefficient of Pod')
```

1.2 setup

SETUP

```
def setup(self):
   """Declare inputs and outputs"""

    # Inputs
    self.add_input('AirDensity',
                   0.5,
                   desc="Air Density around the aeroshell")
    self.add_input('Velocity',
                   0,
                   desc="Speed of the aeroshell (and of the pod)")

    # Outputs
    self.add_output('Lift', 0.0,
                    units="kg*m/s^2",
                    desc="Lifting force due to the aeroshell")
    self.add_output('Drag', 0.0,
                    units="kg*m/s^2",
                    desc="Drag force due to the aeroshell")
```

```
    # Independence
    self.declare_partials('Lift',
                          'DragCoefficient',
                          dependent=False)
    self.declare_partials('Drag',
                          'LiftCoefficient',
                          dependent=False)
```

1.3 compute COMPUTE

```
def compute(self, inputs, outputs):
    """Compute outputs"""

    # Properties of the tube
    rho = inputs['AirDensity']

    # Aerodynamic Coefficients
    c_l = self.options['LiftCoefficient']
    c_d = self.options['DragCoefficient']

    # Properties of the pod
    area = self.options['Area']
    vel = inputs['Velocity']

    lift = - 0.5*rho*c_l*area*vel*vel
    drag = 0.5*rho*c_d*area*vel*vel

    outputs['Lift'] = lift
    outputs['Drag'] = drag
```

1.4 compute partials COMPUTE_PARTIALS

```
def compute_partials(self, inputs, partials):
    """ Computation of partial derivatives."""

    c_l = self.options["LiftCoefficient"]
    c_d = self.options["DragCoefficient"]
    area = self.options["Area"]
    vel = inputs["Velocity"]
```

```
    rho = inputs["AirDensity"]

    partials['Lift', 'AirDensity'] = -0.5*c_l*area*vel*vel
    partials['Lift', 'Velocity'] = - c_l*area*rho*vel
```

2 Wheels

```
from openmdao.api import ExplicitComponent

<<Wheels_wheelStress>>
class Wheels(ExplicitComponent):
    """ Wheel Material """
    <<Wheels_initialize>>
    <<Wheels_setup>>
    <<Wheels_compute>>
```

2.1 initialize

INITIALIZE

```
def initialize(self):
   """Declare options"""

    # Material Properties
    self.options.declare('Density',
                         default=1.,
                         types=np.ScalarType,
                         desc='Density of the wheel material')
    self.options.declare('PoissonsRatio',
                         default=1.
                         types=np.ScalarType,
                         desc="Poisson's Ratio for the wheel material")
    self.options.declare('FrictionCoefficient',
                         default=1,
                         types=np.ScalarType,
                         desc="Friction Coefficient of the wheel material")
    self.options.declare('YieldCircumferentialStress',
                          default=1,
                          types=np.ScalarType,
                          desc="Max Circumferential Stress")
    self.options.declare('YieldRadialStress',
```

```
                      default=1,
                      types=np.ScalarType,
                      desc="Max Radial Stress")

    # Engineering Properties
    self.options.declare('FactorOfSafety',
                      default=1,
                      types=np.ScalarType,
                      desc="Factor of Safety for the wheels")

    # Wheel Properties
    self.options.declare('InnerRadius',
                         default=1.,
                         types=np.ScalarType,
                         desc="Inner Radius of the wheel")
    self.options.declare('OuterRadius',
                         default=1.,
                         types=np.ScalarType,
                         desc="Outer Radius of the wheel")
    self.options.declare("Multiplicity",
                         default=4.,
                         types=np.ScalarType,
                         desc="Number of wheels used on pod")
```

2.2 setup **SETUP**

```
def setup(self):
    """Declare inputs and outputs"""

    # Inputs
    self.add_input('NormalForce',
                   0.5,
                   desc="Normal Force applied on wheels due to weight of the pod")
    self.add_input('Velocity',
                   0.5,
                   desc="Velocity of the pod")

    # Outputs
    self.add_output('RevolutionsPerMinute',
                    0.5,
```

```
                desc="Revolutions per minute of the wheel")
self.add_output('FrictionForce',
                0.5,
                desc="FrictionForce applied to the wheels of the car")

