## Contents

### 1 Examples

1.1 Airfoil ......................................................... 3
1.2 Transonic Airliner .......................................... 4
   1.2.1 Parameter Definitions ................................ 4
   1.2.2 Wing, Transonic Airliner ............................. 5
   1.2.3 Tailplane, Transonic Airliner ......................... 6
   1.2.4 Fuselage Transonic Airliner ......................... 8
   1.2.5 Wing-Body Fairing: .................................. 9
   1.2.6 Engine + Pylon ....................................... 9
   1.2.7 Miscellaneous operations .......................... 10
   1.2.8 Ipython Cell Renderer: ............................. 11
1.3 Topology model ............................................. 12
   1.3.1 Transonic Airliner Topology ......................... 12
   1.3.2 Predator UAV ......................................... 15
   1.3.3 Fairchild Republic A-10 Thunderbolt .............. 16
   1.3.4 Scaled Composites Proteus ......................... 19
1.4 References .................................................. 21

### 2 Installation

2.1 Conda packages ........................................... 23
2.2 Installation from source .................................. 23

### 3 occ_airconics API reference

3.1 airconics.base module .................................... 25
3.2 Primitives .................................................. 32
3.3 LiftingSurface ............................................. 35
3.4 Fuselage ................................................... 39
3.5 Engine ....................................................... 42
3.6 Topology .................................................... 43
3.7 AirCONICStools ............................................. 46
3.8 examples Subpackage ...................................... 53

### 4 Acknowledgements

4: 57

### 5 Indices and tables

5: 59

Python Module Index ........................................ 61
`occ_airconics` implements a scripted aircraft geometry package for Python, powered by Open CASCADE and PythonOCC.

While the majority of detailed aircraft design is performed by expert users of high-level Computer Aided Drawing (CAD) software, the bottom-up construction philosophy ‘aircraft geometry as computer code’ has recently amassed interest in vehicle concept design and multidisciplinary optimisation. Primary aims of `occ_airconics` are to provide one such implementation through generic aircraft primitives, such as the `Airfoil`, `LiftingSurface`, `Engine` and `Fuselage` classes, with a view that they will be used in configuration-level geometry parametrisation and optimisation. An example of this functionality lies in the built-in transonic airliner model, capable of producing a range of conventional geometries (see above) and a box-wing style aircraft.

One of the key advantages of `occ_airconics` is that it benefits from being built on the open-source full CAD kernel Open CASCADE available through PythonOCC, and therefore has access to an extensive and well-supported library of fast geometry manipulation tools including NURBS (Non-Uniform Rational B-Spline) curves and surfaces.

`occ_airconics` offers a fully cross platform and open source porting of core classes from the popular `AirCONICS` (Aircraft Configuration through Integrated Crossdisciplinary Scripting) plug-in for Rhinoceros 3D. Users of the original `AirCONICS` software should be aware however that the functionality of the original API has changed in `occ_airconics` to fit with the environments and ideologies of CPython and pythonOCC. In particular, the base classes `AirconicsShape` and `AirconicsCollection` are added (see API reference for details).

Installation of `occ_airconics` requires a recent version of PythonOCC, and is compatible with the latest version, 0.17, available from the conda package - see Installation for more information.

Refer to the examples and API reference for a demonstration of the usage of `occ_airconics`. Contributions are welcome, and developers should refer to the Open CASCADE and pythonOCC API documentation for guidelines on manipulation of underlying geometry kernel.

Enjoy `occ_airconics`!

Contents:
1.1 Airfoil

This example on building an airfoil NURBS curve with occ_airconics is included in the occ_airconics core Qt viewer examples.

First import the primitives module in which the Airfoil class is contained and the pythonocc-core Qt viewer:

```python
from airconics import primitives
# Visualisation with Python-OCC (ensure plot windows are set to qt)
from OCC.Display.SimpleGui import init_display
display, start_display, add_menu, add_function_to_menu = init_display()
```

Next, define the inputs to Airfoil class. In this example, we’ll use the SeligProfile type airfoil, leading edge point in origin, unit chord along x axis, no rotation around the x or y axes.

*Note: This class also supports construction of NACA 4 digit profiles using input keyword NACA4Profile. See Airfoil API reference.*

```python
LEPoint = [0., 0., 0.]
ChordLength = 1
Rotation = 0
Twist = 0
AirfoilSeligName = 'daell'

# Instantiate class to set up a generic airfoil with these basic parameters
Af = primitives.Airfoil(LEPoint, ChordLength, Rotation, Twist, 
                        SeligProfile=AirfoilSeligName)
```

Finally, display the curve and chord line

```python
display.DisplayShape(Af.Curve, update=True)
display.DisplayShape(Af.ChordLine, update=True)
start_display()
```
1.2 Transonic Airliner

In this example, the transonic airliner geometry example from the Rhinoceros Airconics plugin [1] is shown. All renderings are static images here, but represent interactive renderings when run as an IPython notebook available here. Interactive shapes can be viewed by clicking the shape hyperlinks however, as produced by the PythonOCC x3dom renderer.

For examples using the pythonocc-core Qt viewer, refer to the occ-airconics examples/core directory

```python
from airconics import liftingsurface, engine, fuselage_cml
import airconics.AirCONICSTools as act
from airconics.Addons.WebServer.TornadoWeb import TornadoWebRenderer
from IPython.display import display
```

1.2.1 Parameter Definitions

Parameters used here correspond to a geometry similar to that of the Boeing 787-8

```
Propulsion=1
EngineDia=2.9
FuselageScaling=[55.902, 55.902, 55.902]
WingScaleFactor=44.56
WingChordFactor=1.0
Topology=1
EngineSpanStation=0.31
EngineCtrBelowLE=0.3558
EngineCtrFwdOfLE=0.9837
Scarf_deg=3

# Derived Parameters
FuselageHeight = FuselageScaling[2]*0.105
```
1.2.2 Wing, Transonic Airliner

Formulation of lifting surfaces in occ_airconics (and AirCONICS) follows the suggestions in Sobester [2] in which geometry-attached curvilinear functionals are used instead of parameters for shape definition. That is, \( G(\bf{f}, \bf{X}) \), where

\[
\bf{f} = \left[ f_1(\bf{X}_1), f_2(\bf{X}_2), ... f_m(\bf{X}_m) \right],
\]

\[
\bf{X}_i = \left[ x_1^i, x_2^i, ..., x_n^i \right], \forall i = 1, ..., m
\]

as opposed to the conventional \( G(\bf{X}) \) formulation where the shape \( G \) changes in response to changes in design parameters \( \bf{X} \). The functions \( \bf{f}_i \) are defined by:

- \( \text{Sweep (\epsilon)} \)
- \( \text{Chord (\epsilon)} \)
- \( \text{Rotation (\epsilon)} \)
- \( \text{Twist (\epsilon)} \)
- \( \text{Airfoil (\epsilon)} \)

where \( \epsilon \) represents the spanwise coordinate ranging from 0 at the root of the wing to 1 at the tip. Output of the airfoil function uses the airconics.primitives.Airfoil class here, which fits a NURBS curve to airfoil coordinates.

The following code demonstrates construction of a wing using built in examples for a transonic airliner wing and tailplane (below).

```python
definition of lifting surfaces
```

1.2. Transonic Airliner
# Evaluate the root chord:
RootChord = Wing.RootChord

# Display
renderer = TornadoWebRenderer()
Wing.Display(renderer)
display(renderer)

1.2.3 Tailplane, Transonic Airliner

The same Lifting Surface class is used here to generate the fin and tailplane of the aircraft, using a different set of input functionals (also defined in airconics.examples).

```python
from OCC.gp import gp_Ax1, gp_Pnt, gp_Dir
from airconics.examples.tailplane_example_transonic_airliner import *

# Position of the apex of the fin
P = [36.98-0.49-0.02, 0.0, 2.395-0.141]
SegmentNo = 10
```
Fin = liftingsurface.LiftingSurface(P, mySweepAngleFunctionFin,  
    myDihedralFunctionFin,  
    myTwistFunctionFin,  
    myChordFunctionFin,  
    myAirfoilFunctionFin,  
    SegmentNo=SegmentNo,  
    ChordFactor=FinChordFact,  
    ScaleFactor=FinScaleFact)

# Create the rotation axis centered at the apex point in the x direction
RotAxis = gp_Ax1(gp_Pnt(*P), gp_Dir(1, 0, 0))

Fin.RotateComponents(RotAxis, 90)

# Position of the apex of the tailplane
P = [43, 0.000, 1.633+0.02]
SegmentNo = 100
ChordFactor = 1.01
ScaleFactor = 17.3

TP = liftingsurface.LiftingSurface(P, mySweepAngleFunctionTP,  
    myDihedralFunctionTP,  
    myTwistFunctionTP,  
    myChordFunctionTP,  
    myAirfoilFunctionTP,  
    SegmentNo=SegmentNo,  
    ChordFactor=TPChordFact,  
    ScaleFactor=TPScaleFact)

# Display
renderer = TornadoWebRenderer()
Fin.Display(renderer)
TP.Display(renderer)
display(renderer)
1.2.4 Fuselage Transonic Airliner

Fuselage shapes are created following the parameterisation used in Sobester [3]. That is, the outer mould line (OML) is split into a Nose, Central and Tail section, the length of which is described on input to Fuselage class as a percentage of the total length. Rib curves are then formed by fitting a NURBS curve to the intersection points of sectional planar cuts and the guide curves of the extremeties of the OML e.g. Port, top and bottom curves. The OML is fitted in occ_airconics using the Open CASCADE ThruSections loft.

```python
NoseLengthRatio=0.182
TailLengthRatio=0.293

Fus = fuselage_oml.Fuselage(NoseLengthRatio, TailLengthRatio,
                          Scaling=FuselageScaling,
                          NoseCoordinates=[0., 0., 0],
                          CylindricalMidSection=False,
                          Maxi_attempt=5)

# Display
renderer = TornadoWebRenderer()
Fus.Display(renderer)
display(renderer)
```

Network surface fit succesful on attempt 1
Interactive x3dom Fuselage

1.2.5 Wing-Body Fairing:

The wing-body fairing is here created as a simple ellipsoid shape around the root section of the wing.

