# hypervehicle

*Release 0.1.0*

**Kieran Mackle, Ingo Jahn**

**Jan 02, 2024**

# SHOWCASE



HyperVehicle provides a versatile tool to rapidly generate parametric vehicle geometries following a component build-up approach. Take a look at the *Getting Started* guide to get set up, then have a look at the *Examples*.

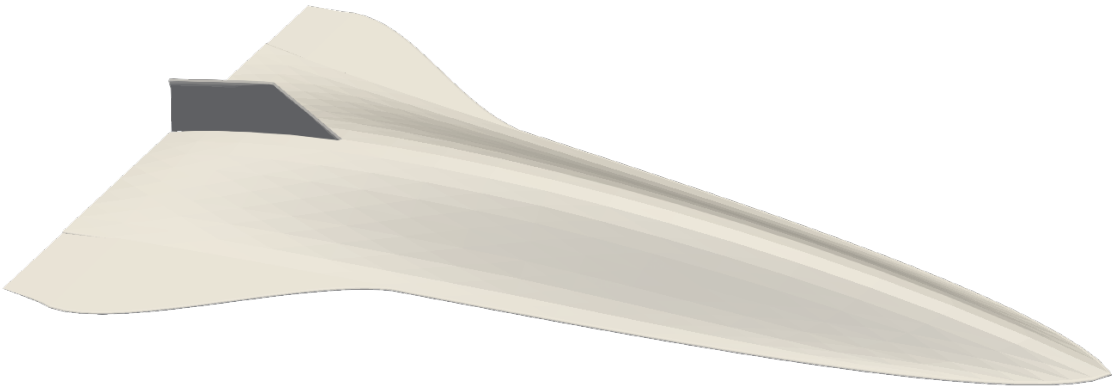
This page is a showcase of vehicle geometries possible using HyperVehicle. Note that all of these vehicles can be accessed via the `hypervehicle.hangar` namespace. For example, to generate the X-43A geometry shown below, the following code can be used.

```
from hypervehicle.hangar import ParametricX43

parametric_generator = ParametricX43()
x43 = parametric_generator.create_instance()
x43.generate()
x43.to_stl()
```

