User's Manual for Genair

Hugo Gagnon

February 1, 2015

Contents

1 Preliminaries

1.1 Scope

Genair is a high-fidelity conceptual aircraft design tool. At the moment it is most useful to generate the outer mold line of general aircraft such as the blended wing-body, the box wing, the truss-braced wing, etc. It does that through a component-based approach, whereby wings, fuselages, nacelles, etc., can be arbitrarily assembled in three-dimensional space.

Genair has been developed for high-fidelity, exploratory aerodynamic shape optimization. As such, it emphasizes on *high-quality* surfaces, *flexible* aircraft components, and an *interactive* interface. The graphical capability of the interface has been kept to a minimum. The intention was to prioritize the development of an intuitive API meant to steer large-scale, multidisciplinary optimizations.

Genair parameterizes aircraft components with non-uniform rational B-splines (NURBS), the same mathematical representation used in CAD. Hence, Genair can be thought of as a hybrid between CAD and a parametric aircraft design tool. The main difference with CAD is that model topology is ignored. Moreover, surface trimming at wing-body junctions and the like is still somewhat experimental.[1]

Genair uses object-oriented programming in Python to encapsulate aircraft components. Each component defines its own "construction recipe", thereby hiding most of the geometry generation process to the user. Naturally, a good understanding of NURBS or even Python is still required when designing custom shapes or implementing new components.

1.2 Suggested Readings

Genair is not your typical CAD package, in that it remains a fairly low-level library of classes and functions. The advantage is an increase in flexibility; the disadvantage is the steep learning curve.

Here are some topics I suggest you brush up on before you install Genair.

1.2.1 Non-Uniform Rational B-Splines

Anyone serious about aircraft design should know about NURBS. The go-to reference for NURBS is, surprise surprise, "The NURBS Book" [1]. I suggest you read Chapters 1 to 4, and skim through Chapters 5 to 10. If you are short on time then at least read "On NURBS: a survey" [2].

By the end of your readings you should be able to answer questions such as: what is the effect of multiple knots or coincident control points on the geometric and parametric continuity of NURBS curves and surfaces?

[1] In practice, at least in Prof. David Zingg's research group, these limitations are inconsequential since, first, we assume fixed topology throughout an optimization, and, second, surface trimming is in any case best handled by the geometry kernel (e.g. Parasolid) native to the mesh generator (e.g. ANSYS ICEM CFD) used to generate a watertight grid.

1.2.2 Free-Form Deformation

Free-form deformation (FFD) [3] is not only central to the geometry control system implemented in Jetstream (Appendix A), but is also used in Genair for the manual design of aircraft components. In both cases a significant departure from standard FFD is the fact that it is the control points of the deforming objects that are manipulated as opposed to their discretization.

1.2.3 Python, NumPy, and IPython

To use Genair effectively you will need a working knowledge of Python. Fortunately, Python is very easy to learn and use. I suggest you read "The Python Tutorial", which is part of the standard documentation [4]. Pay special attention to the chapters on "Data Structures" and "Classes".

What makes Python a scientific computing language comparable to Matlab is NumPy. Genair makes extensive use of NumPy's "array object". To learn about the array object and its use in linear algebra I suggest you read the "Tentative NumPy Tutorial" [5]. Pay special attention to the section on "Copies and Views".

Finally, Genair uses IPython for its command-line interface. IPython can be thought of as the "Command Window" of Matlab but for Python. I suggest you read the "Introduction" chapter of IPython's documentation [7]. Pay special attention to the section called "Enhanced interactive Python shell", including the subsection called "Main features of the interactive shell".

2 Introduction

2.1 Installation

2.1.1 Python Distribution

Before you install Genair you will need a Python 2 interpreter [4] along with the NumPy [5], SciPy [6], IPython [7], PyOpenGL [8], pyglet [9], and PIL [10] packages. While it is possible to install all of these using the standard Distutils, I highly recommend using a third-party Python distribution such as Continuum Analytics' Anaconda [11]. Anaconda is free of charge and ships with a very handy package manager called "conda".

The following instructions assume that you have installed Anaconda, but they can be easily adapted to any Python distribution.