_Note that this component will be displayed only in the final model._

```python
# WingBodyFairing: A simple ellipsoid:
from airconics.base import AirconicsShape
WTBFZ = RootChord*0.009 # 0.2
WTBFheight = 1.8*0.1212*RootChord # 2.7
WTBFwidth = 1.08*FuselageWidth
WTBFXCentre = WingApex[0] + RootChord/2.0 + RootChord*0.1297 # 23.8
WTBFlength = 1.167*RootChord # 26

WBF_shape = act.make_ellipsoid([WTBFXCentre, 0, WTBFZ], WTBFlength, WTBFwidth, WTBFheight)
WBF = AirconicsShape(components={'WBF': WBF_shape})
```

1.2.6 Engine + Pylon

First, obtain the wing section and chord at which the engine will be fitted, then fit then engine. The default inputs to the Engine class produce a turbofan engine with Nacelle similar to that of the RR Trent 1000 / GEnx and its pylon (currently a flat plate only).
from airconics import engine

EngineSection, Chord = act.CutSect(Wing['Surface'], EngineSpanStation)
CEP = Chord.EndPoint()
Centreloc = [CEP.X()-EngineCtrFwdOfLE*NacelleLength,
            CEP.Y(),
            CEP.Z()-EngineCtrBelowLE*NacelleLength]

eng = engine.Engine(Chord,
                    CentreLocation=Centreloc,
                    ScarfAngle=Scarf_deg,
                    HighlightRadius=EngineDia/2.0,
                    MeanNacelleLength=NacelleLength)

# Display
renderer = TornadoWebRenderer()
eng.Display(renderer)
display(renderer)

1.2.7 Miscellaneous operations

# Trim the inboard section of the main wing:
CutCirc = act.make_circle3pt([0,WTBFwidth/4.,-45], [0,WTBFwidth/4.,45], [90,WTBFwidth/4.,0])
CutCircDisk = act.PlanarSurf(CutCirc)
Wing['Surface'] = act.TrimShapebyPlane(Wing['Surface'], CutCircDisk)

# Mirror the main wing and tailplane using class methods:
Wing2 = Wing.MirrorComponents(plane='xz')
TP2 = TP.MirrorComponents(plane='xz')
eng2 = eng.MirrorComponents(plane='xz')

can work? True
error status: - Ok
Note: MirrorComponents currently mirrors only the shape components, other attributes will not be mirrored

Note: MirrorComponents currently mirrors only the shape components, other attributes will not be mirrored

Note: MirrorComponents currently mirrors only the shape components, other attributes will not be mirrored

1.2.8 Ipython Cell Renderer:

Now render the finished airliner model:

```python
from airconics.Addons.WebServer import TornadoWeb
renderer = TornadoWeb.TornadoWebRenderer()
# display all entities:
# Fuselage and wing-body fairing
Fus.Display(renderer)
WBF.Display(renderer)

# The Wings:
Wing.Display(renderer)
Wing2.Display(renderer)

# The Tailplane:
TP.Display(renderer)
TP2.Display(renderer)

# The Fin:
Fin.Display(renderer)

# The Engines:
eng.Display(renderer)
eng2.Display(renderer)

# Finally show the renderer
display(renderer)
```
1.3 Topology model

This is a work in progress towards a topologically flexible model based on the tree-type definition described in Sobester [1]. Note the geometry is not currently defined by the tree however, the tree is simply stored as a result of adding components - this is for demonstration only, and the process is yet to be automated.

1.3.1 Transonic Airliner Topology

First, we’ll try to add the previously created transonic airliner components to a topology, including the number of descendant nodes that will be attached to each, and then display the resulting tree graph. The LISP representation of this tree could be described as

Fuselage(Fin, Mirror[ (TailPlane, Wing(Engine))]

where opening brackets indicate that the following component is to be ‘attached’ to the preceeding shape. Using the shorthand described in [1], this is equivalent to \( \langle E(L, \|L, L(P)) \rangle \), where \( \langle E \rangle \) is an enclosure/fuselage object, \( \langle L \rangle \) is a lifting surface, \( \| \) is a mirror plane and \( \langle P \rangle \) is a propulsion unit [1]. Study the Airliner model above and recursively work through the components, starting from the fuselage, and think about the sub-components are attached to them to assert that this is true.

The \( \langle xz \rangle \) mirror plane is included in this representation, between central objects (Fuselage, Fin) and the mirrored objects (Tail Plane, Wing, Engine). Notice that the dotted line box surrounds the entities that will be mirrored when Topology.Build() is called.

```
from airconics import Topology
from IPython.display import Image
import pydot
```
```python
topo_renderer = TornadoWebRenderer()

topo = Topology()

# Note: no checks are done on the validity of the tree yet,
topo.AddPart(Fus, 'Fuselage', 3)
topo.AddPart(Fin, 'Fin', 0)

# Need to add a mirror plane here, arity zero
from OCC.gp import gp_Ax2, gp_Dir, gp_Pnt
xz_pln = gp_Ax2(gp_Pnt(0, 0, 0), gp_Dir(0, 1, 0))
topo.AddPart(xz_pln, 'Mirror', 0)

# These are the mirrored entities, with their arities
topo.AddPart(TP, 'Tail Plane', 0)
topo.AddPart(Wing, 'Wing', 1)
topo.AddPart(eng, 'Engine', 0)

# print the Topology (resembles a LISP tree)
print(topo)

# Create the graph with pydot
graph = pydot.graph_from_dot_data(topo.export_graphviz())
Image(graph.create_png())

Skipping geometry construction for Topology
E(L, |L, L(P))
```

# This line will mirror geometry 'under' (added after) the mirror plane
topo.Build()

topo.Display(topo_renderer)
display(topo_renderer)

1.3. Topology model
Interactive x3dom Airliner

Let’s try some further tests to the topology class representation using some other examples. For now, these are empty geometries, and inputs to the Fuselage, LiftingSurface and Engine classes are not yet included in the Topology tree.
1.3.2 Predator UAV

US Air Force

# Setup
# Create mock components, without generating any geometry
fus = Fuselage(construct_geometry=False)
engine = Engine(construct_geometry=False)
fin = LiftingSurface(construct_geometry=False)
mirror_pln = gp_Ax2()
wing = LiftingSurface(construct_geometry=False)
Vfin = LiftingSurface(construct_geometry=False)

# For now we must manually add parts and affinities
topo = Topology()
topo.AddPart(fus, 'Fuselage', 4)
topo.AddPart(engine, 'engine', 0)
topo.AddPart(fin, 'fin', 0)
topo.AddPart(mirror_pln, 'Mirror', 0)
topo.AddPart(wing, 'wing', 0)
topo.AddPart(Vfin, 'V-Fin', 0)

print(topo)

graph = pydot.graph_from_dot_data(topo.export_graphviz())
Image(graph.create_png())

Skipping geometry construction for Fuselage
No HChord specified to fit engine to: creating default

1.3. Topology model
1.3.3 Fairchild Republic A-10 Thunderbolt

Photo source: Airman Magazine 1999
# Setup

# Create mock components, without generating any geometry
fus = Fuselage(construct_geometry=False)
mirror_pln = gp_Ax2()
engine = Engine(construct_geometry=False)
wing = LiftingSurface(construct_geometry=False)
tailplane = LiftingSurface(construct_geometry=False)
tail_fin = LiftingSurface(construct_geometry=False)

topo = Topology()
topo.AddPart(fus, 'Fuselage', 3)
topo.AddPart(mirror_pln, 'mirror', 0)
topo.AddPart(engine, 'powerplant', 0)
topo.AddPart(tailplane, 'Tailplane', 1)
topo.AddPart(tail_fin, "Tail fin", 0)
topo.AddPart(wing, "wing", 0)

print(topo)

graph = pydot.graph_from_dot_data(topo.export_graphviz())
Image(graph.create_png())

Skipping geometry construction for Fuselage
No HChord specified to fit engine to: creating default
Skipping geometry construction for Engine
Lifting Surface functional parameters not defined:
  Initialising without geometry construction
Skipping geometry construction for LiftingSurface
Lifting Surface functional parameters not defined:
  Initialising without geometry construction
Skipping geometry construction for LiftingSurface
Lifting Surface functional parameters not defined:
  Initialising without geometry construction
Skipping geometry construction for LiftingSurface
Skipping geometry construction for Topology
E(|P, L(L), L)
1.3.4 Scaled Composites Proteus

Photo source: NASA

```python
# Setup
# Create mock components, without generating any geometry
fus = Fuselage(construct_geometry=False)
mirror_pln = gp_Ax2()
engine = Engine(construct_geometry=False)
wing_in = LiftingSurface(construct_geometry=False)
tailplane = LiftingSurface(construct_geometry=False)
pod = Fuselage(construct_geometry=False)
finup = LiftingSurface(construct_geometry=False)
findown = LiftingSurface(construct_geometry=False)
wing_out = LiftingSurface(construct_geometry=False)

topo = Topology()
topo.AddPart(fus, 'Fuselage', 3)
topo.AddPart(mirror_pln, 'mirror', 0)
topo.AddPart(engine, 'powerplant', 0)
topo.AddPart(wing, "wing", 0)
topo.AddPart(wing_in, "TP/inbbd wing", 1)
topo.AddPart(pod, 'Pod/tail boom', 3)
```
```python
topo.AddPart(wing_out, "outbd wing", 0)
topo.AddPart(finup, "Fin (up)", 0)
topo.AddPart(findown, "Fin (down)", 0)

    for node in topo._Tree:
        print(node)

    graph = pydot.graph_from_dot_data(topo.export_graphviz())
    Image(graph.create_png())
```

Skipping geometry construction for Fuselage
No HChord specified to fit engine to: creating default
Skipping geometry construction for Engine
Lifting Surface functional parameters not defined:
Initialising without geometry construction
Skipping geometry construction for LiftingSurface
Lifting Surface functional parameters not defined:
Initialising without geometry construction
Skipping geometry construction for LiftingSurface
Skipping geometry construction for Fuselage
Lifting Surface functional parameters not defined:
Initialising without geometry construction
Skipping geometry construction for LiftingSurface
Lifting Surface functional parameters not defined:
Initialising without geometry construction
Skipping geometry construction for LiftingSurface
Skipping geometry construction for LiftingSurface
Skipping geometry construction for Topology
(Fuselage, E, 3)
(mirror, I, 0)
(powerplant, P, 0)
(wing, L, 0)
(TP/inbbd wing, L, 1)
(Pod/tail boom, E, 3)
(outbd wing, L, 0)
(Fin (up), L, 0)
(Fin (down), L, 0)
1.4 References


occ_airconics accesses the powerful Open CASCADE geometry kernel through the PythonOCC package. It is possible to build pythonocc-core from source by following their installation instructions, however a convenient and recommended alternative is to use the prebuilt conda packages suitable for win32/win64/osx64/linux64 users.

Note that occ_airconics is not currently available through PyPI.

## 2.1 Conda packages

pythonocc-core is listed as a dependency of occ_airconics, therefore users should simply add the appropriate conda channels to their ~/.condarc file:

```bash
conda config --add channels dlr-sc # the pythonocc-core channel
conda config --add channels prchambers # the occ_airconics channel
```

Or do this manually by editing their ~/.condarc contents, e.g.:

```bash
channels:
  - https://conda.anaconda.org/dlr-sc
  - https://conda.anaconda.org/prchambers
  - defaults
```

Then install occ_airconics via

```bash
conda install occ_airconics
```

And that’s it! pythonocc-core will be installed automatically.

## 2.2 Installation from source

Obtain and build a copy of pythonocc-core from GitHub following their instructions.