CHAPTER

# ONE

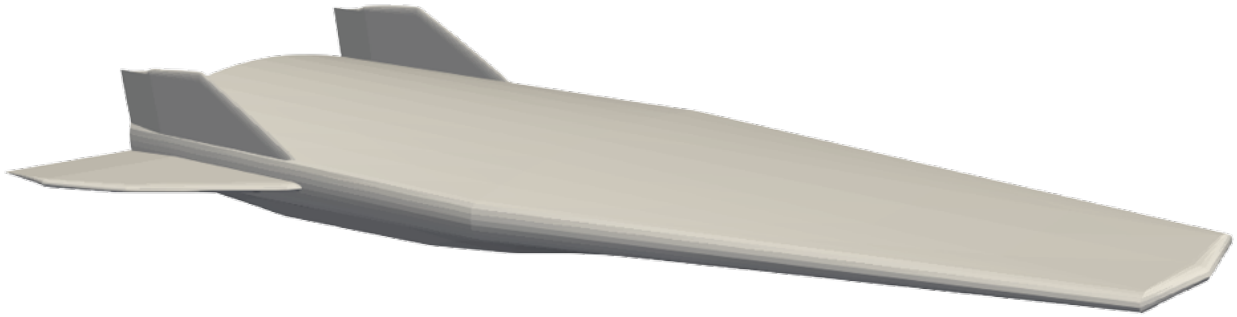
# GENERIC HYPERSONIC WAVERIDER



CHAPTER

# TWO

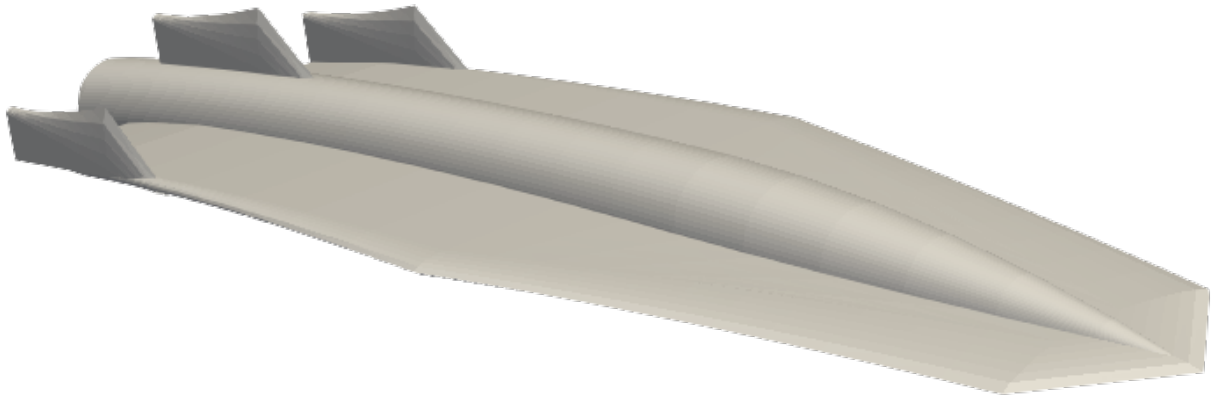
## X-43A



CHAPTER

# THREE

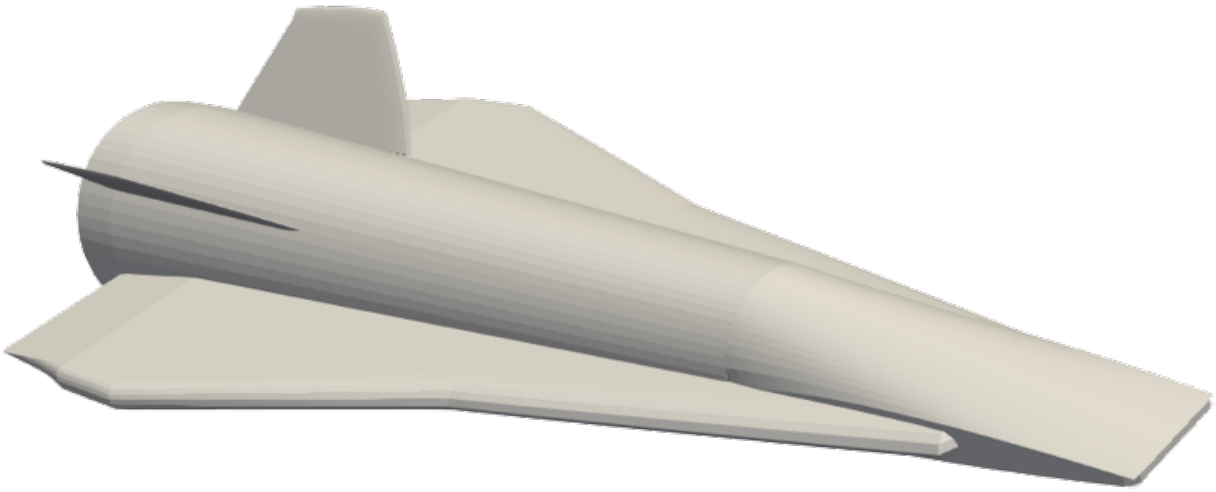
## HIFIRE4



CHAPTER

# FOUR

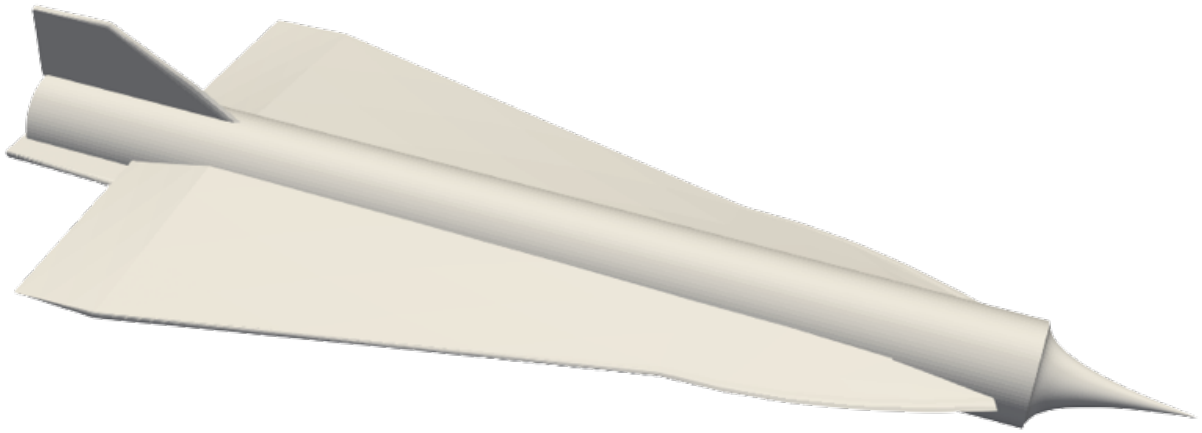
# HIFIRE8



CHAPTER

# FIVE

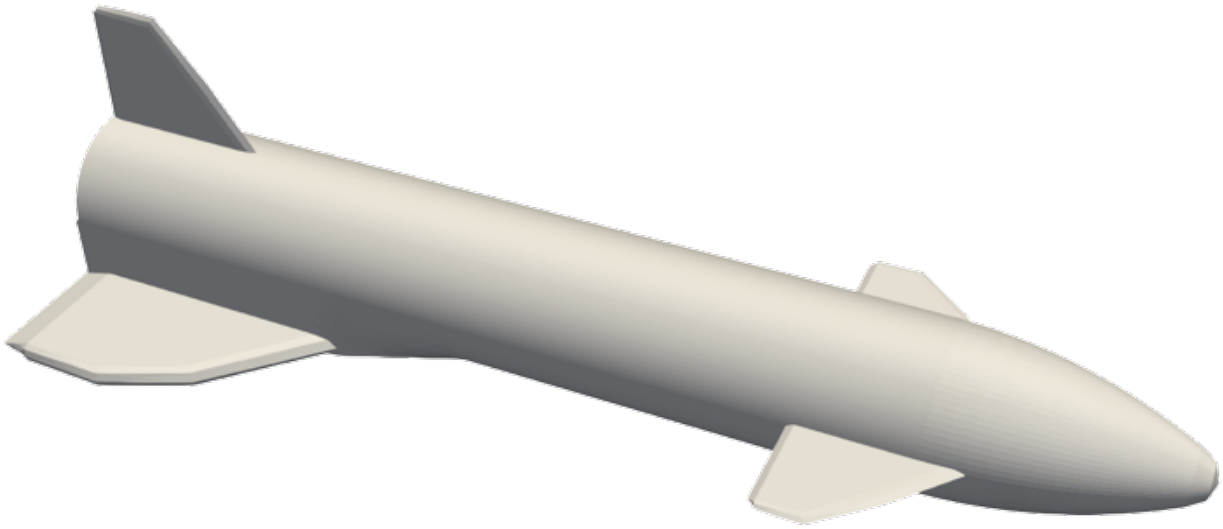
# LOCKHEED D-21



CHAPTER

SIX

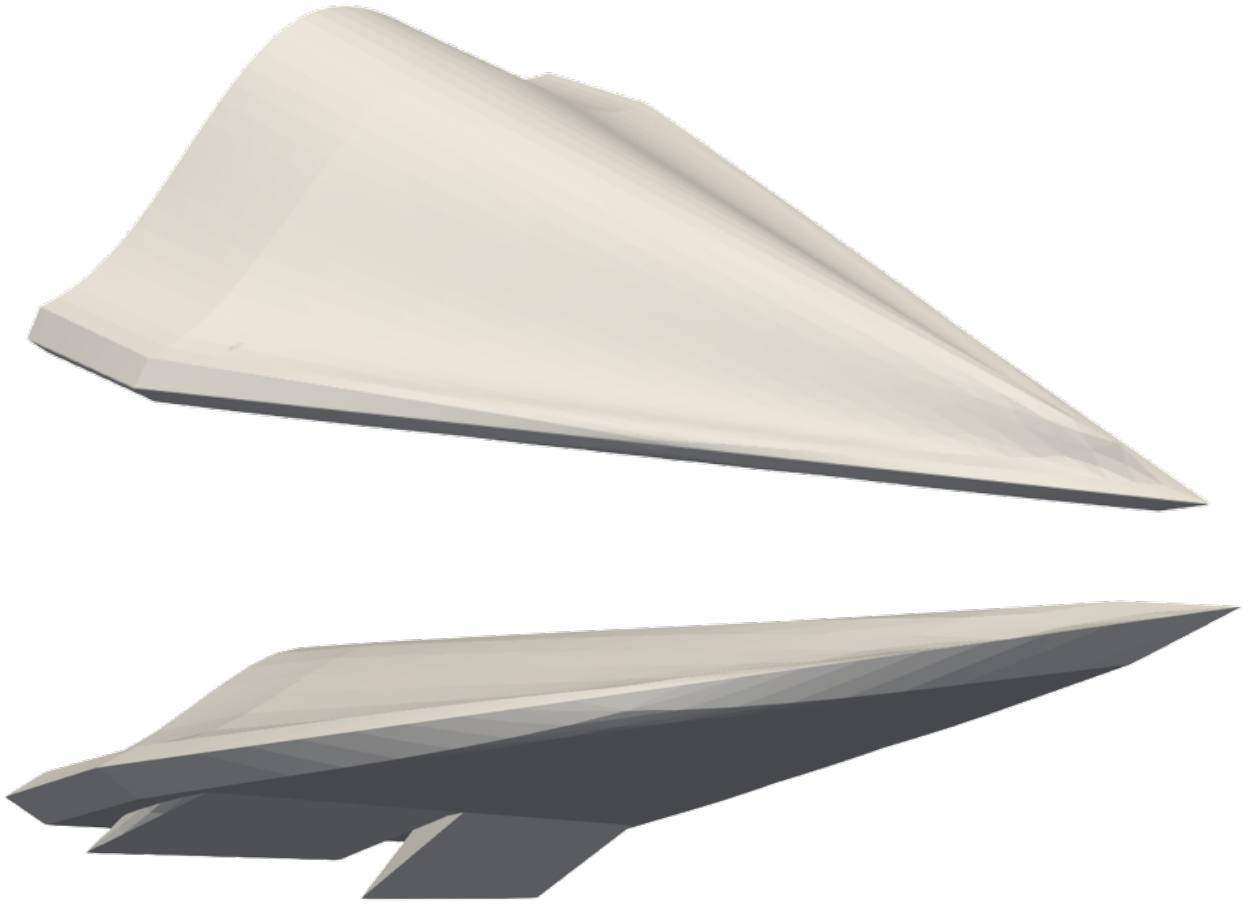
# DLR REFEX



CHAPTER

SEVEN

# HYPERSONIC TECHNOLOGY VEHICLE



CHAPTER

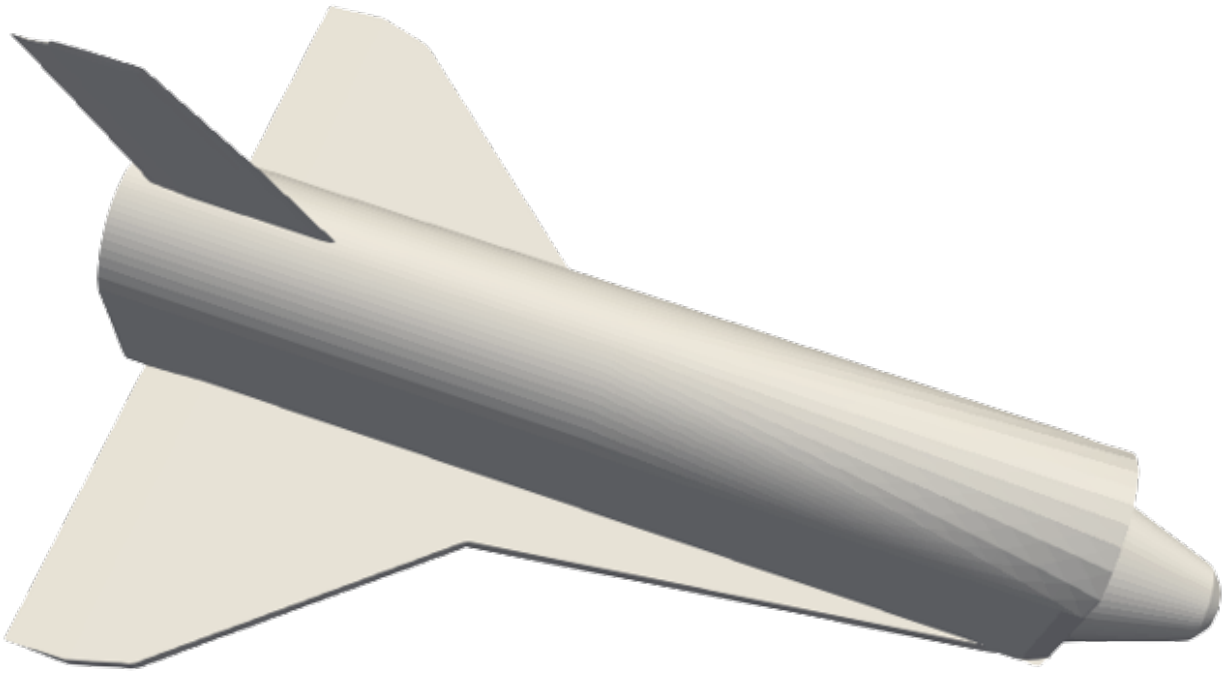
# EIGHT

## SPACEX FALCON9

CHAPTER

NINE

# SPACE SHUTTLE



CHAPTER

TEN

# PUBLICATIONS OF HYPERVEHICLE

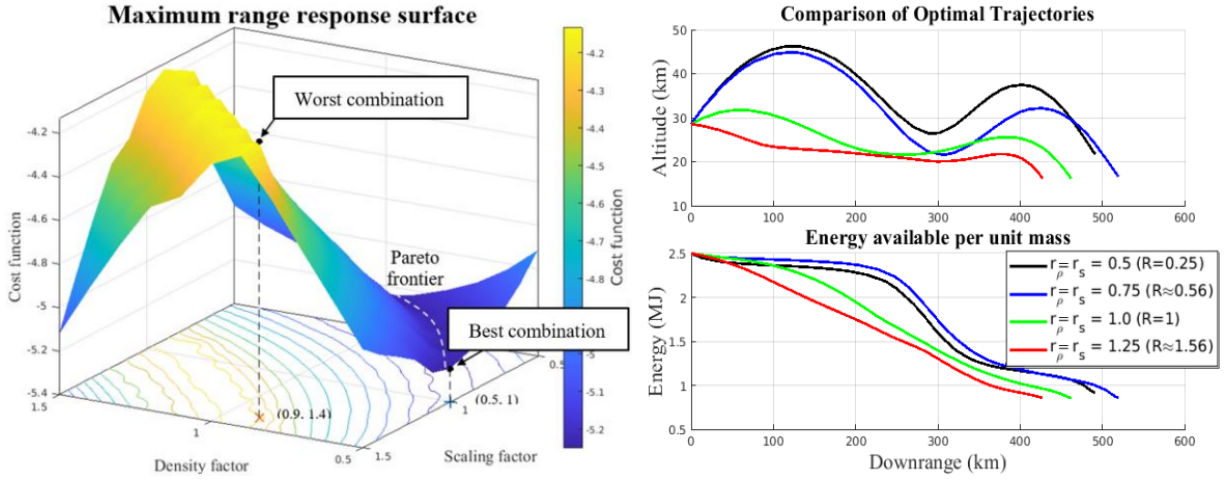
Listed here are some publications made with the use of HyperVehicle.

## 10.1 2022 Publications

### 10.1.1 Co-Designing Hypersonic Vehicle Geometry and Trajectory

Published in: 23AFMC

To address the hypersonic vehicle design challenge of highly integrated subsystems and competing design requirements, a co-design approach is essential. This approach involves the design and optimisation of various system components in parallel across an entire mission trajectory, to obtain a vehicle configuration more optimal than any single trade study can produce. In this paper, a parameterised mock-up of the X-43A is used as a baseline vehicle in a developed co-design framework. By co-designing the vehicle geometry, parameterised as the vehicle's mean density and scale for a maximum range glide problem, a 15% increase in performance was achieved. This increase in performance was attributed to the optimisation of the vehicle's wing loading and lift-to-drag ratio to suit a skip-glide trajectory. This paper therefore demonstrates the capability of a co-design methodology to uncover the central performance-driving design features under the influence of subsystem interactions. This adds value to the preliminary design stage, highlighting underlying mission-critical design features.

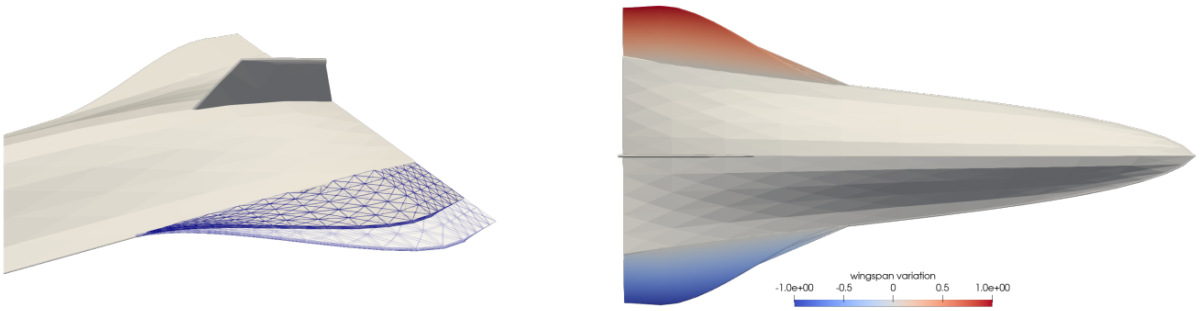


## 10.2 2024 Publications

### 10.2.1 Efficient and Flexible Methodology for the Aerodynamic Shape Optimization of Hypersonic Vehicle Concepts in a High-Dimensional Design Space

*To be presented at AIAA SciTech Forum and Exposition 2024*

Aerodynamic shape of optimization of hypersonic vehicles parameterized by a large number of design variables (e.g. more than 10) is challenging due to the high computational cost of CFD evaluations. Gradient-based optimization methods are commonly used for this reason, along with the adjoint method applied to the governing equations of the flow to provide the optimizer with a search direction. Downsides of this approach include that it requires specialized CFD solvers, highly converged CFD simulations, and it can still be costly when optimizing against multiple performance objectives. Exploration studies in a high-dimensional design space are particularly beneficial during the conceptual design phases, and thus an optimization approach that is easy to implement, flexible, and quick to run is essential. In this paper, we develop an alternative approach to obtain the search direction for gradient-based aerodynamic shape optimization. The proposed approach is efficient, scalable, and CFD solver-agnostic. An approximate Jacobian is obtained by applying lower order aerodynamic models (such as Piston theory or Van Dyke's second order theory) locally, thus allowing pressure sensitivities to design parameters to be calculated without the need for further costly CFD simulations. The accuracy of this approach is demonstrated by estimating pressure sensitivities for a canonical shape, and the results are verified by comparison to finite difference of CFD solutions. We then demonstrate the approach as a suitable means for shape optimization by optimizing a generic hypersonic waverider for maximum Lift-to-Drag ratio (/), whilst also respecting an internal volume constraint. Using an 8 CPU workstation, the optimized configuration (parameterized by 16 design variables) is obtained in little over 4 hours, with a 33% increase in /.




(a) Geometry surface mesh variation due to a perturbation in wingspan parameter (magnitude exaggerated for illustrative purposes). Vertex-parameter sensitivities are calculated for each individual vertex of the geometry mesh.

(b) Visualization of resultant parameter sensitivity, illustrating that the wingspan parameter extends the wing surface mesh in the spanwise direction.

**Fig. 3 Visualization of parametric wingspan geometry sensitivity as generated by `HyperVehicle`.**

### 10.2.2 Developing a Co-Design Framework for Hypersonic Vehicle Aerodynamics and Trajectory

*To be presented at AIAA SciTech Forum and Exposition 2024*

The extreme conditions at which hypersonic vehicles are required to operate necessitate a novel method of design, capable of producing vehicles that can perform across their entire mission trajectory. Traditional methods of design using trade studies of design parameters are incapable of capturing complex and non-linear subsystem interactions, which dominate hypersonic flight vehicles. Further challenges arise from the many competing design requirements, including packaging constraints, aerodynamic requirements, and flight path objectives. While traditional multi-objective design optimization methods attempt to address these challenges, they fail to account for the entirety of a mission's flight trajectory, and how the design parameters impact the objective at a system-level. This paper presents a novel

and computationally tractable approach to co-design a vehicle's geometry and flight trajectory simultaneously, in order to obtain an optimal vehicle shape and flight path for a specified mission objective. That is, the vehicle shape is and trajectory are optimized in the same phase to obtain a the highest performing system solution in regards mission-level objectives. Importantly, the computational cost of the proposed method scales favourably with the number of vehicle design parameters, and is designed to reduce the number of computational fluid dynamic simulations. The approach is demonstrated by optimizing a hypersonic glider (waverider) parametrically defined using 16 variables to attain maximum range while also obeying an internal volume constraint. The optimal vehicle configuration obtained through the proposed co-designed framework exhibits an 11.6 % improvement over the nominal configuration.