If you have not already done so, start by updating conda followed by Anaconda itself:

```
$ conda update conda
$ conda update anaconda
```

Next, create a new conda environment called "genair" and activate it in your current shell:

```
$ conda create -n genair pip
$ source activate genair
```

The second command should have modified your shell path and prepended the string "`(genair)`" to your terminal prompt. To install the packages listed above proceed with

```
(genair)$ conda install ipython numpy scipy pil
```

Then, depending on whether "`conda search pyopengl`" returns something or not (as of February 2015 it does on OS X but not on Linux), use either one of

```
(genair)$ conda install pyopengl
```

or

```
(genair)$ pip install pyopengl
```

Finally, install pyglet:

```
(genair)$ pip install pyglet
```

To test IPython and NumPy, try the following in a new terminal:

```
(genair)$ ipython
```

which should put you in an IPython shell. From there type[2]

```
>>> import numpy as np
>>> np.__version__
```

and see if the version string matches the one returned by "`!conda list`".

From time to time, say every other month, you may want to update the packages to their latest version:[3]

```
(genair)$ conda update --all
(genair)$ pip install --upgrade pyglet
```

2.1.2 Genair

Installing Genair should be as simple as

```
$ git clone /nfs/carv/d1/people/comp-aero/genair.git
```

To test your clone of Genair, "`cd`" into "`genair/`" and type[4]

```
$ source activate genair
(genair)$ ./main.py
```

[2] By convention, in this manual the default Python prompt (`>>>`) is used rather than IPython's prompt (`In [n]`, where $n \in \mathbb{Z}^+$). [3] Repeat the second command for PyOpenGL if you installed it with pip. [4] The first command assumes that you are using Anaconda; see the previous section.

which should, again, put you in an IPython shell. The real test, however, is whether pyglet supports your graphics card. If it does *not* then you probably already know it from the error messages triggered by executing "`main.py`". If there were no such messages then cross your fingers and try this one last command:

```
>>> draw(Point())
```

which should open a black window with a yellow point in the middle of it. You can close that window by pressing the Escape key and you can exit Genair like you would exit IPython, i.e. by typing "`exit`" at the prompt.

2.2 Quick Start

To give you a sense of what Genair can do I propose to go over a simple tutorial where you will generate a box wing.

Start by launching Genair:[5]

```
$ source activate genair
(genair)$ ./main.py
```

which will automatically put you in the "`play/`" subdirectory of Genair's root directory.[6] You can see this for yourself by typing "`%pwd`" at the prompt.[7] By typing "`%ls`" you will also see that there is an "`airfoils/`" directory in "`play/`". "`%cd`" into it and type "`%ls`" once again. As you can see the directory already contains a number of airfoil data files.

For this first tutorial you will fit the NACA 0012 airfoil. To do so you will need the Airfoil class, but first you should learn how to use it. In Genair, the easiest way to get information is through the "`?`" operator, e.g.[8]

```
>>> Airfoil?
```

Notice how the docstring tells you what the Airfoil class does and in particular how it is intended to be used. In this case, it just so happens that the example given also uses the NACA 0012 airfoil, so go ahead and follow it step by step, starting with:

```
>>> af = Airfoil('n0012.dat')
```

This commands creates an instance of the Airfoil class and assigns it to the variable "`af`".[9]

[5] Again, the first command assumes that you are using the Anaconda Python distribution; see Section 2.1.1. [6] The "`play/`" subdirectory is not tracked by Git so use it as you please. [7] In IPython, the "`%`" character denotes a magic function. To learn more about IPython's magic function system type "`%magic`". [8] The "`?`" operator is another extremely useful feature of IPython; it lets you inspect an object's docstring without leaving the prompt. Similarly, the "`??`" operator lets you inspect not only an object's docstring but also its source code. Both "`?`" and "`??`" can be used on *any* Python object, e.g. try using them on the "`%pwd`" magic function. [9] At this point it is worth mentioning that, similar to Matlab, IPython can list the variables defined in your namespace by typing "`%whos`".

The output of the previous command will have informed you that the data file contains 132 points. Indeed, for this particular instance of the NACA 0012 airfoil, the lower and upper halves are each defined with 66 points. Type "`!head n0012.dat`" to verify this information. Note also that the first line of "`n0012.dat`" has been stored in

```
>>> af.name
```

In contrast, the "`issymmetric`" and "`issharp`" attributes are both evaluated at runtime.