Then clone occ_airconics from GitHub with:

```bash
git clone https://github.com/p-chambers/occ_airconics
```

And install with

```bash
    cd occ_airconics
    python setup.py install
```
Or

`pip install occ_airconics`

Developers should also add the develop flag, i.e.

`python setup.py install develop`
3.1 airconics.base module

Base classes used by OCC_Airconics

Container classes (AirconicsBase, AirconicsShape, AirconicsCollection) which behaves like a dictionary of sub component shapes/parts with some extended functionality.

Created on Mon Apr 18 10:26:22 2016
@author: pchambers

class airconics.base.AirconicsBase(*args, **kwargs)
   Bases: _abcoll.MutableMapping, object

   Base container class from which other base classes are derived from. This is a an abstract base class and should not be used directly by users.

   Notes

   When properly defined in inherited functions, this class should behave like a dictionary.
   As this class inherits from MutableMapping, any class inheriting from AirconicsBase must also define the abstract methods of Mutable mapping, i.e. __setitem__, __getitem__, __len__, __iter__, __delitem__

   Methods

   Build(*args, **kwargs)
   Display(*args, **kwargs)
   Write(*args, **kwargs)
   clear() -> None. Remove all items from D.
   get(k[, d]) -> D[k] if k in D, otherwise d.
   items() -> list of D's (key, value) items.
   iteritems() -> an iterator over the (key, value) pairs.
   iterkeys() -> an iterator over the keys of D.
   itervalues() -> an iterator over the values of D.
   keys() -> list of D's keys.
   pop(k[, d]) -> v, d is returned if given, otherwise KeyError is raised.
Table 3.1 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>popitem()</td>
<td>as a 2-tuple; but raise KeyError if D is empty.</td>
</tr>
<tr>
<td>setdefault((k,[d]) -&gt; D.get(k,d), ...)</td>
<td>If E present and has a .keys() method, does: for k in E: D[k] = E[k]</td>
</tr>
<tr>
<td>update((E,...))</td>
<td></td>
</tr>
<tr>
<td>values()</td>
<td></td>
</tr>
</tbody>
</table>

| Build(*args,**kwargs)             |                                                                            |
| Display(*args,**kwargs)          |                                                                            |
| Write(*args,**kwargs)            |                                                                            |

class airconics.base.AirconicsCollection: $\textbf{parameters}$ parts: dictionary

<table>
<thead>
<tr>
<th>Notes</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Derived classes should call the AirconicsCollection init with</td>
<td>super(DerivedClass, self).<strong>init</strong>(self, *args, **kwargs)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>_Parts</td>
<td>Mapping of name(string):component(AirconicsShape) pairs. Note that this</td>
</tr>
<tr>
<td></td>
<td>should not be interacted with directly, and instead users should use assignment or the AddPart method: :Example: &gt;&gt;&gt; a = AirconicsCollection() &gt;&gt;&gt; a['name'] = part &gt;&gt;&gt; #OR: a.AddPart('name', part) This also supports mapping of attributes to parts, i.e: :Example: &gt;&gt;&gt; a['name'] == a._Parts.name # returns True</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AddPart(part[, name])</td>
<td>Adds a component to self</td>
</tr>
<tr>
<td>Build()</td>
<td>Does nothing for AirconicsCollection.</td>
</tr>
<tr>
<td>Display(context[, material, color])</td>
<td>Displays all Parts of the engine to input context</td>
</tr>
<tr>
<td>Write(filename[, single_export])</td>
<td>Writes the Parts contained in this instance to file specified by filename.</td>
</tr>
<tr>
<td>clear()</td>
<td>None. Remove all items from D.</td>
</tr>
<tr>
<td>get((k,[d]) -&gt; D[k] if k in D, ...)</td>
<td></td>
</tr>
<tr>
<td>items()</td>
<td>list of D’s (key, value) pairs, ...</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>iteritems()</code></td>
<td>an iterator over the (key, ...)</td>
</tr>
<tr>
<td><code>iterkeys()</code></td>
<td>an iterator over the keys of D</td>
</tr>
<tr>
<td><code>itervalues()</code></td>
<td></td>
</tr>
<tr>
<td><code>keys()</code></td>
<td>list of D’s keys</td>
</tr>
<tr>
<td><code>pop(k[,d])</code></td>
<td>If key is not found, d is returned if given, otherwise KeyError is raised.</td>
</tr>
<tr>
<td><code>popitem()</code></td>
<td>as a 2-tuple; but raise KeyError if D is empty.</td>
</tr>
<tr>
<td><code>setdefault()</code></td>
<td>(k[,d]) -&gt; D.get(k,d), ...</td>
</tr>
<tr>
<td><code>update([E, ...])</code></td>
<td>If E present and has a .keys() method, does: for k in E: D[k] = E[k]</td>
</tr>
<tr>
<td><code>values()</code></td>
<td>list of D’s values</td>
</tr>
</tbody>
</table>

### AddPart

`AddPart(part, name=None)`

Adds a component to self

#### Parameters

- `part`: TopoDS_Shape
- `name`: string

### Build

`Build()`

Does nothing for AirconicsCollection.

This method allows AirconicsCollection to be instantiated alone, as `Build` is called in the `__init__`. ‘Build’ Should be redefined by all derived classes.

### Notes

- If Class.Build is not redefined in a derived class, confusion may arise as no geometry will result from passing construct_geometry=True

### Display

`Display(context, material=<Mock id='140564314113040'>, color=None)`

Displays all Parts of the engine to input context

#### Parameters

- `context`: OCC.Display.OCCViewer.Viewer3d or WebRenderer
- `material`: OCC.Graphic3d_NOM_* type

The display context - should have a Display or DisplayShape method

The material for display: note some renderers do not allow this

### Write

`Write(filename, single_export=True)`

Writes the Parts contained in this instance to file specified by filename.

One file is produced, unless single_export is False when one file is written for each Part.

#### Parameters

- `filename` [string] the BASE.ext name of the file e.g. ‘airliner.stp’. Note the part name will be prepended to the base name of each output file
- `single_export` [bool] returns a single output file if true, otherwise writes one file per part

#### Returns

- `status`: list

The flattened list of error codes returned by writing each part

### See also:

- AirconicsBase

---

3.1. airconics.base module
class `AirconicsShape`:

```python
class AirconicsShape(components={}, construct_geometry=False, **kwargs):
    Bases: airconics.base.AirconicsBase
```

Base class from which airconics parts will be made.

AirconicsShapes represent a ‘part’ of an aircraft e.g. the engine, which consists of a group of logically shape ‘components’ but with no relative or relational contact information: class methods are intended to manipulate the part as a whole.

This is intended as a base class, but can be used as a simple amorphic collection of shape components.

Example

```python
>>> shape = Airconics()
>>> shape['wing'] = wing  # OR
>>> shape.AddComponent(wing, 'WingSurface')
```

Parameters

- **components**: dictionary of components to be added as attributes
  - To add attributes directly. Values must be OCC.TopoDS.TopoDS_Shape
- **construct_geometry**: bool
  - If true, Build method will be called on construction. Defaults to False for AirconicsShape, as the Build method only prints. Derived classes should pass construct_geometry=True if a Build method should be called on construction
- **kwargs**: All other keyword arguments will be added as an attribute to the resulting class calling `super(subclass, self).__init__`

See also:

- `AirconicsCollection`

Notes

Derived classes should call the `AirconicsCollection` init with

```python
super(DerivedClass, self).__init__(self, *args, **kwargs)
```

Attributes

- `_Components`: (Airconics Container) Mapping of name(string):component(TopoDS_Shape) pairs. Note that this should not be interacted with directly, and instead users should use assignment or the AddComponent method:

  ```python
  #Example: >>> a = AirconicsShape() >>> a[‘name’] = shape >>>
  #OR: a.AddComponent(‘name’, shape) This also supports mapping of attributes to parts, i.e:
  #Example: >>> a[‘name’] == a._Components.name # returns True
  ```

Methods
### AddComponent(component[, name])

Add a component to self.

**Parameters**
- **component**: TopoDS_Shape
- **name**: string (default=None)

### Build()

Does nothing for AirconicsShape.

### Display(context[, material, color])

Displays all components of this instance to input context.

**Parameters**
- **context**: OCC.Display.OCCViewer.Viewer3d or WebRenderer
- **material**: OCC.Graphic3d_NOM_* type (default=ALUMINUM)
- **color**: None

### DisplayBBox(display[, single])

Displays the bounding box on input display.

### Extents([tol, as_vec])

Returns the extents of the bounding box encapsulating all shapes in self.

### MirrorComponents([plane, axe2])

Returns a mirrored version of this airconics shape.

### PrintComponents()

Lists the names of components in self.

### RemoveComponent(name)

Removes a named component from self.

### RotateComponents(ax, deg)

Rotation of each component in self._Components around ax by deg.

### ScaleComponents_Uniformal(factor[, origin])

General scaling and translation of components in self.

### TransformComponents_Nonuniformal(scaling, vec)

General scaling and translation of components in self.

### TranslateComponents(vec)

Apply translation by vec to each component in self.

### Write(filename[, single_export])

Writes the Components in this Airconics shape to filename using file format specified in extension of filename.

---

**Notes**

- If Class.Build is not redefined in a derived class, confusion may arise as no geometry will result from passing construct_geometry=True

**Display**

**Parameters**
- **context**: OCC.Display.OCCViewer.Viewer3d or WebRenderer
- **material**: OCC.Graphic3d_NOM_* type (default=ALUMINUM)
- **color**: None

The material for display: note some renderers do not allow this.
**DisplayBBox** *(display, single=True)*
Displays the bounding box on input display.

**Parameters**
- **display**: OCC.Display.OCCViewer.Viewer3d
- **single**: bool (default True)

Note that this function only currently works with the OCC core viewer as the bounding box is an AIS_Shape handle.

If false, separate bounding boxes will be drawn for each component.

**Extents** *(tol=1e-06, as_vec=False)*
Returns the extents of the bounding box encapsulating all shapes in self.__Components__.

**Parameters**
- **tol**: scalar (default 1e-6)
  - tolerance of the triangulation used in the bounding box extents
- **as_vec**: bool (default True)
  - Returns two OCC.gp.gp_Vec objects if True

**Returns**
- **extents**: tuple of scalar or OCC.gp.gp_Vec
  - Type depends on input ‘as_vec’. If as_vec is False, this returns a tuple xmin, ymin, zmin, xmax, ymax, zmax; otherwise, the min and max vectors will be returned as OCC types.

**MirrorComponents** *(plane='xz', axe2=None)*
Returns a mirrored version of this airconics shape.

**Parameters**
- **plane**: string (default='xz')
  - The plane in which to mirror components
- **axe2**: OCC.gp.gp_Ax2
  - The axes through which to mirror (overwrites input ‘plane’)

**Returns**
- **mirrored**: AirconicsShape
  - the mirrored shape

**Notes**

Due to problem with swig and deepcopy, the mirrored object is the base class ‘AirconicsShape”, not the original type. This is will remove other subclass-derived attributes and methods.