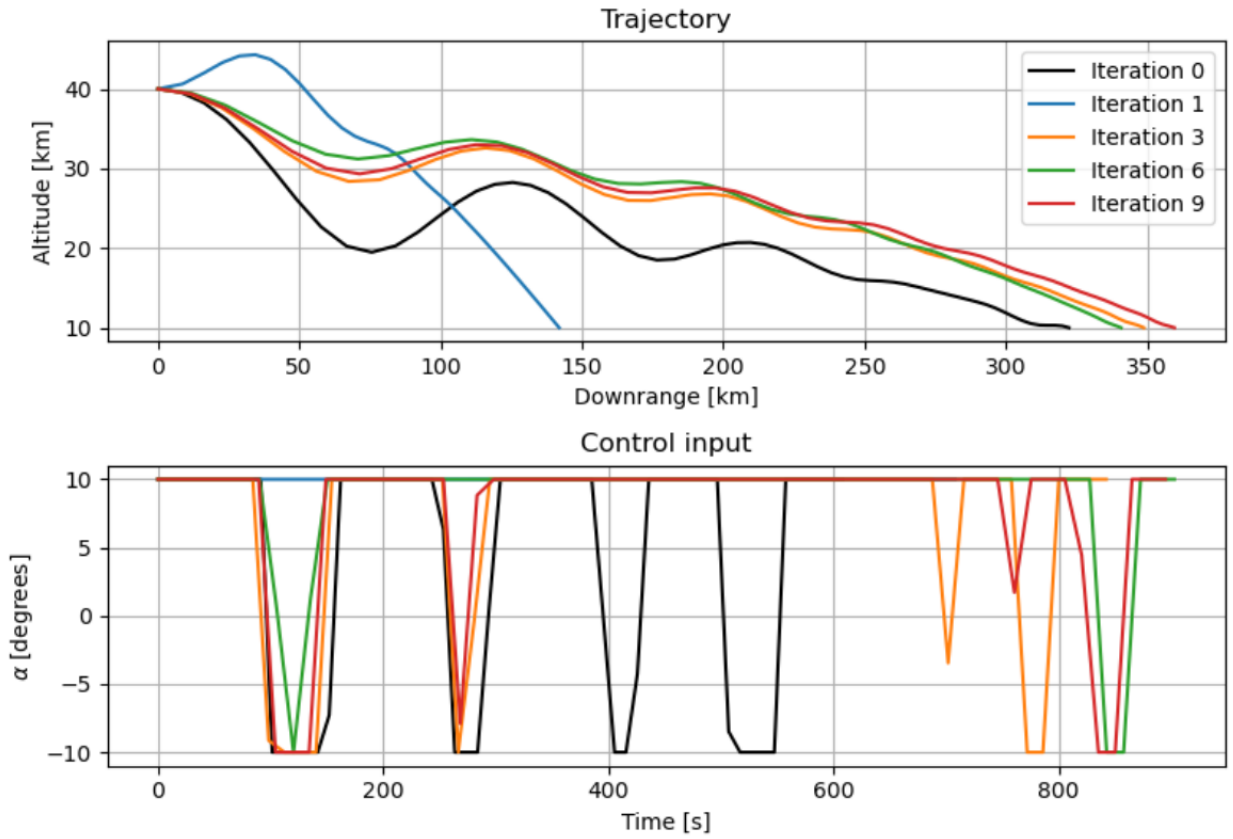



**Fig. 9 Evolution of the optimal trajectory during the co-design process.**

CHAPTER

ELEVEN

# GETTING STARTED WITH HYPERVEHICLE

## 11.1 Installation

### 11.1.1 Installation from PyPI

```
pip install hypervehicle
```

### 11.1.2 Installation from source

To install `hypervehicle` from source, use the command below.

```
pip install git+https://github.com/kieran-mackle/hypervehicle
```

### 11.1.3 Testing the installation

To check that everything has been installed properly and HyperVehicle is ready to go, run the command below.

```
python3 -m pytest tests/
```

## 11.2 Optional Dependencies

### 11.2.1 PyMESH

To access more advanced features of HyperVehicle, such as compeonent mesh merging and cleaning, PyMesh is required. You do not need to install this to use HyperVehicle, but it can come in handy, especially when extracting sensitivities for multi-component geometries.

CHAPTER

# TWELVE

# OVERVIEW OF HYPERVEHICLE

The HyperVehicle package employs a component build up approach. Using the core component types, a wide range of geometries can be constructed. While there is an upfront cost associated with constructing a good parametric model, the generalised nature of HyperVehicle allows you to programmatically define relationships between geometric components.

## 12.1 The HyperVehicle Workflow

Before diving into the component definitions, this section provides an overview of the HyperVehicle workflow, and how vehicle geometries are created.

A geometry is constructed by combining a collection of *components* together. Each component type of `hypervehicle` inherits from the *`Component`* class.

Once the components of a geometry are defined, they can be added to an instance of the *`Vehicle`* class, which acts as a container for multiple *`Component`*s. Components can be added to a `Vehicle` object using the *`Vehicle.add_component()`* method. The *`Vehicle`* object offers many useful features and methods, allowing you to name a vehicle, name individual components, apply transformations, estimate the vehicle's inertial properties, and more.

CHAPTER

# THIRTEEN

# HYPERVEHICLE COMPONENTS

As described in the *HyperVehicle overview*, a component-based approach is used to construct geometries. We classify two types of components: generic components and vehicle components. The former refers to components defined by operations you may already be familiar with from a CAD context, while the latter refers to components made specifically to help define hypersonic vehicle geometries.

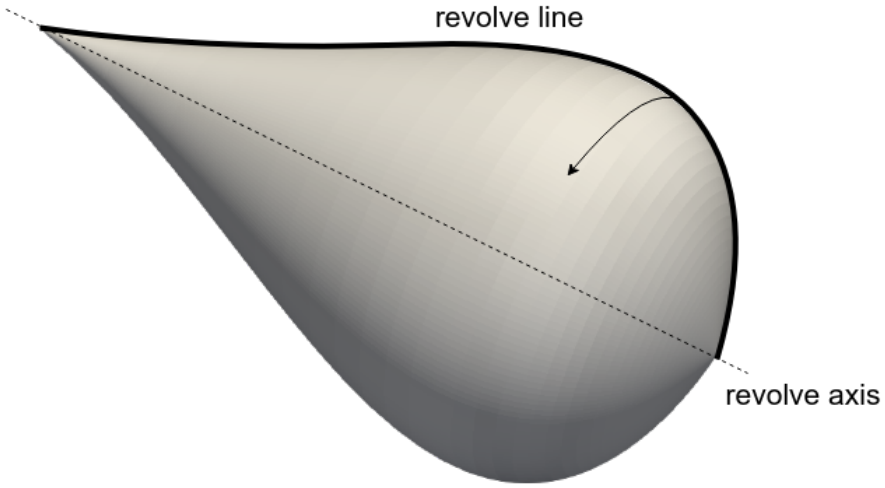
## 13.1 Generic Component Types

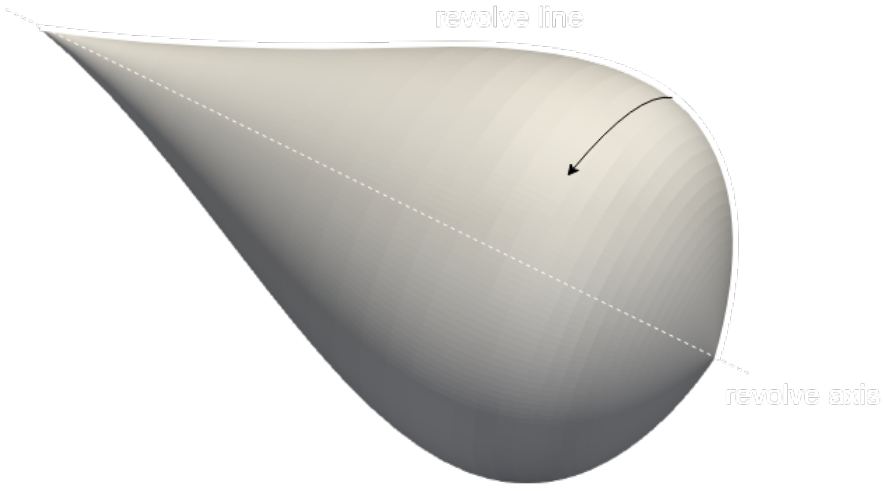
The following page documents the generic component types of HyperVehicle. These are not vehicle-specific. Note that an example script is provided for each component type in the HyperVehicle examples directory.

### 13.1.1 Revolved Component

Example: `revolved.py`

Revolved geometries can be constructed using the *`RevolvedComponent`*. This component is defined by the line to be revolved, as shown in the example below.

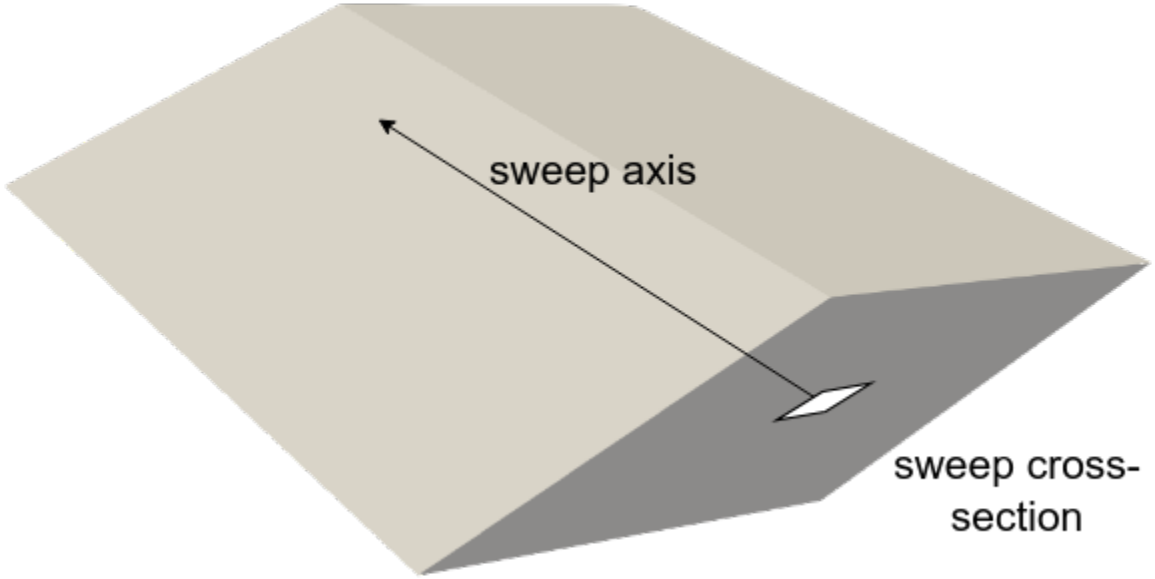

revolve line
revolve axis


revolve line
revolve axis

### 13.1.2 Swept Component

Example: `swept.py`

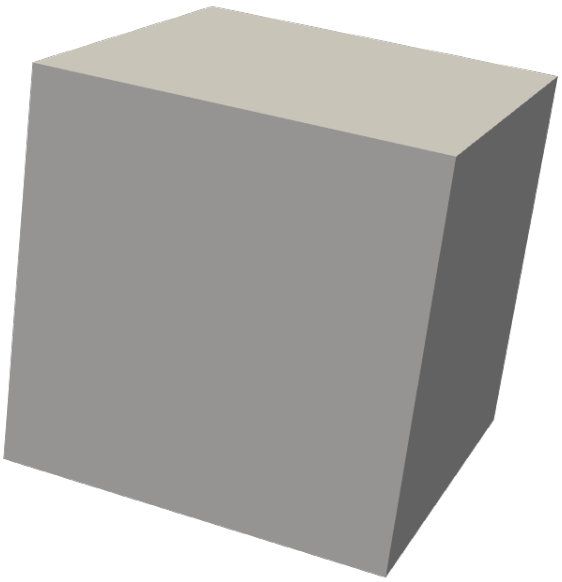
Swept geometries can be constructed using a *`SweptComponent`* type. This component requires a series of cross-sectional patches to be defined. These cross-sections then get swept through the sweep axis, to form a geometry such as that shown below.




### 13.1.3 Cube Component

Example: `cube.py`

A cube can be constructed via the *`Cube`* class. It only requires the cube side length to be provided. The cente of the cube can also be specified.



### 13.1.4 Sphere Component

Example: `sphere.py`

A sphere can be constructed using the *`Sphere`* class. Like the cube, it only requires a radius to be defined. The cente of the sphere can also be specified.

### 13.1.5 Composite Component

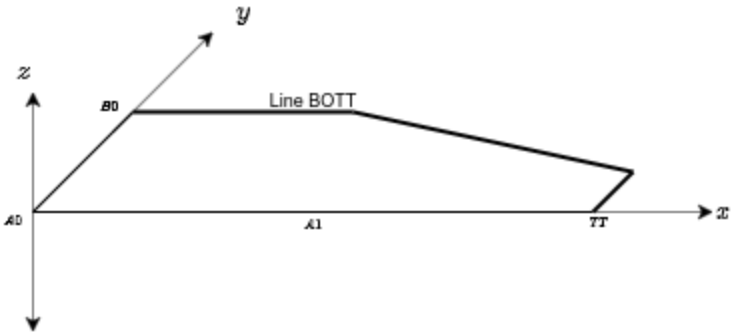
In some advanced instances, you may wish to combine the patches of various components into a single component. Or, you may have defined your own patch types. For this purpose, the *CompositeComponent* class is available. This class allows a user to stack individual *Component* objects to form a composite, which can then be added to the *Vehicle*, for example.

## 13.2 Vehicle Specific Component Types

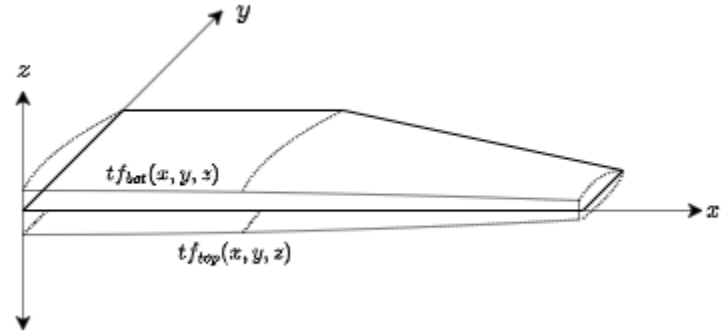
The following section describes the component types defined for helping specifically with vehicle geometries.