At this point feel free to inspect the data points in an OpenGL window:[10]

```
>>> draw(af)
```

Press the F2 key to switch to a xz view and then zoom in using the mouse middle button.[11] When you are done close the window by pressing the Escape key.[12]

Next, proceed with the remaining instructions of the example:[13]

```
>>> af.fit()
>>> af.sharpen()
>>> af.fit()
>>> af.transform()
```

The last method transforms the airfoil so that its chord length becomes unity and its quarter-chord point coincides with the global origin.[14]

You should now save your work to avoid repeating the same steps everytime you want to use a NACA 0012 airfoil. Start by making a new subdirectory in "`play/`" called "`tutorial/`", "`%cd`" into it, and then save your Airfoil object likewise:[15,16]

```
>>> save(af, fn='n0012.p')
```

For the sake of demonstration, relaunch Genair, navigate to "`tutorial/`", and type

```
>>> af, = load('n0012.p')
```

The Airfoil object that "`af`" now points to should be in the same exact state than before and can thus be used in the same exact way.

Now to the box wing. We will assume that the three wing segments (lower, tip fin, and upper) meet at right angles.[17] This time you will need the Wing class, so, as before, start by learning about it:

[10] The reason why this command draws points and not a NURBS curve is because the airfoil has not beeen fitted yet. [11] The OpenGL interface is explained in Section 3.2. Meanwhile, have a look at the header of "`plot/controller.py`". [12] As explained in Section 3.2, although Genair supports multiple windows running at the same time, it is best to close them when not needed. [13] Don't forget to use the "?" operator to learn about the purpose and default arguments of each method. [14] These transformations matter when constructing a Wing object (Section 4.1.3). [15] Try the "`%mkdir` magic function. [16] The next section gives an overview of the saving and loading mechanism. [17] An alternative would be to generate smooth corner fillets.

```
>>> Wing?
```

Unsurprisingly, the docstring informs you that you need at least one Airfoil object to instantiate a Wing object. So go ahead and type

```
>>> lower = Wing(af)
```

Next, create the "trajectory curve" along which "`af`" will be swept:[18]

```
>>> T = nurbs.tb.make_linear_curve(Point(), Point(y=3))
```

Assuming that you don't want any twist nor any taper you can finalize the Wing object with the following two commands:[19,20]

```
>>> lower.orient(T)
>>> lower.fit()
```

Finally, give it some sweep:[21,22]

```
>>> lower.glue()
>>> lower.sweep = 30
```

To generate the tip "`fin`" Wing object, repeat the same exact process but use "`Point(z=1)`" instead of "`Point(y=3)`". Also, instead of sweeping it, give it a 90 degree dihedral:

```
>>> fin.glue()
>>> fin.dihedral = 90
```

You could use a similar approach to generate the "`upper`" Wing object, but in this case it is more convenient to simply copy the "`lower`" Wing object and to give to that copy a 180 degree dihedral:[23]

```
>>> upper = lower.copy()
>>> upper.glue()
>>> upper.dihedral = 180
```

Now is a good time to save your work:

```
>>> save(lower, fin, upper, fn='lower_fin_upper.p')
```

[18] This is a common example of where you are required to directly interact with the NURBS library. The library is discussed in Chapter 3; for now, to see other functions available in its "toolbox", type "`nurbs.tb.`" at the prompt followed by the Tab key. [19] As explained in Section 4.1.3, twist and taper are easily specified through B-spline functions. [20] By default, the second command will show you the NURBS representation of the final Wing object. In Genair, keep in mind that what you see is only a *triangulation* of the actual NURBS curves and surfaces. Type "`draw?`" to find out more. [21] Alternatively, one could have generated a trajectory curve with the point "`Point(x=1.73205081, y=3, z=0)`" before orienting and fitting the wing. Try it! [22] The "glue/unglue" mechanism is explained in Section 4.1.1. Basically, it is necessary so that, for example, when sweeping a wing not only does its NURBS representation shear but also its trajectory curve, wing tip (if any), etc. [23] Again, this example reflects the fact that in Genair there are often multiple ways to arrive at the same result.