It is also expected that the remaining attributes and methods will not be required or meaningful after mirroring, however this behaviour may change in future versions.

**PrintComponents** ()
Lists the names of components in self.

**RemoveComponent** *(name)*
Removes a named component from self.

**Parameters**
- **name**: string

**RotateComponents** *(ax, deg)*
Rotation of each component in self._Components around ax by angle deg.
**Parameters** 

- **`ax`**: OCC.gp.gp_Ax1
  The axis of rotation

- **`deg`**: scalar
  Rotation in degrees

**ScaleComponents_Uinformal** *(`factor`, `origin`)*

General scaling and translation of components in self (applies act.transform_nonuniform)

**Parameters**

- **`origin`**: gp_Pnt
  The origin of the scaling operation

- **`factor`**: scalar
  The scaling factor to apply in x,y,z

**TransformComponents_Nonuniformal** *(`scaling`, `vec`)*

General scaling and translation of components in self (applies act.transform_nonuniform)

**Parameters**

- **`scaling`**: list or array, length 3
  [x, y, z] scaling factors

- **`vec`**: List of x,y,z or gp_Vec
  the translation vector (default is [0,0,0])

**TranslateComponents** *(`vec`)*

Apply translation by vec to each component in self

**Parameters**

- **`vec`**: OCC.gp.gp_vec
  vector through which components will be translated

**Write** *(`filename`, `single_export=True`)*

 Writes the Components in this Airconics shape to `filename` using file format specified in extension of `filename`. Currently stl only (TODO: step, iges)

**Parameters**

- **`filename`**: string
  the BASE.ext name of the file e.g. ‘airliner.stp’. Note the Component name will be prepended to the base name of each output file

- **`single_export`**: bool
  Writes a single output file if true, otherwise writes one file per component

**Returns**

- **`status`**: list of int
  error status of the file output of EACH component

**Notes**

File format is extracted from `filename`.

stl file write will prepend `filename` onto the Component name to be written to file (cannot write multiple files)
3.2 Primitives

class airconics.primitives.Airfoil(LeadingEdgePoint=[0.0, 0.0, 0.0], ChordLength=1, Rotation=0, Twist=0, SeligProfile=None, Naca4Profile=None, Naca5Profile=None, CRMProfile=None, CRM_Epsilon=0.0, InterpProfile=None, Epsilon=0, Af1=None, Af2=None, Eps1=0, Eps2=1, EnforceSharpTE=False)

Bases: object

Class for defining a range of spline-fitted airfoil curves

Parameters

LeadingEdgePoint : array of float (3)
    (x, y, z) origin of the airfoil LE

ChordLength : scalar
    Length of the airfoil chord

Rotation : scalar
    Angle (deg) at which the base airfoil is inclined (angle of attack, rotation around y axis)

Twist : scalar
    Angle (deg) at which the base airfoil is twisted (dihedral, rotation around x axis)

SeligProfile : string
    Name of the Selig airfoil: see http://m-selig.ae.illinois.edu/ads/coord_database.html

NACA4Profile : string
    Name of the airfoil in NACA 4 format

NACA5Profile : string
    Name of the airfoil in NACA 5 format. TODO: NACA5 profile not yet implemented

CRM_Profile : bool
    If true, airfoil profile will be interpolated from Common Research Model (CRM). Must also declare ‘CRMEpsilon’ variable.

CRM_Epsilon : float
    Spanwise fraction between 0 and 1 to interpolate profile from CRM

InterpProfile : bool
    If True, a set of points between Af1 and Af2 will be interpolated for BSpline curve fitting (see AddInterp2)

Eps : scalar
    Spanwise coordinate between Eps1 and Eps2 (used if InterpProfile is True)

Af1 : airconics.Airfoil
    The Airfoil located at spanwise location Eps1. Af1.x and Af1.z. (used if InterpProfile is True)

Af2 : airconics.Airfoil
    The Airfoil located at spanwise location Eps2. (used if InterpProfile is True)

Eps1 : scalar
Spanwise coordinate location of Airfoil Af1. Expected to range from 0 (root of a lifting surface) to 1 (tip of a lifting surface) (used if InterpProfile is True)

\textbf{Eps2} : scalar

Spanwise coordinate location of Airfoil Af2. Expected to range from 0 (root of a lifting surface) to 1 (tip of a lifting surface), and also expected to be greater than Eps1 (used if InterpProfile is True)

\textbf{EnforceSharpTE} : bool

Enforces sharp trailing edge (NACA airfoils only)

\section*{Notes}

\begin{itemize}
  \item NACA5 profiles are not yet supported in OCC_AirCONICS.
  \item Preference is that users allow the class constructor to handle building the Airfoil i.e. pass all physical definitions as class arguments.
  \item Although the physical attributes can changed i.e. rotation, twist, ChordLength, LeadingEdgePoint etc., it is the users responsibility to rebuild the Airfoil with the ‘Add***Airfoil’ afterwards
\end{itemize}

\section*{Attributes}

\begin{tabular}{|l|l|}
  \hline
  points & (array of scalar, shape (N, 2)) The x-z coordinates of points on the airfoils surface \\
  \hline
  Curve - OCC.Geom.Handle_Geom_BsplineCurve & The generated airfoil spline \\
  \hline
\end{tabular}

\section*{Methods}

\begin{itemize}
  \item AddAirfoilFromSeligFile \newline
  \hspace{1cm} AddCRMLinear \newline
  \hspace{1cm} AddLinear2 \newline
  \hspace{1cm} AddNACA4
\end{itemize}

\textbf{AddAirfoilFromSeligFile} \texttt{(SeligProfile, Smoothing=1)}

Adds an airfoil generated by fitting a NURBS curve to a set of points whose coordinates are given in a Selig formatted file

\begin{itemize}
  \item \textbf{Parameters} SeligProfile : string
    \hspace{1cm} base selig airfoil name e.g. ‘b707a’.
  \item Smoothing : int (default=1)
    \hspace{1cm} TODO: Airfoil curve smoothing
  \item \textbf{Returns} None
\end{itemize}

\section*{Notes}

See Selig database online for other available base names
AddCRMLinear \((CRM\_Epsilon, Smoothing=1)\)
Linearly interpolate airfoil curve from CRM

**Parameters**

CRM\_Epsilon : scalar
Spanwise coordinate at which to sample the CRM airfoil database (range between 0 and 1)

Smoothing : int
TODO: Airfoil curve smoothing

**Returns**
None

AddLinear2 \((Eps, Af1, Af2, Eps1=0, Eps2=1)\)
Interpolates the bspline control points fitted between two other Airfoil objects.
Interpolates the x and z values of an Airfoil at spanwise location Eps between Af1 (at Eps1) and Af2 (at Eps2). The BSpline Curve is then fitted to the resulting points with Airfoil._fitAirfoiltoPoints, and transformed to the orientation specified in self.Rotation, self.Twist, self.ChordLength and self.LEPoint via the _TransformAirfoil function.

**Parameters**

Eps : scalar
Spanwise coordinate between Eps1 and Eps2

Af1 : airconics.Airfoil
The Airfoil located at spanwise location Eps1. Af1.x and Af1.z

Af2 : airconics.Airfoil
The Airfoil located at spanwise location Eps2.

Eps1 : scalar
Spanwise coordinate location of Airfoil Af1. Expected to range from 0 (root of a lifting surface) to 1 (tip of a lifting surface)

Eps2 : scalar
Spanwise coordinate location of Airfoil Af2. Expected to range from 0 (root of a lifting surface) to 1 (tip of a lifting surface), and also expected to be greater than Eps1

AddNACA4 \((Naca4Profile, Smoothing=1)\)
Adds a NACA 4 digit airfoil to the current document

**Parameters**

Naca4Profile : string
Naca 4 profile identifier. Should be length 4 string, however also accepts negative camber i.e. ‘-5310’ gives a flipped camber airfoil (primarily used for box wing)

Smoothing - int
TODO: fair airfoil curve

**Returns**
None
3.3 LiftingSurface

class airconics.liftingsurface.LiftingSurface(ApexPoint=<Mock name='mock()' id='140564123154768'>, SweepFunct=False, DihedralFunct=False, TwistFunct=False, ChordFunct=False, AirfoilFunct=False, ChordFactor=1, ScaleFactor=1, OptimizeChordScale=0, LooseSurf=1, SegmentNo=11, TipRequired=False, max_degree=8, continuity=<Mock id='140564123154640'>, construct_geometry=True)

Bases: airconics.base.AirconicsShape

Airconics class for defining lifting surface shapes

Parameters

ApexPoint - array, length 3

Foremost point of the wing (x direction). Updating will rebuild the geometry.

SweepFunct - function

function defining the leading edge sweep vs epsilon spanwise variable coordinate between 0 and 1 (curvilinear attached coordinates). Updating will rebuild the geometry.

DihedralFunct - function

function defining the leading edge dihedral vs epsilon spanwise variable coordinate between 0 and 1 (curvilinear attached). Updating will rebuild the geometry.

TwistFunct - function

function defining the sectional twist vs epsilon spanwise variable coordinate between 0 and 1 (curvilinear attached). Updating will rebuild the geometry.

ChordFunct - function

function defining the leading edge chord vs epsilon spanwise variable coordinate between 0 and 1 (curvilinear attached) Updating will rebuild the geometry.

AirfoilFunct - function

function defining the sectional Airfoil (see primitives.Airfoil) vs epsilon spanwise variable coordinate between 0 and 1 (curvilinear attached). Updating will rebuild the geometry.

ChordFactor - int (default = 1)

Scaling factor applied in chordwise direction. Updating will rebuild the geometry.

ScaleFactor - int (default = 1)

Scaling factor applied in all directions (uniform). Updating will rebuild the geometry.

OptimizeChordScale - int or bool (default = 0)

TODO: Not yet used.

LooseSurf - (default = 1)

TODO:

NSegments - int (default = 11)
Number of segments to sample the wing defined by input functions. Updating will rebuild the geometry.

**TipRequired** - bool (default = False)

TODO: Not yet used adds the wing tip face to components if true

**max_degree** - (default = 8)

maximum degree of the fitted NURBS surface

**continuity** - OCC.GeomAbs.GeomAbs_XX Type

the order of continuity i.e. C^0, C^1, C^2... would be GeomAbs_C0, GeomAbs_C1, GeomAbs_C2 ...