### 13.2.1 Wing Components

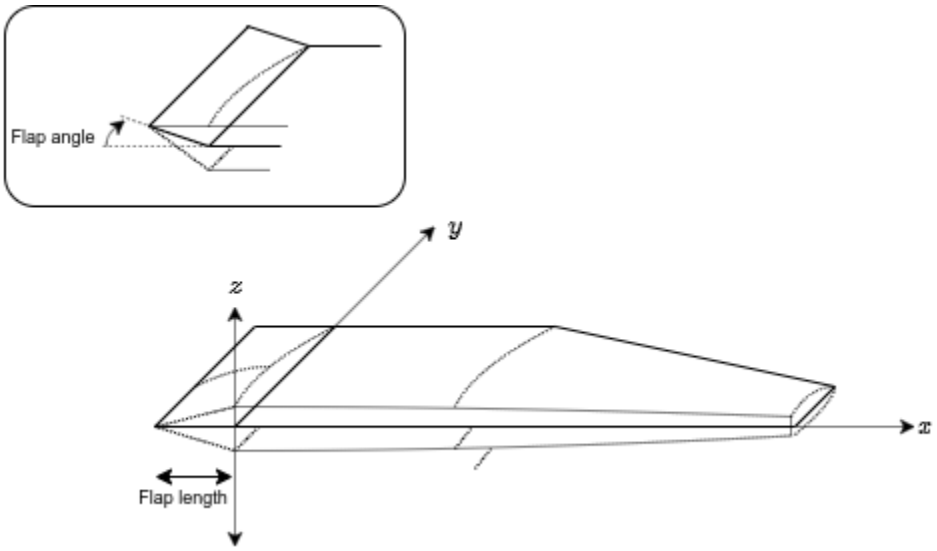
For constructing wings, the *Wing* component function is available. A wing component is constructed by first defining the planform according to the points defined in the schematic below.




Next, thickness is added to the wing using user-defined thickness functions. These function can be as simple as providing a constant thickness, or as complex as providing 3-dimensionally varying thickness.

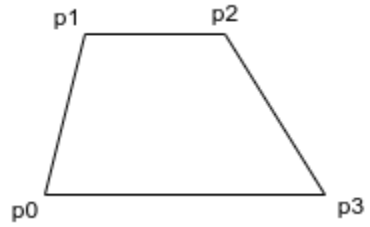



A trailing-edge flap can easily be added to a wing component by defining the flap length and flap type. The flap angle can also be provided to deflect the flap.

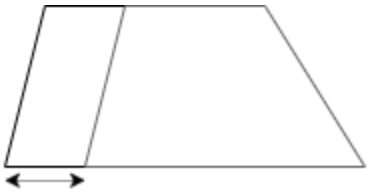


### 13.2.2 Fin Components

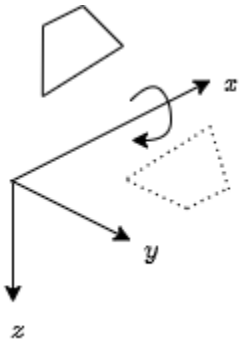
Another helper component is the *Fin*. Fin components are very similar to wing components, but offer a few convenient options to assist in positioning. As with a wing, a fin is first defined by its planform according to the points shown below.



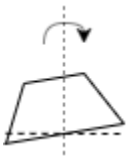

A rudder can also be added to the trailing-edge of a fin, producing something like that shown below.



To assist in fin positioning, the fin angle argument can be used to specify the angle of the fin, as rotated about the vehicles longitudinal (x) axis.




The pivot angle of the fin can also be controlled using the pivot angle argument. The pivot point can also be specified.



CHAPTER

FOURTEEN

# BUILDING A VEHICLE WITH *HYPERVEHICLE*

Docs coming soon! The following will be addressed:

- creating a vehicle instance
- adding components to the vehicle
- applying component transformations
- applying vehicle transformations

Plus other information about the *Vehicle* object.

CHAPTER

FIFTEEN

# GENERATING GEOMETRY SENSITIVITIES

HyperVehicle also features the generation of geometry sensitivities via method of finite differences. This page outlines how to use this capability.

## 15.1 Nomenclature

| Symbol | Description |
|---|---|
| $\underline{\theta}$ | The set of design parameters defining a geometry |
| $\mathcal{G}$ | The geometry mesh |

## 15.2 Description of Implementation

1. A HyperVehicle parameterised geometry `Generator` is defined, along with the parameters to be varied.
2. Geometry meshes for the nominal geometry $\mathcal{G}_{\text{nominal}}$ are generated.
3. For each parameter $\theta_i$ to be varied:
   - The perturbed parameter value is defined as $\theta_i' = \theta_i + \delta\theta_i$.
   - Geometry meshes for the parameter set including $\theta_i'$ are generated, yielding $\mathcal{G}_i'$.
   - The sensitivity of each vertex of $\mathcal{G}$ to the parmater perturbation $\theta_i'$ is calculated by comparing the perturbed geometry $\mathcal{G}_i'$ to the nominal geometry $\mathcal{G}_{\text{nominal}}$. That is, $\delta\mathcal{G}_i = \mathcal{G}_i' - \mathcal{G}_{\text{nominal}}$.
   - Finally, the sensitivity is approximated via:

$$\frac{\partial \mathcal{G}}{\partial \theta} \approx \frac{\delta \mathcal{G}}{\delta \theta}$$

## 15.3 Working with Mutliple-Component Geometries

### 15.3.1 Merging components with PyMesh

If you have PyMesh installed, simply pass `merge=True` when calling *`Vehicle.to_stl()`*. This will merge all components of your Vehicle into a single file.

### 15.3.2 Cart3D intersected Components

Cart3D is a component based solver, meaning oftentimes, many individual components are stacked to form the final geometry. This final geometry is a single body, made up of many combined components. This body has a different mesh to the individual component STL meshes.

When working with multiple components, we need to map the component sensitivities onto the final combined mesh. The utility function `append_sensitivities_to_tri` contained within the `hypervehicle.utilities` module provides this capability. See the code below, which first finds the sensitivity files using `glob`, then runs `append_sensitivities_to_tri`. The Cart3D file, `Components.i.tri` will be overwritten, with the sensitivity data appended. A CSV file called `all_components_sensitivity.csv` will also be written, containing the sensitivity information as point data. This file is compatible with PySAGAS.

```
import glob
from hypervehicle.utilities import append_sensitivities_to_tri

sens_files = glob.glob("*sensitivity*")
append_sensitivities_to_tri(sens_files)
```

You can view this data of the `Components.i.tri` file in ParaView, by first converting the file into a Tecplot format using the Cart3D utility function `trix`:

```
trix Components.i.tri -T
```

## 15.4 Example

Interested in seeing this theory in application? Check out the *parameter sensitivities* tutorial.

CHAPTER

SIXTEEN

# HYPERVEHICLE EXAMPLES

Take a look at the examples below to see how HyperVehicle can be used to generate 3D geometries.

## 16.1 Sharp Wedge Tutorial

This page provides a simple example to get started with HyperVehicle. It will cover how to construct the sharp wedge shown below.

**See also:**

The geometry developed in this example is also used in the *sensitivity tutorial*.

### 16.1.1 Parameters

Its a good idea to start a new geometry with an idea of the parameters which will define it. For a simple geometry such as this, the wingspan, chord length and thickness are the obvious choices.

```
# Define geometry parameters
wingspan = 1
chord = 1
thickness = 0.1
```

### 16.1.2 Swept Component

Since this geometry has a uniform cross-section, using a swept component is ideal.

```
# Create vehicle object
wedge = Vehicle()
wedge.configure(name="Wedge", verbosity=1)

# Define wedge cross-section points
#                   ^ +y
#                   |
#            W    _ - _   N        ___
#            _ -    |   - _         |
#  +x <--- < -------------- >       | thickness
#            - _    |   _ -         |
```

(continues on next page)

(continued from previous page)

```
#            S   -  _ -     E      ___
#
#          |-----------------|
#                wingspan

NW = Vector3(x=0, y=0.5 * self.thicknes)
NE = Vector3(
    x=-0.5 * self.chord,
    y=0,
)
SE = Vector3(x=0, y=-0.5 * self.thicknes)
SW = Vector3(x=0.5 * self.chord, y=0)

# Define patches forming wedge
sections = []
for i in [-1, 1]:
    z_loc = 0.5 * i * self.wingspan
    axial_shift = Vector3(x=0, y=0, z=z_loc)

    N = Line(p0=NW + axial_shift, p1=NE + axial_shift)
    S = Line(p0=SW + axial_shift, p1=SE + axial_shift)
    E = Line(p0=SE + axial_shift, p1=NE + axial_shift)
    W = Line(p0=SW + axial_shift, p1=NW + axial_shift)

    patch = CoonsPatch(north=N, south=S, east=E, west=W)
    sections.append(patch)

fuselage = Fuselage(
    cross_sections=sections,
    sweep_axis="z",
    stl_resolution=10,
)
wedge.add_component(fuselage)

# Generate
wedge.generate()
```

## 16.2 X-43A Demonstrator Tutorial

This page covers how to build a parametric model resembling the X-43A flight demonstrator.

### 16.2.1 Package Imports

Begin by importing all the required parackages and modules.

```
import numpy as np
from hypervehicle import Vehicle
from hypervehicle.components import Wing, Fin, common
from hypervehicle.geometry import Vector3, Bezier, Line, Polyline
```

### 16.2.2 Geometric Parameters

Next, let's define some parameters which will be used to construct the components of the vehicle. These can be hard coded (as below) or treated as variables and loaded in dynamically.

```
# Geometric parameters
body_length = 3.7  # From body rear to nose
body_width = 1  # Body width
body_angle = 0  # Body slant angle (+ve nose down) (degrees)

wing_span = 0.4  # Span of each wing
wing_length = 1.00  # Wing length
wing_TE_back = 0.25
flap_length = 0.3  # Flap length

engine_h = 0.1  # Height of engine section

fin_height = 0.3  # Height of tail fin
fin_length = 1.0  # Length of fin
rudder_length = 0.2  # Length of tail fin rudder

# Configuration parameters
flap_angle = 0  # Elevon angle (+ve up) (degrees)
rudder_angle = 0  # Rudder angle (+ve to +y) (degrees)
```

### 16.2.3 Vehicle Construction

Now we can begin constructing the vehicle.

#### Instantiation

First, create an instance of `Vehicle`, the *hypervehicle* component aggregation class. We can add each component to this vehicle instance, "stacking" them as we go.

```
# Create Vehicle instance
x43 = Vehicle()
x43.configure(name="X-43A", verbosity=1)
```

### Body Geometry

The body of the vehicle is created from a **wing** component type.

```
# WING 1
#     B0------------------------ B1
#     |                            ---___
#     |                                  ---__ B2
#     |                                        |
#     A0-------------------------A1---------------TT

# Define nominal parameters
L_nom = 3.7

L = body_length
beta = np.deg2rad(body_angle)
LE_height = 0.05 * L / L_nom
TE_height = 0.2 * L / L_nom

A0 = Vector3(x=0, y=0)
A1 = Vector3(x=1.8 * L / L_nom, y=0)
TT = Vector3(x=L, y=0)
B0 = Vector3(x=0, y=body_width / 2)
B1 = Vector3(x=A1.x, y=B0.y)
B2 = Vector3(x=TT.x, y=0.4 * B0.y)

B0B1 = Line(p0=B0, p1=B1)
B1B2 = Line(p0=B1, p1=B2)
B2TT = Line(p0=B2, p1=TT)

Line_B0TT = Polyline([B0B1, B1B2, B2TT])

# Top rounding Bezier points
rounding_thickness = 0.025
p1 = Vector3(x=0, y=0, z=rounding_thickness)
p2 = Vector3(x=0, y=0.9 * B0.y, z=p1.z)
p3 = Vector3(x=0, y=B0.y, z=0)
rounding_bez = Bezier([p1, p2, p3])


def get_local_y(x):
    if x < B1.x:
        local_y = B0.y
    else:
        local_y = (x - B2.x) * (B2.y - B1.y) / (B2.x - B1.x) + B2.y
    return local_y


def wing1_tf_top(x, y, z=0):
    # Nominal thickness
    z_m = -L * np.tan(beta)
    z_u = z_m - 0.5 * TE_height
    beta_u = -np.arctan((z_u + 0.5 * LE_height) / L)
```

(continues on next page)

(continued from previous page)

```
    z_1 = (x - L) * np.tan(beta_u) - LE_height / 2

    # Rounding thickness
    local_y = get_local_y(x)
    z_2 = -(L - x) * rounding_bez(y / local_y).z

    # Sum
    z_val = z_1 + z_2

    return Vector3(x=0, y=0, z=z_val)


def wing1_tf_bot(x, y, z=0):
    z_m = -L * np.tan(beta)
    z_l = z_m + 0.5 * TE_height
    beta_l = -np.arctan((z_l - 0.5 * LE_height) / L)
    z_val = (x - L) * np.tan(beta_l) + LE_height / 2
    return Vector3(x=0, y=0, z=z_val)


def leading_edge_width_function(r):
    temp = Bezier(
        [Vector3(x=0.0, y=0.02), Vector3(x=0.75, y=0.05), Vector3(x=1.0, y=0.05)]
    )
    le_width = temp(r).y
    return le_width

# Define the body component
body = Wing(
    A0=A0,
    A1=A1,
    TT=TT,
    B0=B0,
    Line_B0TT=Line_B0TT,
    top_tf=wing1_tf_top,
    bot_tf=wing1_tf_bot,
    LE_wf=leading_edge_width_function,
)
```

### Wing Geometry

Next, construct the wings using another Wing component.

```
#  fB0--__fB1
#   \      \_
#    \       \
#     B0-----fTT----------------- B1
#     |                        ---___
#     |                              ---__ B2
#     |                                   |
#     A0-------------------------A1---------------TT
```