You may have noticed that all three wing segments are positioned relative to the global origin. (See it for yourself: "`draw(lower, fin, upper)`".) What remains to be done is to connect them one after the other and to make sure that the connections are watertight. The WingMerger class does that for you:

```
>>> box = WingMerger(lower, fin, upper)
>>> box.merge()
```

Finally, you should save your work one last time:

```
>>> save(box, fn='box.p')
```

and perhaps even export the geometry in IGES format if you intend on using it in an external application such as ANSYS ICEM CFD. The next section describes how to do this.

2.3 Saving and Loading

The "`save`" and "`load`" functions demonstrated in the previous section are thin wrappers built around the "`cpickle`" module.[24] A "pickle" is a binary file, typically with the extension "`.p`", that stores any number of any Python objects (well, almost). In Genair, pickles are typically very small in size and are thus ideal to share work with others.

It is important to understand that pickles are not specific to Genair. For example, the snippet

```
>>> def power2(x):
...     return x**2
>>> save(0.1, np.array([3, 2, 1]), power2)
```

works as expected.

You have probably figured out by now that the "`save`" function can take an arbitrary number of arguments. Conversely, "`load`" takes only one argument but returns a list of arbitrary length. While intuitive, this approach works best if you are familiar with the concept of "unpacking".[25] For example, assuming "`a`" is a Python list of length 3, use

```
>>> save(*a)
```

instead of

```
>>> save(a[0], a[1], a[2])
```

Similarly, use

```
>>> a0, a1, a2 = load()
```

instead of[26,27]

[24] Part of the Python's standard library [4]. [25] This concept is also very useful when it comes to using the "`draw`" function described in Section 3.2.1. [26] The second command is more succintly written as: "`a0, a1, a2 = a`". [27] FYI length-1 sequences can also be unpacked, e.g. "`a0, = a`" versus "`a0 = a[0]`".

```
>>> a = load()
>>> a0 = a[0]; a1 = a[1]; a2 = a[2]
```

The “`geom.io`” module defines a few other functions related to I/O.[28] Out of those you will probably find “`save_IGES`” most useful. Note that ultimately “`save_IGES`” only saves points and NURBS curves and surfaces; everything else gets discarded. For example, assuming “`af`” is an Airfoil object, then

```
>>> geom.io.save_IGES(af)
```

only saves the NURBS curves representing the lower and upper portions of the airfoil.[29]

2.4 Code Overview

If you have read and understood Sections 2.2 and 2.3 then you already know more about Genair than you think. To learn about some other useful features, e.g. geometrically nonlinear wings and automatically generated wingtips, you can go ahead and jump to the relevant sections without worrying too much about the code structure.

If, on the other hand, you plan on implementing new features, or even on becoming proficient with Genair, I strongly suggest that you spend some time understanding the machinery behind the interface. Indeed, the true intent of this manual is not so much as documenting each and every aspect of Genair as to explaining its design philosophy.

Central to this philosophy is the notion of inheritance. As seen from the class diagram of Figure 1, all classes defined in Genair (at least those shown here) are derived from the PlotObject class, which means that an instance of any one of those classes will be recognized by the “`draw`” function. This is easily remembered. In general, the end-user (you) only needs to know about three “fundamental” classes: Point, NURBSObject, and Part (all of which are shown in Figure 1). The first is universal and easily understood. The other two are specific to Genair and are based on the same general principle that a superclass inherits the properties and methods of its base class. For example, the FFDVolume class inherits its “`.p`” attribute from the NURBSObject class and the Wing class inherits its “`glue`” method from the Part class.[30] Now, even though the Point, NURBSObject, and Part classes are fundamentally different, they still share some common methods too, including “`copy`”, “`translate`”, “`rotate`”, “`mirror`”, “`scale`”, and “`shear`”.

For example, say “`crv`”, “`fus`”, and “`pts`” are a Curve, a Fuselage, and a *list* of Point objects, respectively; then, the following commands are valid:

[28] Again, use the tab completion feature of IPython to see all available functions. [29] To be precise, its data points would be saved if the airfoil is not fitted. This is because “`save_IGES`” only saves what would be displayed by the “`draw`” function, i.e. what is returned by the “`_draw`” method of a Part object. More on this topic in Sections 3.2.1 and 4.1.1. [30] Use Python’s help system to see the full inheritance tree of an object, e.g. “`help(Wing)`”.