**construct Geometry** : bool

If true, Build method will be called on construction

See also:

airconics.primitives.Airfoil, airconics.examples.wing_example_transonic_airliner

Notes

- Output surface is stored in self['Surface']
- 
- See airconics.examples.wing_example_transonic_airliner for example input functions

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>self['Surface']</td>
<td>TopoDS_Shape</td>
<td>The generated lifting surface</td>
</tr>
<tr>
<td>Sections</td>
<td>list of airconics.primitives.Airfoil objects</td>
<td>The rib curves from which the main surface is lofted. Updating any of the spanwise functions (ChordFunct, TwistFunct, etc...), SpanFactor, ChordFactor Raises an error if attempting to write over it manually.</td>
</tr>
<tr>
<td>RootChord</td>
<td>Scalar</td>
<td>The length of the Root Chord. Updated by GenerateLiftingSurface</td>
</tr>
<tr>
<td>AR</td>
<td>Scalar</td>
<td>The Aspect ratio of the Lifting Surface. Updated on call to Build</td>
</tr>
<tr>
<td>LSP_area</td>
<td>Scalar</td>
<td>the projected area of the lifting surface. Updated on call to Build</td>
</tr>
<tr>
<td>SA</td>
<td>Scalar</td>
<td>the wetted area of the lifting surface. Updated on call to Build</td>
</tr>
<tr>
<td>ActualSemiSpan</td>
<td>Scalar</td>
<td>Calculated semi span of the lifting surface. Updated on call to Build</td>
</tr>
</tbody>
</table>

Methods

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddComponent</td>
</tr>
<tr>
<td>Build</td>
</tr>
<tr>
<td>CalculateAspectRatio</td>
</tr>
<tr>
<td>CalculateProjectedArea</td>
</tr>
<tr>
<td>CalculateSemiSpan</td>
</tr>
<tr>
<td>CreateConstructionGeometry</td>
</tr>
<tr>
<td>Display</td>
</tr>
</tbody>
</table>

Continued on next page
### AirfoilFunct

#### ApexPoint

**Build()**

Builds the section curves and lifting surface using the current

Uses the current ChordFactor, ScaleFactor, NSegments, ApexPoint, and all spanwise variation functions (e.g. ChordFunct) defined in self to produce a surface

**See also:**

airconics.examples.wing_example_transonic_airliner

**Notes**

Called on initialisation of a lifting surface class.

**Example**

```python
globals()['Wing'] = liftingsurface.LiftingSurface(P,
mySweepAngleFunction,
myDihedralFunction,
myTwistFunction,
myChordFunction,
myAirfoilFunction)

globals()['Surface'] = Wing['Surface']
```
CalculateAspectRatio()
Calculates and returns the aspect ratio of this lifting surface

Uses information about the wings projected area and the current bounding box. If the project area (LSP_Area) is zero, this will be calculated.

Returns AR : Scalar
Aspect ratio, calculated by $b^2 / A$, where $b$ is the semi span and $A$ is the project area of this lifting surface

See also:

CalculateProjectedArea()
Calculates the projected area of the current lifting surface

From Airconics documentation: In some cases the projected section cannot all be lofted in one go (it happens when parts of the wing fold back onto themselves), so we loft them section by section and compute the area as a sum.

CalculateSemiSpan()
Calculates and returns the span of this lifting surface.

Uses the OCC bounding box algorithm.

Returns ActualSemiSpan : Scalar

ChordFactor
ChordFunct

CreateConstructionGeometry()
Creates the plane and vector used for projecting wetted area

DihedralFunct

Fit_BlendedTipDevice(rootchord_norm, spanfraction=0.1, cant=40, transition=0.1, sweep=40, taper=0.7)
Fits a blended wing tip device [1],

Parameters rootchord_norm : scalar
The root chord of the straight part of the winglet, normalised by the tip chord of this lifting surface.

spanfraction : scalar
span of the winglet normalised by the span of the main wing

cant : scalar (default 40)
Angle (deg) of the wing tip from vertical

transition : scalar
The percentage along the span at which the transition to a straight segment is located

sweep : scalar
Sweep angle of the wing tip (uniform along the straight section)

taper : scalar
ratio of the tip chord to the root chord of the straight segment
References


GenerateLeadingEdge()
Epsilon coordinate attached to leading edge defines sweep Returns airfoil leading edge points

GenerateLiftingSurface()
Builds a lifting surface (wing, tailplane, etc.) with the Chord and Scale factors, and section list defined in self.
This function should be called after GenerateSectionCurves. Note that both operations are performed with ‘Build’, which should be used preferentially

Returns None

Notes

Add a (‘Surface’: Shape) key value pair to self.

GenerateSectionCurves()
Generates the loft section curves based on the current functional parameters and ChordFactor of the object.
Uses self._AirfoilFunct, _ChordFunct etc. and other attributes to update the content of self._Sections.

Returns None

3.4 Fuselage

class airconics.fuselage_oml.Fuselage(NoseLengthRatio=0.182, TailLengthRatio=0.293, Scaling=[55.902, 55.902, 55.902], NoseCoordinates=[0.0, 0.0, 0], CylindricalMidSection=False, SimplificationReqd=False, Maxi_attempt=5, construct_geometry=True)

Bases: airconics.base.AirconicsShape

AirCONICS Fuselage class: builds a parameterised instance of an aircraft fuselage

Parameters NoseLengthRatio : Scalar
The fraction of nose to fuselage length (default 0.182)

TailLengthRatio : Scalar
The fraction of tail to fuselage length (default 0.293)

Scaling : array, length 3
(x, y, z) scaling factor

NoseCoordinates : array of float
Location of nose apex

**CylindricalMidSection** : bool

If True, fuselage will have a cylindrical midsection

**SimplificationReqd** : bool

TODO

**MaxFittingAttempts** : integer

Maximum number of times to attempt to fit surface to guide curves

**construct_geometry** : bool

If true, Build method will be called on construction

**Notes**

Geometry building is done on initialisation of a Fuselage instance. It is therefore not expected that users will do this through the BuildFuselageOML function

**Methods**

```
AddComponent
AirlinerFuselagePlanView
AirlinerFuselageSideView
Build
BuildFuselageOML
CockpitWindowContours
Display
DisplayBBox
Extents
FuselageLongitudinalGuideCurves
MakeWindow
MirrorComponents
PrintComponents
RemoveComponent
RotateComponents
ScaleComponents_Uniformal
TransformComponents_Nonuniformal
TransformOML
TranslateComponents
WindowContour
Write
clear
get
items
iteritems
iterkeys
itervalues
keys
```
AirlinerFuselagePlanView\((NoseLengthRatio, TailLengthRatio)\)
   Internal function. Defines the control polygons of the fuselage in side view

AirlinerFuselageSideView\((NoseLengthRatio, TailLengthRatio)\)
   Internal function. Defines the control polygons of the fuselage in side view

Build()
   Overrides the AirconicsShape empty Build method.
   Calls BuildFuselageOML, which has been maintained for older versions.

BuildFuselageOML\((Max\_attempt=5)\)
   Builds the Fuselage outer mould line Notes —— It is not expected that users will interact with this directly.
   Use the Fuslage class initialisation fuction instead

CockpitWindowContours\((Height=1.62, Depth=5)\)
   This function is currently not tested

FuselageLongitudinalGuideCurves\((NoseLengthRatio, TailLengthRatio)\)
   Internal function. Defines the four longitudinal curves that outline the fuselage (outer mould line).

MakeWindow\((Xwc, Zwc)\)
   Makes at Window centered at Wxc Zwc using the bspline wire returned by WindowContour
   THIS FUNCTION IS IN DEVELOPMENT AND NOT YET TESTED FULLY

   Parameters  Xwc : scalar
               The window center x coordinate

               Zwc : scalar
               The window center z coordinate

   Returns  WinStbd : TopoDS_Shape
                  The window surface cut out (starboard side)

                  WinPort : TopoDS_Shape
                  The window surface cut out (Port side)

   Notes

   Changes the contents of self['OML']. Makes both the port and starboard windows at the input location.

TransformOML()
   Use parameters defined in self to scale and translate the fuselage

WindowContour\((WinCenter)\)
   Creates and returns the contour of the window at WinCenter

       Parameters  WinCenter : list or array, length 2
                   The [X, Z] coordinate of the center of the window

<table>
<thead>
<tr>
<th>pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>popitem</td>
</tr>
<tr>
<td>setdefault</td>
</tr>
<tr>
<td>update</td>
</tr>
<tr>
<td>values</td>
</tr>
</tbody>
</table>

Table 3.6 – continued from previous page
Returns \( W_{\text{wire}} \) : TopoDS_Wire

The wire of the B-spline contour

### 3.5 Engine

class airconics.engine.Engine (\( H\text{Chord}=0, \) CentreLocation=[0, 0, 0], \( \text{ScarfAngle}=3, \) HighlightRadius=1.45, MeanNacelleLength=5.67, construct\_geometry=True)

Bases: airconics.base.AirconicsShape

A class for generating aircraft engine and pylon geometries. Currently only yields a turbofan engine with nacelle similar to that of an RR Trent 1000 / GEnx. Shapes produced include the nacelle, spinner cone, tail cone, Fan disk, Bypass disk, and pylon symmetry plane. The nacelle is produced by inclining an inlet disk by its scarf angle about the span-wise (y) axis and uniformly spacing airfoil ‘ribs’ before lofting a surface through them. The pylon is currently the symmetry plane of a fully pylon only

**Parameters**

- **HChord** : OCC.Geom.Handle_Geom_TrimmedCurve
  The chord line at which the engine will be fitted. The result of OCC.GC.GC_MakeSegment.Value() (can be return from helper function CutSect from AirCONICStools).

- **CentreLocation** : list, length 3 (default=[0,0,0])
  Location of the centre of the inlet highlight disk

- **ScarfAngle** : scalar, deg (default=3)
  Angle of inclination of engine intake (rotated around y axis)

- **HighlightRadius** : scalar (default=1.45)
  Intake highlight radius

- **MeanNacelleLength** : scalar (default=5.67)
  Mean length of the nacelle, to be used as the airfoil rib chordlength

- **construct\_geometry** : bool
  If true, Build method will be called on construction

See also:

airconics.base.AirconicsShape, airconics.primitives.Airfoil

**Notes**

- Also calls the initialiser of parent class AirconicsShape which stores all keywords as attributes

**Attributes**

- **_Components** : (dictionary of shapes)

**Methods**
Airconics Documentation, Release

AddComponent
Build
BuildTurbofanNacelle
Display
DisplayBBox
Extents
MirrorComponents
PrintComponents
RemoveComponent
RotateComponents
ScaleComponents_Uniformal
TransformComponents_Nonuniformal
TranslateComponents
Write
clear
get
items
iteritems
iterkeys
itervalues
keys
pop
popitem
setdefault
update
values

Build()
Currently only calls BuildTurbofanNacelle.

Notes
May add options for other engine types
BuildTurbofanNacelle()
The defaults yield a nacelle similar to that of an RR Trent 1000 / GEnx
#TODO: break this down into modular function calls

3.6 Topology

class airconics.topology.Topology(parts={}, construct_geometry=False)
    Bases: airconics.base.AirconicsCollection

Class to define abstract aircraft topologies as extensible lists of lifting surfaces, enclosure, and propulsion type objects.