(continues on next page)

(continued from previous page)

```
flap_thickness = 0.03 * L / L_nom
fflap_angle = np.deg2rad(-flap_angle)

fA0 = Vector3(x=B0.x + flap_length, y=B0.y)
fA1 = Vector3(x=A0.x + 0.5 * wing_length, y=get_local_y(A0.x + 0.5 * wing_length))
fTT = Vector3(x=A0.x + wing_length, y=get_local_y(A0.x + wing_length))

fB0 = Vector3(x=A0.x - wing_TE_back + flap_length, y=B0.y + wing_span)
fB1 = Vector3(x=A0.x + 0.4 * wing_length, y=B0.y + 0.75 * wing_span)

fB0B1 = Line(p0=fB0, p1=fB1)
fB1TT = Line(p0=fB1, p1=fTT)

Line_fB0TT = Polyline([fB0B1, fB1TT])


def wing2_tf_top(x, y, z=0):
    z_ba = -(x - L) * np.tan(beta)
    return Vector3(x=0, y=0, z=-flap_thickness / 2 - z_ba)


def wing2_tf_bot(x, y, z=0):
    z_ba = -(x - L) * np.tan(beta)
    return Vector3(x=0, y=0, z=flap_thickness / 2 - z_ba)

# Define the wing component
wing = Wing(
    A0=fA0,
    A1=fA1,
    TT=fTT,
    B0=fB0,
    Line_B0TT=Line_fB0TT,
    top_tf=wing2_tf_top,
    bot_tf=wing2_tf_bot,
    flap_length=flap_length,
    flap_angle=flap_angle,
    LE_wf=leading_edge_width_function,
)
```

### Inlet Geometry

The inlet is also defined by a Wing component.

```
#     B0------------------------ B1
#     |                            --__
#     |                                --_ B2
#     |                                    |
#     A0-------------------------A1-----------TT
iA0 = A0
iA1 = A1
```

(continues on next page)

(continued from previous page)

```
iTT = Vector3(x=0.5 * (TT.x + B1.x), y=0)
iB0 = Vector3(x=B0.x, y=B0.y)
iB1 = Vector3(x=B1.x, y=B1.y)
iB2 = Vector3(x=iTT.x, y=((iTT.x - B2.x) * (B2.y - B1.y) / (B2.x - B1.x) + B2.y))

iB0B1 = Line(p0=iB0, p1=iB1)
iB1B2 = Line(p0=iB1, p1=iB2)
iB2TT = Line(p0=iB2, p1=iTT)

Line_iB0TT = Polyline([iB0B1, iB1B2, iB2TT])

inlet_start = 0.8 * B1.x
exit_start = 0.3 * (B1.x - B0.x)


def inlet_tf_top(x, y, z=0):
    vehicle_z = wing1_tf_bot(x, y, z).z
    return Vector3(x=0, y=0, z=vehicle_z - 0.05)


def inlet_tf_bot(x, y, z=0):

    # Get z-location of vehicle body
    vehicle_body = wing1_tf_bot(x, y, z).z

    if x < exit_start:
        z_val = (engine_h / (exit_start - A0.x)) * x
    elif x > inlet_start:
        z_val = (-engine_h / (iTT.x - inlet_start)) * (x - iTT.x)
    else:
        z_val = engine_h

    # Calculate taper z
    taper_start_pc = 0.8  # Percentage of y_local where taper begins
    y_local = get_local_y(x)
    if y > taper_start_pc * y_local:
        z_taper = (5 * y / y_local - 4) * z_val
    else:
        z_taper = 0

    return Vector3(x=0, y=0, z=z_val + vehicle_body - z_taper + 0.00001)


def leading_edge_width_function2(r):
    temp = Bezier(
        [Vector3(x=0.0, y=0.001), Vector3(x=0.5, y=0.001), Vector3(x=1.0, y=0.001)]
    )
    le_width = temp(r).y
    return le_width

# Define inlet component
inlet = Wing(
```

(continues on next page)

(continued from previous page)

```
    A0=iA0,
    A1=iA1,
    TT=iTT,
    B0=iB0,
    Line_B0TT=Line_iB0TT,
    top_tf=inlet_tf_top,
    bot_tf=inlet_tf_bot,
    LE_wf=leading_edge_width_function2,
)
```

### Fin Geometry

The final components are the fins. These are created with Fin components.

```
#   |--p1-----p2
#   |  |         \
#   |  |           \
#   |  |             \
#   |--p0_____________p3

fin_p3x = fin_length
fin_thickness = 0.02 * L / L_nom
fin_offset = -(fin_p3x - L) * np.tan(beta)  # offset upwards
y_shift = B0.y

p0 = Vector3(x=rudder_length, y=fin_offset)
p1 = Vector3(x=rudder_length, y=fin_offset + fin_height)
p2 = Vector3(x=0.5 * fin_p3x, y=fin_offset + fin_height)
p3 = Vector3(x=fin_p3x, y=fin_offset)

# Construct p1p3 path
p1p2 = Line(p1, p2)
p2p3 = Line(p2, p3)
p1p3 = Polyline(segments=[p1p2, p2p3])


def fin_offset_function(x, y, z):
    return Vector3(x=0, y=y_shift, z=0)


# Define the fin angles: this is the angle to rotate the fin, constructed in
# the x-y plane, to its final position, by rotating about the x axis
fin_angle = np.deg2rad(-90)

# Rudder angle (fin flap)
rudder_angle = np.deg2rad(rudder_angle)

fin1 = Fin(
    p0=p0,
    p1=p1,
    p2=p2,
```

(continues on next page)

(continued from previous page)

```
    p3=p3,
    fin_thickness=fin_thickness,
    fin_angle=fin_angle,
    top_thickness_function=common.uniform_thickness_function(fin_thickness, "top"),
    bot_thickness_function=common.uniform_thickness_function(fin_thickness, "bot"),
    LE_func=leading_edge_width_function,
    mirror=False,
    rudder_type="sharp",
    rudder_length=rudder_length,
    rudder_angle=rudder_angle,
    pivot_angle=0,
    pivot_point=Vector3(x=0, y=0),
    offset_func=fin_offset_function,
)

# Make second fin to account for rudder angle
def fin_offset_function2(x, y, z):
    return Vector3(x=0, y=-y_shift, z=0)

fin2 = Fin(
    p0=p0,
    p1=p1,
    p2=p2,
    p3=p3,
    fin_thickness=fin_thickness,
    fin_angle=fin_angle,
    top_thickness_function=common.uniform_thickness_function(fin_thickness, "top"),
    bot_thickness_function=common.uniform_thickness_function(fin_thickness, "bot"),
    LE_func=leading_edge_width_function,
    mirror=False,
    rudder_type="sharp",
    rudder_length=rudder_length,
    rudder_angle=rudder_angle,
    pivot_angle=0,
    pivot_point=Vector3(x=0, y=0),
    offset_func=fin_offset_function2,
)
```

### Stacking Components

Now, all of the component defined above can be added to the vehicle using the *Vehicle.add_component()* method.

```
x43.add_component(body, reflection_axis="y")
x43.add_component(wing, reflection_axis="y")
x43.add_component(inlet, reflection_axis="y")
x43.add_component(fin1)
x43.add_component(fin2)
```

### 16.2.4 Vehicle Analysis

After all components have been added to a Vehicle object, the component patches can be generated using the *Vehicle.generate()* method.

```
x43.generate()
```

#### STL Generation

To write the component patches to STL files, use the *Vehicle.to_stl()* method.

```
x43.to_stl(prefix="x43")
```

#### Inertial Estimates

The inertial properties of a vehicle can be also estimated using the STL meshes generated by *hypervehicle*. This requires providing densities for each component. In the example below, such densities were calculated from the known densities of the X-43A. The result is such that the effective densities multiplied by the component volumes results in the nominal mass of the X-43A.

Adjusting the design parameters will impact the resultant volumes calculated, and so the mass model will be impacted also.

All you need to do is define a `densities` dictionary, containing the density of each component of the vehicle, as enumerated in `Vehicle._enumerated_components`. Then, use the *Vehicle.analyse()* command, as shown below. This will also save the vehicle's inertial properties as attributes.

```
densities = {
    "wing_1": 1680,
    "wing_2": 5590,
    "wing_3": 1680,
    "fin_1": 5590,
    "fin_2": 5590,
}
volume, mass, cog, inertia = x43.analyse(densities)
```

CHAPTER

# SEVENTEEN

# GENERATING PARAMETER SENSITVITIES

This tutorial will demonstrate how to generate geometric parameter sensitivities using HyperVehicle.

**See also:**

This example follows on from the *sharp wedge* tutorial.

## 17.1 Workflow

### 17.1.1 Nominal Geometry

First, create the nominal geometry configuration. For this example, we will be using the *sharp wedge* geometry constructed previously.

### 17.1.2 Parametric Geometry Generator

The next step is to refactor the geometry generation code into a `ParametricGenerator` object. This object inherits from the *Generator* object. This class has two methods: the *Generator.__init__()* method, where all geometric parameters are passed as arguments, and the *create_instance()* method, which returns a *Vehicle* object ready to be *generate()*'d.

The code below provides an example of this class. Note that the named `kwargs` are passed to `super().__init__` to be unpacked and overwrite the default parameters. See *Generator* for more details.

```
from hypervehicle import Vehicle
from hypervehicle.components import Fuselage
from hypervehicle.generator import Generator
from hypervehicle.geometry import Vector3, Line, CoonsPatch


class ParametricWedge(Generator):
    def __init__(self, **kwargs) -> None:
        # Wedge parameters
        self.wingspan = 1
        self.chord = 1
        self.thickness = 0.1

        # Complete instantiation
        super().__init__(**kwargs)
```

(continues on next page)

(continued from previous page)

```
    def create_instance(self) -> Vehicle:
        # Create vehicle object
        wedge = Vehicle()
        wedge.configure(name="Wedge", verbosity=1)

        # Define wedge cross-section points
        #                    ^ +y
        #                    |
        #             W    _ - _   N        ___
        #             _ -    |   - _         |
        #  +x <--- < -------------- >        | thickness
        #             - _    |   _ -         |
        #             S   -  _ -     E      ___
        #
        #          |-----------------|
        #                wingspan

        NW = Vector3(x=0, y=0.5 * self.thickness)
        NE = Vector3(
            x=-0.5 * self.chord,
            y=0,
        )
        SE = Vector3(x=0, y=-0.5 * self.thickness)
        SW = Vector3(x=0.5 * self.chord, y=0)

        # Define patches forming wedge
        sections = []
        for i in [-1, 1]:
            z_loc = 0.5 * i * self.wingspan
            axial_shift = Vector3(x=0, y=0, z=z_loc)

            N = Line(p0=NW + axial_shift, p1=NE + axial_shift)
            S = Line(p0=SW + axial_shift, p1=SE + axial_shift)
            E = Line(p0=SE + axial_shift, p1=NE + axial_shift)
            W = Line(p0=SW + axial_shift, p1=NW + axial_shift)

            patch = CoonsPatch(north=N, south=S, east=E, west=W)
            sections.append(patch)

        fuselage = Fuselage(
            cross_sections=sections,
            sweep_axis="z",
            stl_resolution=10,
        )
        wedge.add_component(fuselage)

        # Generate STL
        return wedge
```

Note that the nominal geometry could be generated using the generator object as well:

```
parametric_wedge_generator = ParametricWedge()
```

(continues on next page)

(continued from previous page)

```
wedge = parametric_wedge_generator.create_instance()
wedge.generate()
```

This is a good check to ensure that the generator has been correctly implemented.

### 17.1.3 Run the Sensitivity Study

With the steps above completed, you can run a sensitivity study by creating an instance of *SensitivityStudy*.

Then, define the design parameters which you would like to get sensitivities to. In this example, we are extracting the geometric sensitivities to each wedge parameter: the thickness, the chord and the wingspan. We specify this with the `parameters` dictionary, passing along the nominal values to perturb about.

Finally, run the study by calling the *dvdp()* method, passing in the design parameters.