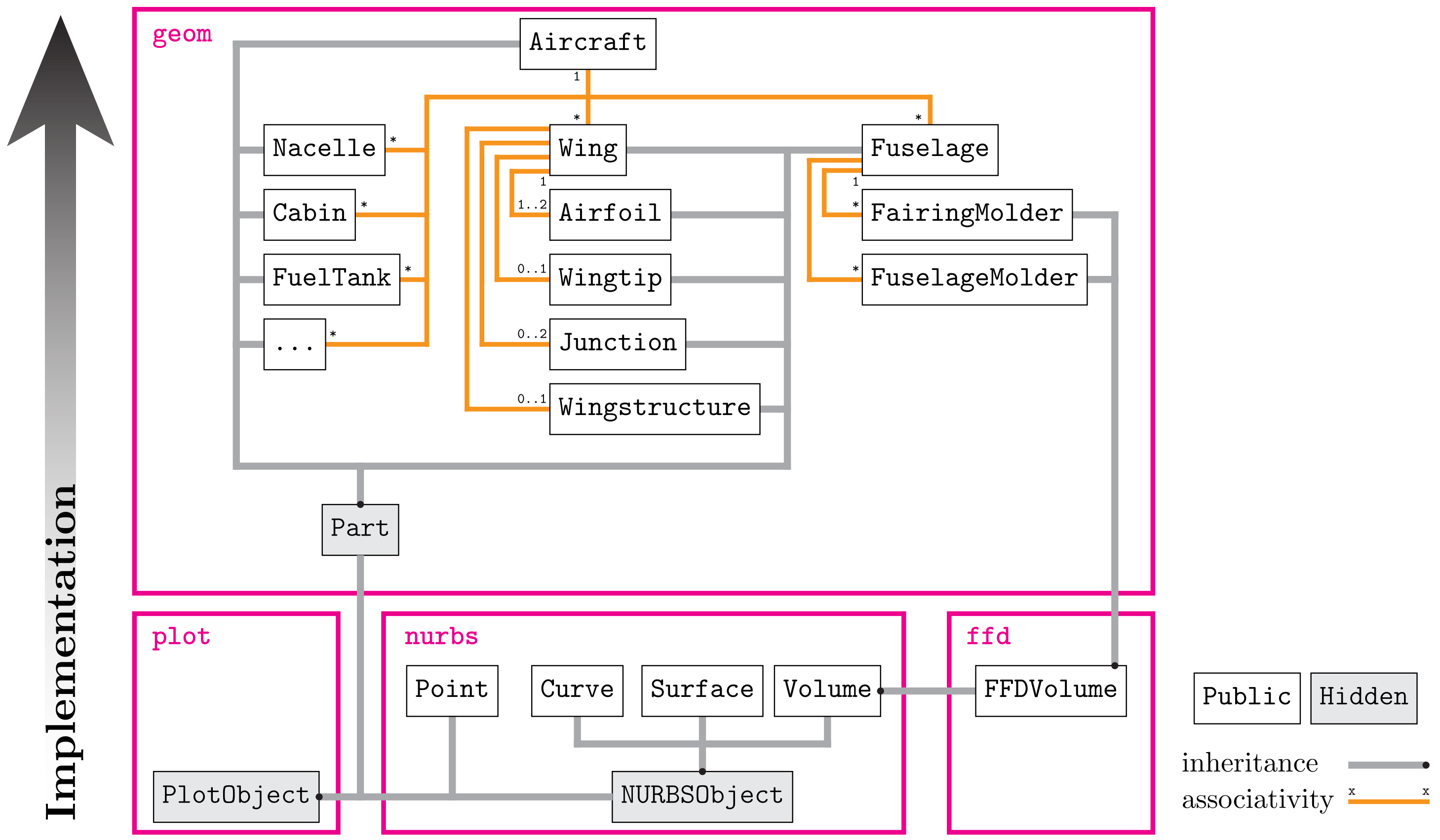

Figure 1: Partial class diagram of Genair. In the associativity tree, the symbol $*$ denotes "zero or more instances" and the notation $y..z$ denotes from "y to z instances".

```
>>> draw(crv, fus, *pts)
>>> crv.elevate(1)
>>> fus.glue(); fus.rotate(90)
>>> for pt in pts:
...     pt.copy()
```

but these are not:

```
>>> draw(pts) # Not a PlotObject
>>> crv.glue() # Not a Part
>>> fus.elevate(1) # Not a NURBSObject
```