# Output Stresses experienced by the wheel
self.add_output('MaximumCircumferentialStress',
               0.5,
               desc="Circumferential Stress experienced due to rotation")
self.add_output('MaximumRadialStress',
               0.5,
               desc="Radial Stress experienced due to rotation")

# Independence
self.declare_partials('CircumferentialStress',
                      'NormalForce',
                      dependent=False)
self.declare_partials('RadialStress',
                      'NormalForce',
                      dependent=False)
```

2.3 compute COMPUTE

```
def compute(self, inputs, outputs):
    """Compute outputs"""

    # Material Properties of the wheel
    density = self.options["Density"]
    m = self.options["PoissonsRatio"]
    c_f = self.options["FrictionCoefficient"]
    circumferential_max = self.options["YieldCircumferentialStress"]
    radial_max = self.options["YieldRadialStress"]

    # Wheel Properties
    r1 = self.options["InnerRadius"]
    r2 = self.options["OuterRadius"]
    multiplicity = self.options["Multiplicity"]

    # Pod Properties
    vel = inputs["Velocity"]
```

```
normal_force = inputs["NormalForce"]

# Engineering
# Derived Properties
omega = vel/r2


# Circumferential & Radial Stresses
if r1 is 0:
    # We're dealing with a solid disc
    # Both stress are max at r = 0
    # Circumferential Stress & Radial Stress
    c_stress, r_stress = wheelRotationalStress(radius = 0,
                                               innerRadius = r1,
                                               outerRadius = r2,
                                               omega = omega,
                                               poissonsRatio = m,
                                               density = density):
else:
    # We're dealing with a hollow disc
    # Both stress are max at r = (r1 * r2) ** (0.5)
    # Circumferential Stress & Radial Stress
    c_stress, r_stress = wheelRotationalStress(radius = (r1*r2)**(0.5),
                                               innerRadius = r1,
                                               outerRadius = r2,
                                               omega = omega,
                                               poissonsRatio = m,
                                               density = density):

if c_stress > circumferential_max:
    failure = "Due to stress above yield circumferential stress"
    raise WheelFailure(failure)
elif c_stress >  circumferential_max / factor_of_safety:
    failure = "Circumferential stress is past allowable "
    raise WheelFailure(failure)

else:
    outputs["CircumferentialStress"] = c_stress

if r_stress > radial_max:
```

```
        raise WheelFailure("Your wheels have ripped apart due to radial stress")
    elif r_stress >  radial_max / factor_of_safety:
        raise WheelFailure("Your radial stress is past the allowable stress")
    else:
        outputs["RadialStress"] = r_stress

    # RevolutionsPerMinute
    # FrictionForce
```

2.3.1 Circumferential Stress

```
def wheelRotationalStress(radius = 0,          # Desired Radius
                          innerRadius = 0,     # Inner Radius of the wheel
                          outerRadius = 1,     # Outer Radius of the wheel
                          omega = 1,           # Rotational Velocity of the wheel
                          poissonsRatio = 1,   # Poisson's Ratio. Denoted by 1/m
                          density = 1):        # Density of the material

    if innerRadius is 0:
        # We're dealing with a solid disc
        C_1 = (3 + poissonsRatio)*(1/4)
        C_1 *= (density*(omega**2)*(outerRadius**2))
        C_2 = 0
    else:
        # We're dealing with a hollow disc
        C_1 = (3 + poissonsRatio)*(1/4)
        C_1 *= (density*(omega**2)*(innerRadius**2 + outerRadius**2))
        C_2 = (3 + poissonsRatio)*(1/8)
        C_2 *= (density*(omega**2)*(innerRadius**2)*(outerRadius**2))

    sigma_radial = C_1/2 + C_2/(radius**2)
    sigma_radial -= (3 + poissonsRatio)*(1/8)*(density*(omega**2)*(radius**2))
    sigma_circum = C_1/2 - C_2/(radius**2)
    sigma_circum -= (1 + 3*poissonsRatio)*(1/8)*(density*(omega**2)*(radius**2))

    return sigma_radial, sigma_circum
```