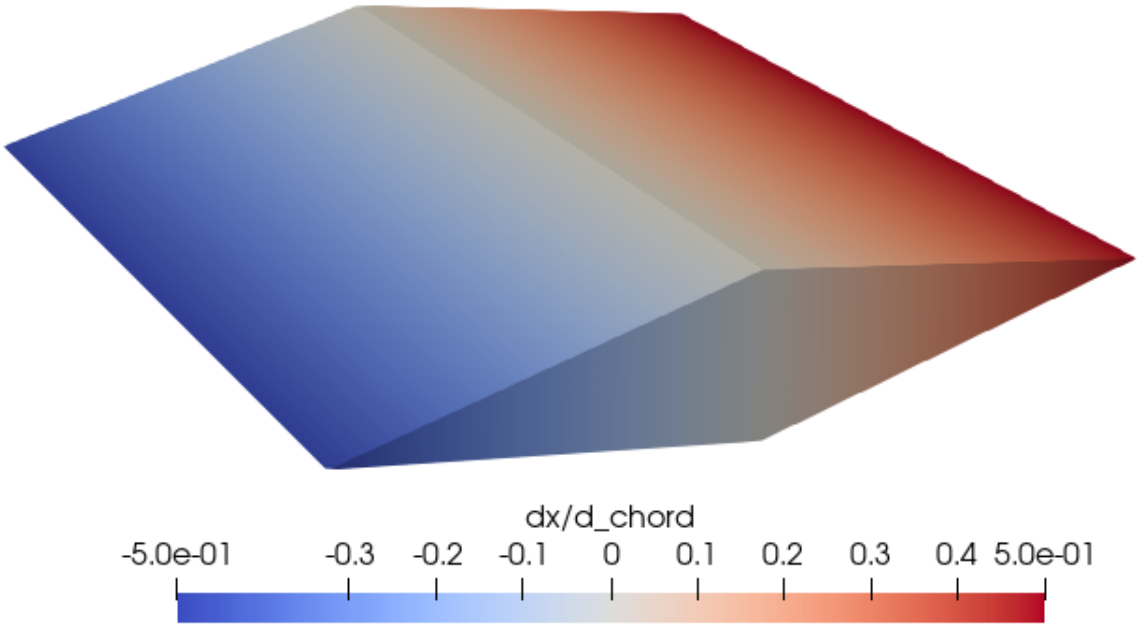
```
from hypervehicle.utilities import SensitivityStudy

# Construct sensitivity study
ss = SensitivityStudy(ParametricWedge)

# Define parameters to get sensitivities to
parameters = {'thickness': 0.1, 'chord': 1, 'wingspan': 1}

# Perform study
sensitivities = ss.dvdp(parameters)

# Save to CSV
ss.to_csv()
```

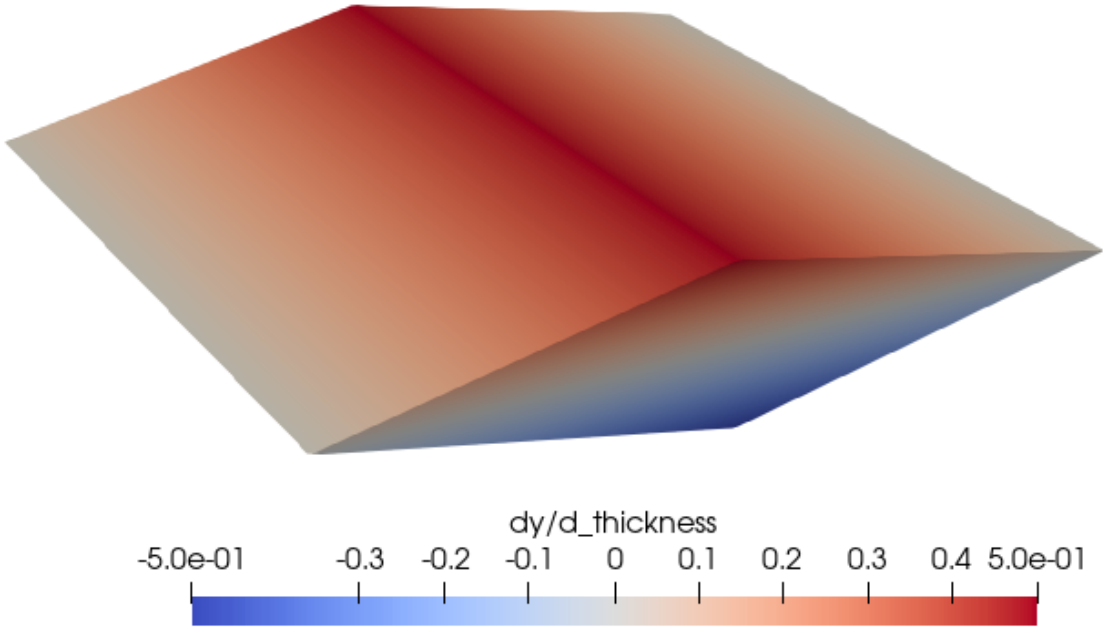
The output of `dvdp` is a nested dictionary, containing pandas `DataFrame`s for each parameter and component of the Vehicle object.

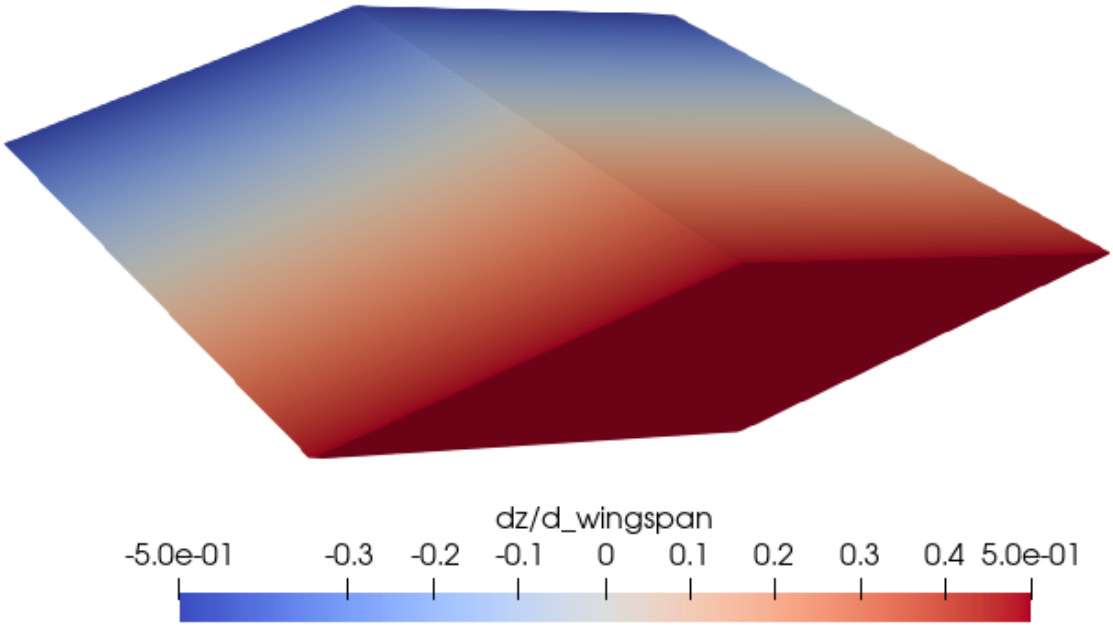
You can also save the results of the study using the `to_csv` method. This will create a unique file for each component.

## 17.2 Visualisation of Sensitivities

The `.csv` files can be opened in ParaView. After opening the file, select Table to Points (set `x`, `y` and `z` columns before applying), then Delaunay 3D filter.




dz/d_wingspan
-5.0e-01
-0.3
-0.2
-0.1
0
0.1
0.2
0.3
0.4
5.0e-01

CHAPTER

# EIGHTEEN

# HYPERVEHICLE VEHICLE CLASS

class hypervehicle.vehicle.**Vehicle**(***kwargs*)

**add_component**(*component:* Component, *name: str = None*, *reflection_axis: str = None*, *append_reflection: bool = True*, *curvatures: List[Tuple[str, Callable, Callable]] = None*, *clustering: Dict[str, Callable] = None*, *transformations: List[Tuple[str, Any]] = None*, *modifier_function: Callable | None = None*, *ghost: bool | None = False*) → None

Adds a new component to the vehicle.

**Parameters**

- **component** (Component) – The component to add.
- **name** (*str, optional*) – The name to assign to this component. If provided, it will be used when writing to STL. The default is None.
- **reflection_axis** (*str, optional*) – Include a reflection of the component about the axis specified (eg. 'x', 'y' or 'z'). The default is None.
- **append_reflection** (*bool, optional*) – When reflecting a new component, add the reflection to the existing component, rather than making it a new component. This is recommended when the combined components will form a closed mesh, but if the components will remain as two isolated bodies, a new component should be created (ie. append_reflection=False). In this case, you can use copy.deepcopy to make a copy of the reflected component when adding it to the vehicle. See the finner example in the hypervehicle hangar. The default is True.
- **curvatures** (*List[Tuple[str, Callable, Callable]], optional*) – A list of the curvatures to apply to the component being added. This list contains a tuple for each curvature. Each curvatue is defined by (axis, curve_func, curve_func_derivative). The default is None.
- **clustering** (*Dict[str, Callable], optional*) – Optionally provide clustering options for the stl meshes. See parametricSurfce2stl for more information. The default is None.
- **transformations** (*List[Tuple[str, Any]], optional*) – A list of transformations to apply to the nominal component. The default is None.
- **modifier_function** (*Callable, optional*) – A function which accepts x,y,z coordinates and returns a Vector3 object with a positional offset. This function is used with an OffsetPatchFunction. The default is None.
- **ghost** (*bool, optional*) – Add a ghost component. When True, this component will be excluded from the files written to STL.

**See also:**

*Vehicle.add_vehicle_transformations*

**add_property**(*name: str*, *value: float*)

Add a named property to the vehicle. Currently only supports float property types.

**add_vehicle_transformations**(*transformations: List[Tuple[str, Any]]*) → None

Add transformations to apply to the vehicle after running generate(). Each transformation in the list should be a tuple of the form (transform_type, *args), where transform_type can be "rotate", or "translate". Note that transformations can be chained.

## 18.1 Extended Summary

- "rotate" : rotate the entire vehicle. The *args for rotate are

angle (float) and axis (str). For example: *[("rotate", 180, "x"), ("rotate", 90, "y")]*.

- "translate" : translate the entire vehicle. The *args for translate

includes the translational offset, specified either as a function (Callable), or as Vector3 object.

**analyse**(*densities: dict*) → Tuple

Evaluates the mesh properties of the vehicle instance.

**Parameters**

**densities** (*Dict[str, float]*) – A dictionary containing the effective densities for each component. Note that the keys of the dict must match the keys of vehicle._named_components. These keys will be consistent with any name tags assigned to components.

**Returns**

- **total_volume** (*float*) – The total volume.
- **total_mass** (*float*) – The toal mass.
- **composite_cog** (*np.array*) – The composite center of gravity.
- **composite_inertia** (*np.array*) – The composite mass moment of inertia.

**analyse_after_generating**(*densities: Dict[str, Any]*) → None

Run the vehicle analysis method immediately after generating patches. Results will be saved to the analysis_results attribute of the vehicle.

**Parameters**

**densities** (*Dict[str, Any]*) – A dictionary containing the effective densities for each component. Note that the keys of the dict must match the keys of vehicle._named_components. These keys will be consistent with any name tags assigned to components.

**configure**(*name: str = None*, *verbosity: int = 1*)

Configure the Vehicle instance.

**generate**()

Generate all components of the vehicle.

**get_non_ghost_components**() → dict[str, *Component*]

Returns all non-ghost components.

**to_stl**(*prefix: str = None*, *merge: bool | List[str] = False*) → None

Writes the vehicle components to STL file. If analysis results are present, they will also be written to file, either as CSV, or using the Numpy tofile method.

**Parameters**

- **prefix** (*str, optional*) – The prefix to use when saving components to STL. Note that if components have been individually assigned name tags, the prefix provided will take precedence. If no prefix is specified, and no component name tag is available, the Vehicle name will be used. The default is None.
- **merge** (*[bool, list[str]], optional*) – Merge components of the vehicle into a single STL file. The merge argument can either be a boolean (with True indicating to merge all components of the vehicle), or a list of the component names to merge. This functionality depends on PyMesh. The default is False.

**See also:**

`utilities.merge_stls`

**transform**(*transformations: List[Tuple[str, Any]]*) → None

Transform vehicle by applying the tranformations.

CHAPTER

# NINETEEN

# HYPERVEHICLE COMPONENTS

## 19.1 Component Definitions

### 19.1.1 Revolved Component

class hypervehicle.components.revolved.**RevolvedComponent**(*revolve_line*, *stl_resolution: int = 4*, *verbosity: int = 1*, *name: str = None*)

Bases: *Component*

**__init__**(*revolve_line*, *stl_resolution: int = 4*, *verbosity: int = 1*, *name: str = None*) → None

Create a revolved component.

**Parameters**

**revolve_line** (*Line|PolyLine|Bezier*) – A line to be revolved about the primary axis.

**generate_patches**()

Generates the parametric patches from the parameter dictionary.

### 19.1.2 Swept Component

class hypervehicle.components.swept.**SweptComponent**(*cross_sections: List[CoonsPatch]*, *sweep_axis: str = 'z'*, *stl_resolution: int = 2*, *verbosity: int = 1*, *name: str = None*)

Bases: *Component*

**__init__**(*cross_sections: List[CoonsPatch]*, *sweep_axis: str = 'z'*, *stl_resolution: int = 2*, *verbosity: int = 1*, *name: str = None*) → None

Create a swept component.

**Parameters**

- **cross_sections** (*list, optional*) – A list of cross-sectional patches to sweep through.
- **sweep_axis** (*str, optional*) – The axis to sweep the cross sections through. The default is z.

**generate_patches**()

Generates the parametric patches from the parameter dictionary.