The remainder of this manual focuses on the Point and NURBSObject classes of the "`nurbs`" package (Chapter 3) and on the Part class of the "`geom`" package (Chapter 4). Before that here is an overview of the file structure used in Genair:

main.py The script used to launch Genair. It imports commonly used objects, such as the Airfoil and Wing classes, the "`draw`" function, etc., and injects them directly into the IPython namespace. These objects are not listed by "`%whos`" and will not be deleted by a soft reset, i.e. "`%reset -s`".

doc/ The directory related to the documentation. Includes this user's manual.

ffd/ (see Figure 1) The directory related to FFD. It defines the FFDVolume class which is used, for example, by the "`geom`" package.

geom/ (see Figure 1) The directory related to aircraft design. It defines the family of Part classes. Implements high-level utilities, including the "`save`" and "`load`" functions and the "`glue/unglue`" mechanism.

nurbs/ (see Figure 1) The directory related to NURBS. It defines the Point class and the family of NURBSObject classes. Implements low-level utilities, including data reduction and point inversion algorithms.

opti/ The directory related to Jetstream. It defines the Grid, Axial, and Joint classes, as well as utilities to help set up the geometry control system for optimization purposes.

play/ The directory that Genair puts you in upon launching. Use it as a sandbox.

plot/ (see Figure 1) The directory related to OpenGL. It defines the PlotObject class together with the "`draw`" function. Integrates the pyglet event loop with the IPython shell.

3 Non-Uniform Rational B-Spline Library

A NURBS library is a set of utilities that specializes in the generation and manipulation of NURBS. In Genair it is a completely independent entity; in particular, it has no notion of the "`geom`" package.[31]

[31] Hence, it may as well be used by an application other than Genair.

The goal of Genair is to automate aircraft design by making smart and efficient use of the NURBS library. However, for increased flexibility, Genair sometimes forces you to directly interact with the library.[32] It is thus helpful to know how it works and what it can and cannot do.

Most classes and functions discussed here are fairly well documented; don't hesitate to use the "?" and "??" operators to learn more about them.

3.1 `Point` and `NURBSObject` Classes

3.1.1 `Point` Class

The Point class is defined in the "`nurbs.point`" module.[33] As explained in:

```
>>> Point?
```

it can be used either as a regular point or, more commonly, as a control point. Example usage:

```
>>> pt = Point(1, 2, 3)
>>> pt.xyz
>>> pt.xyzw
```

The coordinates of a Point object are actually stored on its "`_xyzw`" attribute.[34] Do not modify this attribute directly, e.g. use[35]

```
>>> pt.xyzw = -1.0, None, None, None
```

as opposed to

```
>>> pt._xyzw[0] = -1.0
```

Actually, you can safely modify "`_xyzw`" as long as you make the modifications *in-place*.[36] If you do the only downside is that the position of the point will not be updated in the OpenGL window(s), if any.

3.1.2 `NURBSObject` Class

The NURBSObject class is defined in the "`nurbs.nurbs`" module. It is the base class of the Curve, Surface, and Volume classes (see Figure 1), respectively defined in the "`nurbs.curve`", "`nurbs.surface`", and "`nurbs.volume`" modules. As explained in:

```
>>> nurbs.nurbs.NURBSObject?
```

[32] A common example is when generating a tracjectory curve for a Wing object. [33] Like most classes discussed here the Point class has already been injected in your namespace, i.e. "`Point`" is shorthand for "`nurbs.point.Point`". [34] In Python, the underscore denotes a private object; it is intended to discourage you from using that object. [35] "`None`" tells the Point object to keep a coordinate intact. [36] This is a recurring concept in Genair; make sure to understand the section on "Copies and Views" of the "Tentative NumPy Tutorial" [5].

it is fully defined by a ControlObject object (storing control point coordinates), degree(s), and knot vector(s).