Parameters parts - dictionary
    Should contain the following, {name: (Part, arity)}
    i.e. the string 'name' values are presented as a tuple or list of:
**Part - TopoDS_Shape**  The shape

**arity - int**  the arity (number of descendant nodes) attached to part

A warning is raised if arities are not provided, in which case arity is assumed to be zero

**Notes**

•warning will be raised if no affinities are provided

**example:**  # (Wing is an airconics Lifting Surface instance):  aircraft = Topology(parts={'Wing': (Wing['Surface'], 2)})

Although not enforced, parts should be added to this class recursively (from the top node first) to represent the aircraft’s flattened topological tree suggested by Sobester [1]. It is the user’s responsibility to ensure the input nodes are a valid lisp tree for a correct graph to result (no checks are currently performed)

See Also: AirconicsCollection

**References**


**Attributes**

- _Tree - list  the list of LISP-like instructions (in the order they were called with AddPart)

**Methods**

- AddPart
- Build
- Display
- MirrorSubtree
- Write
- clear
- export_graphviz
- get
- items
- iteritems
- iterkeys
- itervalues
- keys
- pop
- popitem
- setdefault
- update
- values
**AddPart** 
*part, name, arity=0*

Overloads the AddPart method of AirconicsCollection base class to append the arity of the input topology node.

**Parameters**
- **part** - LiftingSurface, Engine or Fuselage class instance
  the part to be added to the tree
- **name** - string
  name of the part (will be used to look up this part in self.aircraft)
- **arity** - int
  The number of terminals attached to this part; this will be randomized at a later stage

**Notes**
This method is expected to be used recursively, therefore the order in which parts are added dictates the tree topology. The first item added will be the top of the tree.

See also: AirconicsCollection.AddPart

**Build**
Recursively builds all sub components in the current topology tree if self.construct_geometry is true. Will also mirror components if a mirror node has been added, regardless of if construct_geometry is true.

Uses the the Build method of all sub components. Any user defined classes must therefore define the Build method in order for this to work correctly.

**MirrorSubtree**
Mirrors the geometry where required, based on the current topology tree.

Does nothing is no mirror plane has been added

**export_graphviz**
Returns a string, Graphviz script for visualizing the topology tree.
Currently only set up to allow a single mirror terminal

Returns **output** : string
The Graphviz script to plot the tree representation of the program.

**Notes**
This function is originally from GPLearns _Program class, but has been modified. Can be visualised with pydot.

**Example**

```python
>>> topo = Topology()  # Add some parts with topo.addPart
>>> graph = pydot.graph_from_dot_data(topo.export_graphviz())
>>> Image(graph.create_png())
```

May add a dependency on GPLearn later and overload the appropriate class methods.
3.7 AirCONICStools

Various geometry operations of geometric pythonocc primitives for OCC_AirCONICS

Created on Fri Dec 4 11:58:52 2015

@author: pchambers

airconics.AirCONICStools.AddCone (BasePoint, Radius, height, direction=<Mock name='mock()' id='140564126747280'>)

Generates a cone shape originating at BasePoint with base Radius and height (points in the direction of input 'direction)

Parameters  
BasePoint : OCC.gp.gp_Pnt or array length 3
    The centre base point
Radius : scalar
    Cone base radius
height : scalar
    Cone height
direction : OCC.gp.gp_Dir (default: positive x direction)
    the direction of the cones axis i.e. normal to the base: defaults to x axis

Returns  
shape : TopoDS_Shape
    The generated Cone

airconics.AirCONICStools.AddSurfaceLoft (objs, continuity=<Mock id='140564313664528'>, check_compatibility=True, solid=True, first_vertex=None, last_vertex=None, max_degree=8, close_sections=True)

Create a lift surface through curve objects

Parameters  
objs : list of python classes
    Each obj is expected to have an obj.Curve attribute : see airconics.primitives.airfoil class
continuity : OCC.GeomAbs.GeomAbs_XX type (default C2)
    The order of continuity (C^0, C^1, C^2, G^0, ....)
check_compatibility : bool (default=True)
    Adds a surface compatibility check to the builder
solid : bool (default=True)
    Creates a solid object from the loft if True
first_vertex : TopoDS_Vertex (optional, default=None)
    The starting vertex of the surface to add to the ‘ThruSections’ algorithm
last_vertex : TopoDS_Vertex (optional, default=None)
    The end vertex of the surface to add to the ‘ThruSections’ algorithm
max_degree : int (default=8)
    The order of the fitted NURBS surface
**close_sections**: bool (default=True):

Connects the start and end point of the loft rib curves if true. This has the same effect as adding an airfoil trailing edge.

**Returns shape**: TopoDS_Shape
The generated loft surface

**Notes**

Uses OCC.BRepOffsetAPI.BRepOffsetAPI_ThruSections. This function is ORDER DEPENDANT, i.e. add elements in the order through which they should be lofted

**airconics.AirCONICStools.BBox_FromExtents** *(xmin, ymin, zmin, xmax, ymax, zmax)*
Generates the Wire Edges defining the Bounding Box defined in the input arguments: Can be used to display the bounding box

**airconics.AirCONICStools.CalculateSurfaceArea** *(shape)*
Calculates the surface area of input shape

**Parameters shape**: TopoDS_Shape

**Returns Area**: scalar
Calculated surface area

**airconics.AirCONICStools.CutSect** *(Shape, SpanStation)*

**Parameters Shape**: TopoDS_Shape
The Shape to find planar cut section (parallel to xz plane)

**SpanStation**: scalar in range (0, 1)
y-direction location at which to cut Shape

**Returns Section**: result of OCC.BRepAlgoAPI.BRepAlgoAPI_Section (TopoDS_Shape)
The cut section of shape given a cut plane parallel to xz at input Spanstation.

**Chord**: result of OCC.GC.GC_MakeSegment.Value (Geom_TrimmedCurve)
The Chord line between x direction extremeties

**airconics.AirCONICStools.ExtrudeFace** *(face, vec=<Mock name='mock()' id='140564313766928'>)*
Extrudes a face by input vector

**Parameters face**: TopoDS_Face

**vec**: OCC.gp.gp_Vec
The offset vector to extrude through

**Returns shape**: TopoDS_Shape
The extruded shape

**Notes**

Uses BRepBuilderAPI_MakePrism
airconics.AirCONICStools.FilletFaceCorners(face, radius)
Fillets the corners of the input face

Parameters
- face : TopoDS_Face
- radius : the Fillet radius

airconics.AirCONICStools.Generate_InterpFunction(Values, EpsArray=None, uniform=True)
Generates a lookup interpolation function.
Given an array of spanwise coordinates epsilon along a curvilinear leading-edge attached coordinate system, and a set of values describing e.g. Chord, Sweep at each station, generate and return a function f(epsilon) which will give the interpolated value.

Parameters
- Values : array of float
  Values of e.g. chordlength, sweep at each spanwise location in EpsArray
- EpsArray : array of float
  Distribution of spanwise coordinates at which the Values are known
- uniform : bool
  If True, assumes that Values corresponds to uniformly distribution epsilon locations along the lifting surface span

Returns
- f : function
  the function which returns the interpolated epsilon

airconics.AirCONICStools.ObjectsExtents(breps, tol=1e-06, as_vec=False)
Compute the extents in the X, Y and Z direction (in the current coordinate system) of the objects listed in the argument.

Parameters
- breps : list of TopoDS_Shape
  The shapes to be added for bounding box calculation
- tol : float (default=1e-6)
  Tolerance for bounding box calculation
- as_vec : bool (default=False)
  If true, returns minimum and maximum points as tuple of gp_Vec

Returns
- xmin, ymin, zmin, xmax, ymax, zmax : scalar
  the min and max points of bbox (returned if as_vec=False)
- (gp_Vec(xmin, ymin, zmin), gp_Vec(xmax, ymax, zmax)) : tuple of gp_Vec
  the min and max points of bbox (returned in as_vec=True)

Notes
Due to the underlying OCC.Bnd.Bnd_Box functions, the bounding box is calculated via triangulation of the shapes to avoid inclusion of the control points of NURBS curves in bounding box calculation

airconics.AirCONICStools.PlanarSurf(geomcurve)
Adds a planar surface to curve

Parameters
- geomcurve : OCC.Geom type curve
The edge of the profile

**Returns surf**: TopoDS_face

the planar surface

`airconics.AirCONICStools.SplitShapeFromProjection(shape, wire, direction, return_section=True)`

Splits shape by the projection of wire onto its face

**Parameters**

- **shape**: TopoDS_Shape
  - the brep to subtract from
- **wire**: TopoDS_Wire
  - the tool to use for projection and splitting
- **direction**: OCC.gp.gp_Dir
  - the direction to project the wire
- **return_section**: bool
  - returns the split shape

**Returns**

- **newshape**: TopoDS_Shape
  - input shape with wire subtracted
- **section**: the shape which was substracted
  - (returned only if return_section is true)

**Notes**

Currently assumes splits the first face only

`airconics.AirCONICStools.TrimShapebyPlane(Shape, Plane, pnt=<Mock name='mock()' id='140564123491472'>)`

Trims an OCC shape by plane. Default trims the negative y side of the plane

**Parameters**

- **Shape**: TopoDS_Shape
  - Plane: expect TopoDS_Face
  - pnt: point defining which side of the halfspace contains its mass

`airconics.AirCONICStools.Uniform_Points_on_Curve(curve, NPoints)`

Returns a list of uniformly spaced points on a curve

**Parameters**

- **crv**: OCC.Geom curve type
  - NPoints: int
    - number of sampling points along the curve

**class** `airconics.AirCONICStools.assert_isdone(to_check, error_statement)`

Bases: object

raises an assertion error when IsDone() returns false, with the error specified in error_statement -> this is from the pythonocc-utils utility-may not use it?