### 19.1.3 Wing Component

class hypervehicle.components.wing.**Wing**(*A0: Vector3 = Vector3(x=0, y=0, z=0), A1: Vector3 = Vector3(x=0, y=0, z=0), TT: Vector3 = Vector3(x=0, y=0, z=0), B0: Vector3 = Vector3(x=0, y=0, z=0), Line_B0TT: Polyline = None, Line_B0TT_TYPE: str = 'Bezier', t_B1: float = None, t_B2: float = None, top_tf: Callable[[float, float, float], Vector3] = None, bot_tf: Callable[[float, float, float], Vector3] = None, LE_wf: Callable[[float], Vector3] = None, LE_type: str = 'custom', tail_option: str = 'FLAP', flap_length: float = 0, flap_angle: float = 0, mirror: bool = True, mirror_new_component: bool = False, close_wing: bool = False, stl_resolution: int = 2, verbosity: int = 1, name: str = None*)

Bases: *Component*

**__init__**(*A0: Vector3 = Vector3(x=0, y=0, z=0), A1: Vector3 = Vector3(x=0, y=0, z=0), TT: Vector3 = Vector3(x=0, y=0, z=0), B0: Vector3 = Vector3(x=0, y=0, z=0), Line_B0TT: Polyline = None, Line_B0TT_TYPE: str = 'Bezier', t_B1: float = None, t_B2: float = None, top_tf: Callable[[float, float, float], Vector3] = None, bot_tf: Callable[[float, float, float], Vector3] = None, LE_wf: Callable[[float], Vector3] = None, LE_type: str = 'custom', tail_option: str = 'FLAP', flap_length: float = 0, flap_angle: float = 0, mirror: bool = True, mirror_new_component: bool = False, close_wing: bool = False, stl_resolution: int = 2, verbosity: int = 1, name: str = None*) → None

Creates a new fin component.

**Parameters**

- **A0** (*Vector3*) – Point p0 of the fin geometry.
- **A1** (*Vector3*) – Point p1 of the fin geometry.
- **TT** (*Vector3*) – Point p2 of the fin geometry.
- **B0** (*Vector3*) – Point p3 of the fin geometry.
- **Line_B0TT** (*Polyline*) – The thickness of the fin.
- **Line_B0TT_TYPE** (*str, optional*) – The axial position angle of the placement of the fin.
- **t_B1** (*float, optional*) – The t value of the first discretisation point. The default is None.
- **t_B2** (*float, optional*) – The t value of the second discretisation point. The default is None.
- **top_tf** (*Callable*) – The thickness function for the top surface of the wing.
- **bot_tf** (*Callable*) – The thickness function for the top surface of the wing.
- **LE_wf** (*Callable, optional*) – The thickness function for the leading edge of the wing.
- **LE_type** (*str, optional*) – The type of LE to create, either “FLAT” or “custom”. The default is “custom”.
- **tail_option** (*str, optional*) – The type of trailing edge to use, currently only “FLAP”. The default is “FLAP”.
- **flap_length** (*float, optional*) – The length of the trailing edge flap. The default is 0.

- **flap_angle** (*float, optional*) – The angle of the flap, specified in radians. The default is 0.
- **mirror** (*bool, optional*) – Mirror the wing. The default is False.
- **mirror_new_component** (*bool, optional*) – Create a new component for the mirrored patches. The default is False.
- **close_wing** (*bool, optional*) – If the wing is not being mirrored, it is useful to set this to True, to close the STL object. The default is False.
- **stl_resolution** (*int, optional*) – The stl resolution to use when creating the mesh for this component. The default is None.
- **verbosity** (*int, optional*) – The verbosity of the component. The default is 1.
- **name** (*str, optional*) – The name tag for the component. The default is None.

**generate_patches**()

Generates the parametric patches from the parameter dictionary.

### 19.1.4 Fin Component

**class** hypervehicle.components.fin.**Fin**(*p0: Vector3*, *p1: Vector3*, *p2: Vector3*, *p3: Vector3*, *fin_thickness: float*, *fin_angle: float*, *top_thickness_function: Callable*, *bot_thickness_function: Callable*, *LE_wf: Callable | None = None*, *mirror: bool | None = False*, *rudder_type: str | None = 'flat'*, *rudder_length: float | None = 0*, *rudder_angle: float | None = 0*, *pivot_angle: float | None = 0*, *pivot_point: Vector3 | None = Vector3(x=0, y=0, z=0.0)*, *offset_func: Callable | None = None*, *stl_resolution: int | None = 2*, *verbosity: int | None = 1*, *name: str | None = None*)

Bases: *Component*

**__init__**(*p0: Vector3*, *p1: Vector3*, *p2: Vector3*, *p3: Vector3*, *fin_thickness: float*, *fin_angle: float*, *top_thickness_function: Callable*, *bot_thickness_function: Callable*, *LE_wf: Callable | None = None*, *mirror: bool | None = False*, *rudder_type: str | None = 'flat'*, *rudder_length: float | None = 0*, *rudder_angle: float | None = 0*, *pivot_angle: float | None = 0*, *pivot_point: Vector3 | None = Vector3(x=0, y=0, z=0.0)*, *offset_func: Callable | None = None*, *stl_resolution: int | None = 2*, *verbosity: int | None = 1*, *name: str | None = None*) → None

Creates a new fin component.

**Parameters**

- **p0** (*Vector3*) – Point p0 of the fin geometry.
- **p1** (*Vector3*) – Point p1 of the fin geometry.
- **p2** (*Vector3*) – Point p2 of the fin geometry.
- **p3** (*Vector3*) – Point p3 of the fin geometry.
- **fin_thickness** (*float*) – The thickness of the fin.
- **fin_angle** (*float*) – The axial position angle of the placement of the fin.
- **top_thickness_function** (*Callable*) – The thickness function for the top surface of the fin.
- **bot_thickness_function** (*Callable*) – The thickness function for the top surface of the fin.

- **LE_wf** (*Callable, optional*) – The thickness function for the leading edge of the fin.
- **mirror** (*bool, optional*) – Mirror the fin. The default is False.
- **rudder_type** (*str, optional*) – The type of rudder to use, either "flat" or "sharp". The default is "flat".
- **rudder_length** (*float, optional*) – The length of the rudder. The default is 0.
- **pivot_angle** (*float, optional*) – The pivot angle of the entire fin, about its central axis. The default is 0.
- **pivot_point** (*Vector3, optional*) – The point about which to apply the pivot_angle. The default is Vector3(0,0,0).
- **offset_func** (*Callable, optional*) – The function to apply when offsetting the fin position. The default is None.
- **stl_resolution** (*int, optional*) – The stl resolution to use when creating the mesh for this component. The default is None.
- **verbosity** (*int, optional*) – The verbosity of the component. The default is 1.
- **name** (*str, optional*) – The name tag for the component. The default is None.

**generate_patches**()

Generates the parametric patches from the parameter dictionary.

### 19.1.5 Composite Component

**class** hypervehicle.components.composite.**CompositeComponent**(*stl_resolution: int = 2*, *verbosity: int = 1*, *name: str = None*)

Bases: *Component*

A composite component.

This Component allows adding multiple components to it, which will all form the patches of this component. Note that this means all components added will share the same stl resolution, regardless of what their previously assigned resolution was. This is because all individual component patches become merged to form the CompositeComponent.patches.

**__init__**(*stl_resolution: int = 2*, *verbosity: int = 1*, *name: str = None*) → None

**add_component**(*component:* Component, *reflection_axis: str = None*, *append_reflection: bool = True*, *curvatures: List[Tuple[str, Callable, Callable]] = None*, *clustering: Dict[str, float] = None*, *transformations: List[Tuple[str, Any]] = None*) → None

Adds a new component to the vehicle.

**Parameters**

- **component** (Component) – The component to add.
- **reflection_axis** (*str, optional*) – Include a reflection of the component about the axis specified (eg. 'x', 'y' or 'z'). The default is None.
- **append_reflection** (*bool, optional*) – When reflecting a new component, add the reflection to the existing component, rather than making it a new component. The default is True.

- **curvatures** (*List[Tuple[str, Callable, Callable]], optional*) – A list of the curvatures to apply to the component being added. This list contains a tuple for each curvature. Each curvatue is defined by (axis, curve_func, curve_func_derivative). The default is None.
- **clustering** (*Dict[str, float], optional*) – Optionally provide clustering options for the stl meshes. The default is None.
- **transformations** (*List[Tuple[str, Any]], optional*) – A list of transformations to apply to the nominal component. The default is None

**generate_patches**()

Generates the parametric patches from the parameter dictionary.

### 19.1.6 Other Components

**class** hypervehicle.components.polygon.**Cube**(*a: float*, *centre: Vector3 = Vector3(x=0, y=0, z=0)*, *stl_resolution: int = 2*, *verbosity: int = 1*, *name: str = None*)

Bases: *Component*

**__init__**(*a: float*, *centre: Vector3 = Vector3(x=0, y=0, z=0)*, *stl_resolution: int = 2*, *verbosity: int = 1*, *name: str = None*) → None

**Parameters**

- **a** (*float*) – The cube side half-length.
- **centre** (*Vector3, optional*) – The centre point of the cube. The default is Vector3(0,0,0).

**generate_patches**()

Generates the parametric patches from the parameter dictionary.

**class** hypervehicle.components.polygon.**Sphere**(*r: float*, *centre: Vector3 = Vector3(x=0, y=0, z=0)*, *stl_resolution: int = 2*, *verbosity: int = 1*, *name: str = None*)

Bases: *Component*

**__init__**(*r: float*, *centre: Vector3 = Vector3(x=0, y=0, z=0)*, *stl_resolution: int = 2*, *verbosity: int = 1*, *name: str = None*) → None

**Parameters**

- **r** (*float*) – The radius of the sphere.
- **centre** (*Vector3*) – The centre point of the sphere. The default is Vector3(0,0,0).

**generate_patches**()

Generates the parametric patches from the parameter dictionary.

## 19.2 Base Component

class hypervehicle.components.component.**Component**(*params: dict = None*, *stl_resolution: int = 2*, *verbosity: int = 1*, *name: str = None*)

**add_clustering_options**(*i_clustering_func: Callable | None = None*, *j_clustering_func: Callable | None = None*)

Add a clustering option to this component.

**Parameters**

- **i_clustering_func** (*Callable, optional*) – The clustering function in the i direction. The default is None.
- **j_clustering_func** (*Callable, optional*) – The clustering function in the j direction. The default is None.

**analyse**()

Evaluates properties of the STL mesh.

**curve**()

Applies a curvature function to the parametric patches.

**grid**()

Creates a discrete grid from the parametric patches.

**reflect**(*axis: str = None*)

Reflects the parametric patches.

**rotate**(*angle: float = 0*, *axis: str = 'y'*)

Rotates the parametric patches.

**surface**(*resolution: int = None*)

Creates the discretised surface data from the parametric patches.

**to_stl**(*outfile: str = None*)

Writes the component to STL file format.

**to_vtk**()

Writes the component to VTK file format.

CHAPTER

# TWENTY

# HYPERVEHICLE UTILITIES

class hypervehicle.utilities.**SensitivityStudy**(*vehicle_constructor*, *verbosity: int | None = 1*)

Computes the geometric sensitivities using finite differencing.

**static _combine**(*nominal_instance*, *sensitivities*)

Combines the sensitivity information for multiple parameters.

**static _compare_meshes**(*mesh1*, *mesh2*, *dp*, *parameter_name: str*) → DataFrame

Compares two meshes with each other and applies finite differencing to quantify their differences.

**Parameters**

- **mesh1** (*Mesh*) – The reference mesh.
- **mesh1** – The perturbed mesh.
- **dp** (*float*) – The parameter perturbation.
- **parameter_name** (*str*) – The name of the parameter.

**Returns**

**df** – A DataFrame of the finite difference results.

**Return type**

pd.DataFrame

**dvdp**(*parameter_dict: dict[str, any]*, *overrides: dict[str, any] | None = None*, *perturbation: float | None = 5*, *write_nominal_stl: bool | None = True*, *nominal_stl_prefix: str | None = None*)

Computes the sensitivity of the geometry with respect to the parameters.

**Parameters**

- **parameter_dict** (*dict*) – A dictionary of the design parameters to perturb, and their nominal values.
- **overrides** (*dict, optional*) – Optional vehicle generator overrides to provide along with the parameter dictionary without variation. The default is None.
- **perturbation** (*float, optional*) – The design parameter perturbation amount, specified as percentage. The default is 20.
- **vehicle_creator_method** (*str, optional*) – The name of the method which returns a hypervehicle.Vehicle instance, ready for generation. The default is 'create_instance'.
- **write_nominal_stl** (*bool, optional*) – A boolean flag to write the nominal geometry STL(s) to file. The default is True.
- **nominal_stl_prefix** (*str, optional*) – The prefix to append when writing STL files for the nominal geometry. If None, no prefix will be used. The default is None.

**Returns**
**sensitivities** – A dictionary containing the sensitivity information for all components of the geometry, relative to the nominal geometry.

**Return type**
dict

`to_csv`(*outdir: str | None = None*)

Writes the sensitivity information to CSV file.

**Parameters**
**outdir** (*str, optional*) – The output directory to write the sensitivity files to. If None, the current working directory will be used. The default is None.

**Returns**
**combined_data_filepath** – The filepath to the combined sensitivity data.

**Return type**
str

hypervehicle.utilities.**append_sensitivities_to_tri**(*dp_filenames: List[str]*, *components_filepath: str | None = 'Components.i.tri'*, *match_tolerance: float | None = 1e-05*, *rounding_tolerance: float | None = 1e-08*, *combined_sens_fn: str | None = 'all_components_sensitivity.csv'*, *outdir: str | None = None*, *verbosity: int | None = 1*) → float

Appends shape sensitivity data to .i.tri file, and writes the sensitivity data to csv file too. This step is required for geometries with multiple components. The .tri file is used to match individual sensitivity files (dp_filenames) to the geometry. The combined sensitivity file is required to calculate flow sensitivities with the .plt file, which has local flow conditions attached.