`airconics.AirCONICStools.boolean_cut(shapeToCutFrom, cuttingShape, debug=False)`

Boolean cut tool from PythonOCC-Utils
**airconics Documentation, Release**

```python
airconics.AirCONICStools.coerce_handle(obj)
coerces an object that has a GetHandle method to call this method and return its handle
```

```python
airconics.AirCONICStools.coslin(TransitionPoint, NCosPoints=24, NLinPoints=24)
Creates a series of abscissas with cosine spacing from 0 to a TransitionPoint and a linear spacing thereafter, up to 1. The TransitionPoint corresponds to pi. Distribution suitable for airfoils defined by points. TransitionPoint must be in the range \([0,1]\).

**Parameters**
- **TransitionPoint**: scalar
  - Point to transition from cosine to linear distribution in range \((0, 1)\)
- **NCosPoints**: int
  - Number of points to space by cosine law between 0 and TransitionPoint
- **NLinPoints**: int
  - Number of points to space by linear law between TransitionPoint and 1

**Returns**
- **Abscissa**: numpy array
  - The generated abscissa
```

```python
airconics.AirCONICStools.export_STEPFile(shapes, filename)
Exports a .stp file containing the input shapes

**Parameters**
- **shapes**: list of TopoDS_Shape
  - Shapes to write to file
- **filename**: string
  - The output filename
```

```python
airconics.AirCONICStools.export_STEPFile_Airconics(AirconicsShapes, filename)
Writes a Step file with names defined in the AirconicsShapes. This function is not fully tested and should not yet be used.

**Notes**

Work in progress

```python
airconics.AirCONICStools.make_circle3pt(pt1, pt2, pt3)
Makes a circle allowing python lists as input points
```

```python
airconics.AirCONICStools.make_edge(*args)
```

```python
airconics.AirCONICStools.make_ellipsoid(centre_pt, dx, dy, dz)
Creates an ellipsoid from non-uniformly scaled unit sphere
```

```python
airconics.AirCONICStools.make_face(*args)
```

```python
airconics.AirCONICStools.make_pipe_shell(spine, profiles, support=None)
```

```python
airconics.AirCONICStools.make_vertex(*args)
```

```python
airconics.AirCONICStools.make_wire(*args)
```

```python
airconics.AirCONICStools.mirror(brep, plane='xz', axe2=None, copy=False)
Originally from pythonocc-utils : might add dependency on this? Mirrors object
```
**Parameters**

- **brep**: OCC.TopoDS.TopoDS_Shape
  - The shape to mirror

- **plane**: string (default = ‘xz’)
  - The name of the plane in which to mirror objects. Acceptable inputs are any of ‘xy’, ‘yx’, ‘yz’, ‘zy’, ‘zy’. Overwritten if axe2 is defined.

- **axe2**: OCC.gp.gp_Ax2
  - The axes through which to mirror (overwrites input ‘plane’)

- **copy**: bool
  - If True, returns a copy of the shape; if False, returns the transformed shape.

**Returns**

- **BRepBuilderAPI_Transform.Shape**: TopoDS_Shape
  - The reflected shape

**Notes**

Pchambers: Added a functionality here to specify a plane using a string so that users could avoid interacting with core OCC objects.

**airconics.AirCONICStools.point_array_to_TColgp_PntArrayType(array, _type=<Mock id='140564313662096'>)**

- **Parameters**
  - **array**: array (Npts x 3) or list
    - Array of xyz points for which to fit a bspline
  - **_type**: type of TColgp array
    - Tested inputs are,
      - TColgp_Array1OfPnt
      - TColgp_HArray1OfPnt
    - See Notes for more information

- **Returns**
  - **pt_arr**: TCOLgp_Array1OfPnt
    - OCC type array of points

**Notes**

Use TColgp_Harray when interpolating a curve from points with the GeomAPI_Interpolate. Use TColgp_Array when interpolating a curve from points with the GeomAPI_PointsToBspline.

**airconics.AirCONICStools.points_from_intersection(plane, curve)**

- **Parameters**
  - **plane**: Geom_Plane
    - The Plane
  - **curve**: Geom_*Curve
    - The Curve

- **Returns**
  - **P**: Point or list of points
A single intersection point (OCC gp gp_Pnt) if one intersection is found, or list of points if more than one is found.

- If No Intersection points were found, returns None

**Notes**

The plane is first converted to a surface As the GeomAPI IntCS class requires this.

```python
airconics.AirCONICStools.points_to_BezierCurve(pnts)
```

Creates a Bezier curve from an array of points.

**Parameters**

- **pnts**: array or list
  
  x, y, z for an array of points. Allowable inputs are numpy arrays (with dimensions (Npoints x 3)), python list with elements [xi, yi, zi] or list of OCC gp gp_Pnt objects

**Returns**

- **crv**: OCC Geom Geom_BezierCurve

```python
airconics.AirCONICStools.points_to_bspline(pnts, deg=3, periodic=False, tangents=None, scale=False, continuity=<Mock id='140564313664528'>)
```

Points to bspline: originally from pythonocc-utils, changed to allow numpy arrays as input

**Returns**

- **crv**: OCC Geom BSplineCurve

```python
airconics.AirCONICStools.project_curve_to_plane(curve, plane, dir)
```

Computes and returns the cylindrically projected curve onto input plane

**Parameters**

- **curve**: geom Curve
  
  plane: Geom Plane
  
  **dir**: gp Dir (default None)

  The cylindrical projection direction. If None, the project will be normal to the plane

**Returns**

- **Hproj_curve**: Handle Geom Curve

```python
airconics.AirCONICStools.project_curve_to_surface(curve, surface, dir)
```

Returns a curve as cylindrically projected onto the surface shape

**Parameters**

- **curve**: Geom curve or TopoDS Edge/Wire
  
  **surface**: TopoDS Shape
  
  **dir**: gp Dir

  the direction of projection

**Returns**

- **res_curve**: geom curve (bspline only?)

```python
airconics.AirCONICStools.rotate(brep, axe, degree, copy=False)
```

Rotates the brep

Originally from pythonocc-utils: might add dependency on this?

**Parameters**

- **brep**: shape to rotate
  
  **axe**: axis of rotation
  
  **degree**: Number of degrees to rotate through
  
  **copy**: bool (default=False)

**Returns**

- **BRepBuilderAPI Transform Shape**: Shape handle
The handle to the rotated shape

```python
airconics.AirCONICStools.scale_uniform(brep, pnt, factor, copy=False)
```

translate a brep over a vector : from pythonocc-utils

```python
airconics.AirCONICStools.transform_nonuniform(brep, factors, vec=[0, 0, 0],
                                            copy=False)
```

Nonuniformly scale brep with respect to pnt by the x y z scaling factors provided in ‘factors’, and translate by vector ‘vec’

**Parameters**
- `factors`: List of factors [Fx, Fy, Fz]
  - Scaling factors with respect to origin (0,0,0)
- `vec`: List of x,y,z or gp_Vec
  - the translation vector (default is [0,0,0])

**Notes**

- Only tested on 3d shapes
- Assumes factors are define with respect to the origin (0,0,0)

```python
airconics.AirCONICStools.translate_topods_from_vector(brep_or_iterable, vec,
                                                   copy=False)
```

Function Originally from pythonocc-utils, modified to work on objects

translates a brep over a vector

**Parameters**
- `brep`: the Topo_DS to translate
- `vec`: the vector defining the translation
- `copy`: copies to brep if True

### 3.8 examples Subpackage

Modules in this subpackage are to be used for reconstructing example Airconics shapes, e.g. the transonic airliner lifting surface functions. Created on Mon Jan 4 17:28:37 2016

Example script for generating a transonic airliner wing external geometry.

@author: pchambers
airconics.examples.wing_example_transonic_airliner.myAirfoilFunctionAirliner(Epsilon, LE-Point, Chord-Funct, Chord-Factor, Dihedral-Funct, Twist-Funct)

Defines the variation of cross section as a function of Epsilon

airconics.examples.wing_example_transonic_airliner.myChordFunctionAirliner(Epsilon)
User-defined function describing the variation of chord as a function of the leading edge coordinate

airconics.examples.wing_example_transonic_airliner.myDihedralFunctionAirliner(Epsilon)
User-defined function describing the variation of dihedral as a function of the leading edge coordinate

airconics.examples.wing_example_transonic_airliner.mySweepAngleFunctionAirliner(Epsilon)
User-defined function describing the variation of sweep angle as a function of the leading edge coordinate

airconics.examples.wing_example_transonic_airliner.myTwistFunctionAirliner(Epsilon)
User-defined function describing the variation of twist as a function of the leading edge coordinate. The coefficients of the polynomial below come from the following twist values taken off the CRM (used for the AIAA drag prediction workshops): Epsilon = 0: twist = 4.24 Epsilon =0.3: twist = 0.593 Epsilon = 1: twist = -3.343

Created on Fri Jan 15 11:39:16 2016

Example functions for generating the lifting surfaces for the tail of a transport aircraft (fin and tailplane external geometry). Also, this shows that a specific planform can be reconstructed (the tail planform geometry here is an approximation of the B787 tail geometry).

# AirCONICS # Aircraft CONfiguration through Integrated Cross-disciplinary Scripting # version 0.2 # Andras Sobester, 2015. # Bug reports to a.sobester@soton.ac.uk or @ASobester please. #
==============================================================================
@author: pchambers

airconics.examples.tailplane_example_transonic_airliner.myAirfoilFunctionFin(Epsilon, LE-Point, Chord-Funct, Chord-Factor, Dihedral-Funct, Twist-Funct)
Defines the variation of cross section as a function of Epsilon

```
airconics.examples.tailplane_example_transonic_airliner.myAirfoilFunctionTP (Epsilon, LE-Point, ChordFunct, ChordFactor, DihedralFunct, TwistFunct)
```

Defines the variation of cross section as a function of Epsilon

```
airconics.examples.tailplane_example_transonic_airliner.myChordFunctionFin (Epsilon)
User-defined function describing the variation of the fin chord as a function of the leading edge coordinate
```

```
airconics.examples.tailplane_example_transonic_airliner.myChordFunctionTP (Epsilon)
User-defined function describing the variation of the tailplane chord as a function of the leading edge coordinate
```

```
airconics.examples.tailplane_example_transonic_airliner.myDihedralFunctionFin (Epsilon)
airconics.examples.tailplane_example_transonic_airliner.myDihedralFunctionTP (Epsilon)
User-defined function describing the variation of the fin sweep angle as a function of the leading edge coordinate
```

```
airconics.examples.tailplane_example_transonic_airliner.mySweepAngleFunctionFin (Epsilon)
airconics.examples.tailplane_example_transonic_airliner.mySweepAngleFunctionTP (Epsilon)
User-defined function describing the variation of the fin sweep angle as a function of the leading edge coordinate
```

```
airconics.examples.tailplane_example_transonic_airliner.myTwistFunctionFin (Epsilon)
airconics.examples.tailplane_example_transonic_airliner.myTwistFunctionTP (Epsilon)
```

Created on Fri Mar 11 11:36:51 2016

@author: pchambers

```
airconics.examples.boxwing.myAirfoilFunctionBoxWing (Epsilon, LEPoint, ChordFunct, ChordFactor, DihedralFunct, TwistFunct)
```

```
airconics.examples.boxwing.myChordFunctionBoxWing (Epsilon)
```

```
airconics.examples.boxwing.myDihedralFunctionBoxWing (Epsilon)
User-defined function describing the variation of dihedral as a function of the leading edge coordinate Notes
— Could also use numpy vectorize in this function
```

```
airconics.examples.boxwing.mySweepAngleFunctionBoxWing (Epsilon)
airconics.examples.boxwing.myTwistFunctionBoxWing (Epsilon)
```
Acknowledgements

occ_airconics began as fork of AirCONICS, and therefore large parts of the code, documentation, examples and manual are attributed to the AirCONICS developers. For more detail on AirCONICS, please refer to the accompanying reference book or recent papers:


CHAPTER 5

Indices and tables

- genindex
- modindex
- search
a
airconics.AirCONICStools, 46
airconics.base, 25
airconics.examples.boxwing, 55
airconics.examples.tailplane_example_transonic_airliner,
  54
airconics.examples.wing_example_transonic_airliner,
  53
AddAirfoilFromSeligFile() (airconics.primitives.Airfoil method), 33
AddComponent() (airconics.base.AirconicsShape method), 29
AddCone() (in module airconics.AirCONICStools), 46
AddCRMLinear() (airconics.primitives.Airfoil method), 33
AddLinear2() (airconics.primitives.Airfoil method), 34
AddNACA4() (airconics.primitives.Airfoil method), 34
AddPart() (airconics.base.AirconicsCollection method), 27
AddPart() (airconics.topology.Topology method), 45
AddSurfaceLoft() (in module airconics.AirCONICStools), 46
airconics.AirCONICStools (module), 46
airconics.base (module), 25
airconics.examples.boxwing (module), 55
airconics.examples.tailplane_example_transonic_airliner (module), 54
airconics.examples.wing_example_transonic_airliner (module), 53
AirconicsBase (class in airconics.base), 25
AirconicsCollection (class in airconics.base), 26
AirconicsShape (class in airconics.base), 28
Airfoil (class in airconics.primitives), 32
AirfoilFunct (airconics.liftingsurface.LiftingSurface attribute), 37
AirlinerFuselagePlanView() (airconics.fuselage_oml.Fuselage method), 41
AirlinerFuselageSideView() (airconics.fuselage_oml.Fuselage method), 41
ApexPoint (airconics.liftingsurface.LiftingSurface attribute), 37
assert_isdone (class in airconics.AirCONICStools), 49

B
BBox_FromExtents() (in module airconics.AirCONICStools), 47
boolean_cut() (in module airconics.AirCONICStools), 49

Build() (airconics.base.AirconicsBase method), 26
Build() (airconics.base.AirconicsCollection method), 27
Build() (airconics.base.AirconicsShape method), 29
Build() (airconics.engine.Engine method), 43
Build() (airconics.fuselage_oml.Fuselage method), 41
Build() (airconics.liftingsurface.LiftingSurface method), 37
Build() (airconics.topology.Topology method), 45
BuildFuselageOML() (airconics.fuselage_oml.Fuselage method), 41
BuildTurbofanNacelle() (airconics.engine.Engine method), 43

C
CalculateAspectRatio() (airconics.liftingsurface.LiftingSurface method), 37
CalculateProjectedArea() (airconics.liftingsurface.LiftingSurface method), 38
CalculateSemiSpan() (airconics.liftingsurface.LiftingSurface method), 38
CalculateSurfaceArea() (in module airconics.AirCONICStools), 47
ChordFactor (airconics.liftingsurface.LiftingSurface attribute), 38
ChordFunct (airconics.liftingsurface.LiftingSurface attribute), 38
CockpitWindowContours() (airconics.fuselage_oml.Fuselage method), 41
coerce_handle() (in module airconics.AirCONICStools), 49
coslin() (in module airconics.AirCONICStools), 50
CreateConstructionGeometry() (airconics.liftingsurface.LiftingSurface method), 38
CutSect() (in module airconics.AirCONICStools), 47
D
DihedralFunct (airconics.liftingsurface.LiftingSurface attribute), 38
Display() (airconics.base.AirconicsBase method), 26
Display() (airconics.base.AirconicsCollection method), 27
Display() (airconics.base.AirconicsShape method), 29
DisplayBBox() (airconics.base.AirconicsShape method), 30

E
Engine (class in airconics.engine), 42
export_graphviz() (airconics.topology.Topology method), 45
export_STEPFile() (in module airconics.AirconicsTools), 50
export_STEPFile_Airconics() (in module airconics.AirconicsTools), 50
Extents() (airconics.base.AirconicsShape method), 30
ExtrudeFace() (in module airconics.AirconicsTools), 47

F
FilletFaceCorners() (in module airconics.AirconicsTools), 47
Fit_BlendedTipDevice() (airconics.liftingsurface.LiftingSurface method), 38
Fuselage (class in airconics.fuselage_oml), 39
FuselageLongitudinalGuideCurves() (airconics.fuselage_oml.Fuselage method), 41

G
Generate_InterpFunction() (in module airconics.AirconicsTools), 48
GenerateLeadingEdge() (airconics.liftingsurface.LiftingSurface method), 39
GenerateLiftingSurface() (airconics.liftingsurface.LiftingSurface method), 39
GenerateSectionCurves() (airconics.liftingsurface.LiftingSurface method), 39

L
LiftingSurface (class in airconics.liftingsurface), 35

M
make_circle3pt() (in module airconics.AirconicsTools), 50
make_edge() (in module airconics.AirconicsTools), 50
make_ellipsoid() (in module airconics.AirconicsTools), 50
make_face() (in module airconics.AirconicsTools), 50
make_pipe_shell() (in module airconics.AirconicsTools), 50
make_vertex() (in module airconics.AirconicsTools), 50
make_wire() (in module airconics.AirconicsTools), 50
MakeWindow() (airconics.fuselage_oml.Fuselage method), 41
mirror() (in module airconics.AirconicsTools), 50
MirrorComponents() (airconics.base.AirconicsShape method), 30
MirrorSubtree() (airconics.topology.Topology method), 45
myAirfoilFunctionAirliner() (in module airconics.examples.wing_example_transonic_airliner), 53
myAirfoilFunctionBoxWing() (in module airconics.examples.boxwing), 55
myAirfoilFunctionFin() (in module airconics.examples.tailplane_example_transonic_airliner), 54
myAirfoilFunctionTP() (in module airconics.examples.tailplane_example_transonic_airliner), 55
myChordFunctionAirliner() (in module airconics.examples.wing_example_transonic_airliner), 54
myChordFunctionBoxWing() (in module airconics.examples.boxwing), 55
myChordFunctionFin() (in module airconics.examples.tailplane_example_transonic_airliner), 55
myChordFunctionTP() (in module airconics.examples.tailplane_example_transonic_airliner), 55
myDihedralFunctionAirliner() (in module airconics.examples.wing_example_transonic_airliner), 54
myDihedralFunctionBoxWing() (in module airconics.examples.boxwing), 55
myDihedralFunctionFin() (in module airconics.examples.tailplane_example_transonic_airliner), 55
myDihedralFunctionTP() (in module airconics.examples.tailplane_example_transonic_airliner), 55
mySweepAngleFunctionAirliner() (in module airconics.examples.wing_example_transonic_airliner), 54
mySweepAngleFunctionBoxWing() (in module airconics.examples.boxwing), 55
mySweepAngleFunctionFin() (in module airconics.examples.tailplane_example_transonic_airliner), 55

64 Index
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Module/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mySweepAngleFunctionTP()</td>
<td>airconics.examples.tailplane_example_transonic_airliner</td>
<td>55</td>
</tr>
<tr>
<td>myTwistFunctionAirliner()</td>
<td>airconics.examples.wing_example_transonic_airliner</td>
<td>54</td>
</tr>
<tr>
<td>myTwistFunctionBoxWing()</td>
<td>airconics.examples.boxwing</td>
<td>55</td>
</tr>
<tr>
<td>myTwistFunctionFin()</td>
<td>airconics.examples.tailplane_example_transonic_airliner</td>
<td>55</td>
</tr>
<tr>
<td>myTwistFunctionTP()</td>
<td>airconics.examples.tailplane_example_transonic_airliner</td>
<td>55</td>
</tr>
<tr>
<td>NSegments</td>
<td>airconics.liftingsurface.LiftingSurface attribute</td>
<td>39</td>
</tr>
<tr>
<td>ObjectsExtents()</td>
<td>airconics.AirCONICStools</td>
<td>48</td>
</tr>
<tr>
<td>PlanarSurf()</td>
<td>airconics.AirCONICStools</td>
<td>48</td>
</tr>
<tr>
<td>point_array_to_TColgp_PntArrayType()</td>
<td>airconics.AirCONICStools</td>
<td>51</td>
</tr>
<tr>
<td>points</td>
<td>airconics.primitives.Airfoil attribute</td>
<td>34</td>
</tr>
<tr>
<td>points_from_intersection()</td>
<td>airconics.AirCONICStools</td>
<td>51</td>
</tr>
<tr>
<td>points_to_BezierCurve()</td>
<td>airconics.AirCONICStools</td>
<td>52</td>
</tr>
<tr>
<td>points_to_bspline()</td>
<td>airconics.AirCONICStools</td>
<td>52</td>
</tr>
<tr>
<td>PrintComponents()</td>
<td>airconics.base.AirconicsShape method</td>
<td>30</td>
</tr>
<tr>
<td>project_curve_to_plane()</td>
<td>airconics.AirCONICStools</td>
<td>52</td>
</tr>
<tr>
<td>project_curve_to_surface()</td>
<td>airconics.AirCONICStools</td>
<td>52</td>
</tr>
<tr>
<td>RemoveComponent()</td>
<td>airconics.base.AirconicsShape method</td>
<td>30</td>
</tr>
<tr>
<td>rotate()</td>
<td>airconics.AirCONICStools</td>
<td>52</td>
</tr>
<tr>
<td>RotateComponents()</td>
<td>airconics.base.AirconicsShape method</td>
<td>30</td>
</tr>
<tr>
<td>ScaleFactor</td>
<td>airconics.liftingsurface.LiftingSurface attribute</td>
<td>39</td>
</tr>
<tr>
<td>Sections</td>
<td>airconics.liftingsurface.LiftingSurface attribute</td>
<td>39</td>
</tr>
<tr>
<td>SplitShapeFromProjection()</td>
<td>airconics.AirCONICStools</td>
<td>49</td>
</tr>
<tr>
<td>SweepFunct</td>
<td>airconics.liftingsurface.LiftingSurface attribute</td>
<td>39</td>
</tr>
<tr>
<td>Topology</td>
<td>airconics.topology</td>
<td>43</td>
</tr>
<tr>
<td>transform_nonuniform()</td>
<td>airconics.AirCONICStools</td>
<td>53</td>
</tr>
<tr>
<td>TransformComponents_Nonuniform()</td>
<td>airconics.base.AirconicsShape method</td>
<td>31</td>
</tr>
<tr>
<td>TransformOML()</td>
<td>airconics.fuselage_oml.Fuselage method</td>
<td>41</td>
</tr>
<tr>
<td>translate_topods_from_vector()</td>
<td>airconics.AirCONICStools</td>
<td>53</td>
</tr>
<tr>
<td>TranslateComponents()</td>
<td>airconics.base.AirconicsShape method</td>
<td>31</td>
</tr>
<tr>
<td>TrimShapebyPlane()</td>
<td>airconics.AirCONICStools</td>
<td>49</td>
</tr>
<tr>
<td>Uniform_Points_on_Curve()</td>
<td>airconics.AirCONICStools</td>
<td>49</td>
</tr>
<tr>
<td>Write()</td>
<td>airconics.base.AirconicsBase method</td>
<td>26</td>
</tr>
<tr>
<td>Write()</td>
<td>airconics.base.AirconicsCollection method</td>
<td>27</td>
</tr>
<tr>
<td>Write()</td>
<td>airconics.base.AirconicsShape method</td>
<td>31</td>
</tr>
</tbody>
</table>

**Index 65**