**Parameters**

- **dp_files** (*list[str]*) – A list of the file names of the sensitivity data.
- **components_filepath** (*str, optional*) – The filepath to the .tri file to be appended to. The default is 'Components.i.tri'.
- **match_tolerance** (*float, optional*) – The precision tolerance for matching point coordinates. The default is 1e-5.
- **rounding_tolerance** (*float, optional*) – The tolerance to round data off to. The default is 1e-8.
- **combined_sens_fn** (*str, optional*) – The filename of the combined geometry sensitivity data. The default is "all_components_sensitivity.csv".
- **outdir** (*str, optional*) – The output directory to write the combined sensitivity file to. If None, the current working directory will be used. The default is None.
- **verbosity** (*int, optional*) – The verbosity of the code. The defualt is 1.

**Returns**
**match_fraction** – The fraction of cells which got matched. If this is below 100%, try decreasing the match tolerance and run again.

**Return type**
float

### Examples

```
>>> dp_files = ['wing_0_body_width_sensitivity.csv',
                'wing_1_body_width_sensitivity.csv']
```

hypervehicle.utilities.**assess_inertial_properties**(*vehicle*, *component_densities: Dict[str, float]*)

**Parameters**

- **vehicle** (Vehicle) – A hypervehicle Vehicle instance.
- **component_densities** (*Dict[str, float]*) – A dictionary containing the effective densities for each component. Note that the keys of the dict must match the keys of vehicle._named_components.

**Returns**

- **vehicle_properties** (*dict*) – A dictionary containing the vehicle's mass, volume, location of CoG and moment of inertia matrix.
- **component_properties** (*dict*) – A dictionary containing the same keys as vehicle_properties, but the values are now dictionaries for each component of the vehicle.

hypervehicle.utilities.**csv_to_delaunay**(*filepath: str*)

Converts a csv file of points to a Delaunay3D surface.

**Parameters**
**filepath** (*str*) – The filepath to the CSV file.

hypervehicle.utilities.**merge_stls**(*stl_files: List[str]*, *name: str | None = None*, *verbosity: int | None = 1*) → str

Merge STL files into a single file. Note that this function depends on having PyMesh installed.

**Parameters**

- **stl_files** (*list[str]*) – A list of the STL file names to be merged.
- **name** (*str, optional*) – The prefix of the combined STL filename output.
- **verbosity** (*int, optional*) – The function verbosity. The default is 1.

**Returns**
**outfile** – The filename of the merged STL.

**Return type**
str

hypervehicle.utilities.**parametricSurfce2stl**(*parametric_surface*, *triangles_per_edge: int*, *si: float = 1.0*, *sj: float = 1.0*, *mirror_y=False*, *flip_faces=False*, *i_clustering_func: callable = None*, *j_clustering_func: callable = None*) → Mesh

Function to convert parametric_surface generated using the Eilmer Geometry Package into a stl mesh object.

**Parameters**

- **parametric_surface** (*Any*) – The parametric surface object.
- **si** (*float, optional*) – The clustering in the i-direction. The default is 1.0.
- **sj** (*float, optional*) – The clustering in the j-direction. The default is 1.0.
- **triangles_per_edge** (*int*) – The resolution for the stl object.

- **mirror_y** (*bool, optional*) – Create mirror image about x-z plane. The default is False.
- **i_clustering_func** (*callable, optional*) – A custom clustering function to apply in the i direction. The default is None.
- **j_clustering_func** (*callable, optional*) – A custom clustering function to apply in the j direction. The default is None.

**Returns**
**stl_mesh** – The numpy-stl mesh.

**Return type**
Mesh

hypervehicle.utilities.**print_banner**()

Prints the hypervehicle banner

CHAPTER

# TWENTYONE

# HYPERVEHICLE GENERATOR

**class** hypervehicle.generator.**AbstractGenerator**(***kwargs*)

Abstract Generator Interface.

**abstract __init__**(***kwargs*) → None

**abstract create_instance**() → *Vehicle*

**class** hypervehicle.generator.**Generator**(***kwargs*)

Hypervehicle Parametric Generator.

**__init__**(***kwargs*) → None

Initialises the generator.

CHAPTER

# TWENTYTWO

# CONTRIBUTION GUIDELINES

To contribute to `HyperVehicle`, please read the instructions below to maintain the styling of the code.

## 22.1 Seek Feedback Early

Please open an issue before a pull request to discuss any changes you wish to make.

## 22.2 Setting up for Development

1. Create a new Python virtual environment to isolate the package. You can do so using `venv` or anaconda.
2. Install the code in editable mode using the command below (run from inside the `hypervehicle` root directory).

```
pip install -e .[all]
```

3. Install the pre-commit hooks. This will check that your changes are formatted correctly before you make any commits.

```
pre-commit install
```

4. Start developing! After following the steps above, you are ready to start developing the code. Make sure to follow the guidelines below.

## 22.3 Developing *HyperVehicle*

- Before making any changes, fork this repository and clone it to your machine.
- Run black on any code you modify. This formats it according to PEP8 standards.
- Document as you go: use numpy style docstrings, and add to the docs where relevant.
- Write unit tests for the code you add, and include them in `tests/`. This project uses pytest.
- Commit code regularly to avoid large commits with many unrelated changes.
- Write meaningful commit messages, following the Conventional Commits standard. This is used for the purpose of maintaining semantic versioning and automated *changelogs*. The Python package `commitizen` is a great tool to help with this, and is already configured for this repo. Simply stage changed code, then use the `cz c` command to make a commit. If you do not wish to do this, your commits will be squashed before being merged into `main`.
- Open a Pull Request when your code is complete and ready to be merged.

## 22.4 Building the Docs

To build the documentation, run the commands below.

```
cd docs/
make html
xdg-open build/html/index.html
```

If you are actively developing the docs, consider using sphinx-autobuild. This will continuosly update the docs for you to see any changes live, rather than re-building repeatadly. From the root directory, run the following command:

```
sphinx-autobuild docs/source/ docs/build/html --open-browser
```

CHAPTER

# TWENTYTHREE

## CITING HYPERVEHICLE

Coming soon!

CHAPTER

# TWENTYFOUR

# CHANGELOG

## 24.1 v0.5.0 (2024-01-02)

### 24.1.1 Feat

- **Component**: added add_clustering_options method
- **utilities**: improve verbosity of senstivity study
- **SensitivityStudy**: optionally provide generator overrides
- **SensitivityStudy**: improved verbosity
- **merge_stls**: return merged stl filename
- **merge_stls**: added verbosity control
- **SensitivityStudy**: writing results to csv also writes combined csv
- **utilities**: added print_banner utility function
- **hangar**: add waverider to hangar
- **Component**: ghost components
- **Vehicle**: include option to merge all stls when writing to_stl
- **utilities**: added stl mesh merge function
- **SensitivityStudy**: calculates and saves component volume and mass sensitivities
- **Vehicle**: component volume and mass included in vehicle assessment
- **Component**: clean mesh on writing to stl
- **Vehicle**: allow specifying component modifier function to manipulate surfaces
- **parametricSurfce2stl**: allow passing custom clustering function
- **Vehicle**: ability to define and differentiate vehicle properties

### 24.1.2 Fix

- **CompositeComponent**: added sphere and cube to allowable components
- **hangar**: include waverider in hangar namespace loading
- **Vehicle.to_stl**: exclude ghost components in stl merge

### 24.1.3 Refactor

- **Component**: only clean mesh on writing to STL
- **Component**: clean STL mesh when generating mesh object instead of on saving to file
- **SensitivityStudy**: component volmass sens replaces just vehicle sens

## 24.2 v0.4.0 (2023-03-10)

### 24.2.1 Feat

- **Component**: generate patch surfaces concurrently

## 24.3 v0.3.0 (2023-02-28)

### 24.3.1 Feat

- **append_sensitivities_to_tri**: allow specifying outdir for combined sens data file
- **Vehicle**: transformation generalised to accept different transform types
- **SensitivityStudy**: added scalar sensitivities (vol, mass, etc)
- **Vehicle**: optionally analyse the vehicle after generation and write results to file

### 24.3.2 Fix

- **Wing**: fixed wing closing method to align with planforms
- **SensitivityStudy**: processing of scalar sensitivities
- **hangar**: close wing of X43
- **ParametricReFEX**: canard angle for reflected cannard

### 24.3.3 Refactor

- output scalar vehicle properties and sensitivities to dedicated directories

## 24.4 v0.2.2 (2023-02-06)

### 24.4.1 Fix

- **hangar**: add finner to hangar namespace import

## 24.5 v0.2.1 (2023-02-06)

### 24.5.1 Fix

- **Component**: overload copy and deepcopy dunders
- **utilities.py**: use component name tags where possible
- **README.md**: broken link to x43a example docs

## 24.6 v0.2.0 (2023-02-02)

### 24.6.1 Feat

- **polygon.py**: updated polygon to standard component type
- **SensitivityStudy**: allow passing nominal stl prefix
- **Vehicle**: improved verbosity of to_stl and prefix control
- **Vehicle**: component name tags will be used when writing to stl
- **Component**: improved repr
- **SensitivityStudy**: allow passing outdir when saving to csv
- **hangar**: expose all vehicles in hangar to package
- **Vehicle**: allow specifying vehicle transformations prior to generate()
- **CompositeComponent**: added new component type
- **append_sensitivities_to_tri**: added control of matching tolerances
- **append_sensitivities_to_tri**: write csv of combined sensitivity data
- **append_sensitivities_to_tri**: capability for multiple parameters
- **Vehicle**: allow chained transformations of components
- **common.py**: added circle patch function
- ability to cluster stl meshing

### 24.6.2 Fix

- **hangar**: updated vehicles to migrate to general components
- **hangar**: call super()
- **hangar**: inherit Generator instead of AbstractGenerator
- **hangar**: attributes for vehicle parameters
- **append_sensitivities_to_tri**: do not write index to csv

### 24.6.3 Refactor

- **Fuselage**: split fuselage component into more general revolved and swept components
- **scripts**: deleted redundant scripts directory

## 24.7 v0.1.0 (2023-01-24)

### 24.7.1 Feat

- **Generator**: implemented base generator class
- **Vehicle**: implemented analyse method
- **utilities**: reimplemented interial properties utility
- **SensitivityStudy**: reimplemented
- **Vehicle**: added enumerated_components attribute
- **Component**: added Component to hypervehicle.components namespace
- **AbstractGenerator**: added abstract generator class
- **Vehicle**: added banner
- **hifire8.py**: translated hifire8 model
- **hifire4.py**: translated hifire4 model
- **Component**: improved component curvature implementation
- **x43.py**: translated x43 model
- **rocket.py**: translated generic rocket model
- **falcon9.py**: translated falcon 9 model
- **htv.py**: translated HTV model
- **Vehicle**: implemented control of component reflections
- **Component**: implemented mesh analysis method
- **Component**: implemented stl_resolution arg
- **Vehicle**: added verbosity control
- **OgiveNose**: implemented ogive nose as common component
- **common.py**: added uniform thickness function generator

- **Component**: check if patches have been generated when creating surfaces
- **Vehicle**: implemented to_stl method with component name iteration
- **Vehicle**: added component counts to vehicle repr
- refex vehicle working
- **geometry.py**: added Arc to namespace
- added legacy methods to components to transition
- **Vehicle**: started new implementation
- **transformations.py**: added standard transformations module
- **Component**: implemented reflect method
- **Component**: added methods for component processing
- **constants**: added constants component definitions
- updated fuselage and fin components
- **wing.py**: updated Wing component
- **component.py**: added abstract class for components
- **utils.py**: added csv to delaunay formatterr
- implemented swept fuselage component
- option to specify sweep axis
- **utils**: added surface perimeter utility class
- added ReFEX to hangar
- allow using default LE func for fin
- added falcon9 to hangar
- component-specific stl resolutions
- ability to specify revolve line for fuselage
- ability to stack fuse components
- added revolved surface class
- added htv to hangar
- added fuselage offset function
- updated docs
- added fin and fuselage constructor methods
- improved sensitivity naming in tri files
- improved formatting
- only write nominal geometry to file
- added distinction between building and writing STL to file
- added utility to add sensitivity data to .i.tri file
- first pass of geometric differentiation complete
- added close wing option

- added a template for using the polygon_formation module.
- Added new module polygon_formation.py and new component polygon.py
- added cube and sphere patch functions
- added tests badge

### 24.7.2 Fix

- **components**: removed stale curvature function arguments
- **Component**: fixed swept fuselage surface gen
- **SensitivitiyStudy**: updated to new structure
- **d21.py**: partially translated d21
- **Wing**: round bisect result for t_B1
- **uniform_thickness_function**: x and y coordinates
- **Vehicle**: transform using transformations from argument
- apply rounding to t_B1 bisect
- **Component**: added temporary means of component reflection
- **Fin**: save fin patches dict
- deleted outdated template file
- **Vehicle**: removed stale Vehicle methods
- flip end face for swept fuselage components
- match stl resolution for swept surfaces
- handle negative cross section locations
- **fuselage.py**: fixed return type hinting
- added offset func arg for fin components
- fuselage curvature implementation
- apply rudder angle before axial rotations
- rotate fuselage for cart3d
- verbosity output
- NumberOfComponents specification
- variable assignment
- fin negative volume error
- old import from idmoc
- tests link
- imports

### 24.7.3 Refactor

- **Vehicle**: changed ordering in transformations tuple
- **geometry.pu**: added Spline to geometry namespace
- merged component legacy methods into init
- split utils.py into geometry.py and utilities.py
- updated all imports to use gdtk namespace

### 24.7.4 Perf

- **SweptPatch**: significantly improved time to **call** swept patch

# PYTHON MODULE INDEX

## h

# INDEX

## Symbols



## A



## C



## D



## F



## G

## H



## M



## P



## R



## S



## T



## V



## W