The ControlObject class is the base class for the ControlPolygon, ControlNet, and ControlVolume classes, respectively required to instantiate a Curve, Surface, and Volume object. As explained in:

```
>>> nurbs.nurbs.ControlObject?
```

it is fully defined by either Point objects or an object matrix. Both formats are stored regardless. Example usage:

```
>>> p0 = Point(1, 0, w=1)
>>> p1 = Point(1, 1, w=1)
>>> p2 = Point(0, 1, w=2)
>>> cpol = ControlPolygon([p0, p1, p2])
```

The “`n`”, “`cpts`”, and “`Pw`” attributes of a ControlObject object store the number of control points (minus one), the Point objects ($N-1$ dimensional NumPy array), and the object matrix (N dimensional NumPy array), respectively.[37]

The “`_xyzw`” array of the Point objects stored in the “`cpts`” array are views (shallow copies) of the corresponding array elements in the “`Pw`” array. Therefore, as long as modifications are performed *in-place*, modifying either type of arrays (“`_xyzw`” or “`Pw`”) modifies the same data. For example,

```
>>> cpol.cpts[2].xyzw = None, 2, None, None
```

is equivalent to

```
>>> cpol.Pw[2,1] = 4
```

Note the 4 instead of the 2. This is because the “`xyzw`” attribute of the Point object automatically converts the coordinates in homogeneous space for you. Thus, “`cpts`” can be thought of as a user-friendly interface to “`Pw`”. Also note that, analogous to the Point class, you must modify “`Pw`” through the “`cpts`” interface in order to correctly update the OpenGL window(s), if any.

Once a ControlObject object is defined, a NURBSObject object is easily instantiated. Continuing our example:[38,39]

```
>>> c = Curve(cpol, (2,))
```

The “`cobj`”, “`p`”, and “`U`” attributes of a NURBSObject object store the ControlObject object, degree(s), and knot vector(s), respectively.

A NURBSObject object can be modified by modifying “`cobj`” or “`U`”. Modifying “`cobj`” consists of modifying “`Pw`” in-place (as explained above). This is typically achieved through one of three interfaces:

1. “`cpts`” (as explained above),

[37] $N = 2$, 3, and 4 for the ControlPolygon, ControlNet, and ControlVolume classes, respectively. [38] The degree is specified as a 1-tuple to be consistent with the Surface and Volume classes. [39] If unspecified the knot vector is assumed uniform.

2. OpenGL (as explained in Section 3.2), and
3. transforms (as explained in Section 3.3).

As for modifying "`U`", the recommended approach is to simply reassign the attribute, e.g.[40]

```
>>> c.U = (np.array([0, 0, 0, 2, 2, 2]),)
```

Internally, the library checks whether the new knot vector(s) are valid.[41] This won't be possible if the modifications are performed in-place. For example, although the command

```
>>> c.U[0][0] = 2
```

is technically valid, the resulting knot vector is invalid.

Modifying "`p`" on an *existing* NURBSObject object does not make sense so the library won't let you. What does make sense is to derive a *new* object that has the same geometric and parametric continuity as the original one but different NURBS degree(s). This is exactly what the "`elevate`" and "`reduce`" methods do, e.g.[42]

```
>>> ce = c.elevate(1)
```

The NURBSObject classes each implement many other useful methods. They are (as of February 2015):

Curve class eval_point, eval_derivatives, eval_curvature, insert, split, extend, refine, decompose, segment, remove, removes, elevate, reduce, project, reverse

Surface class eval_point, eval_derivatives, eval_curvature, insert, split, extract, extend, refine, decompose, segment, remove, removes, elevate, reduce, trim, project, reverse, swap

Volume class eval_point, eval_derivatives, split, extract, extend, refine, elevate, project, reverse, swap

I strongly suggest that you read about and experiment with each one of those methods. Doing so will ultimately increase both your productivity and the quality of your designs. For example, to quickly generate a trajectory curve suitable for a blended winglet, one could do:

```
>>> cpol = ControlPolygon([Point(), Point(y=3)])
>>> T = Curve(cpol, (1,)).elevate(2).insert(0.8, 1)
```

[40] In this example, the same effect can be achieved with the "`nurbs.knot.remap_knot_vec`" function. [41] Have a look at the source code of "`nurbs.knot.check_knot_vec`" to see what constitutes a valid knot vector. [42] As of February 2015 the "`reduce`" method is not implemented on the Volume class.

The second command first instantiates a Curve object of degree 1, which is immediately used to instantiate a Curve object of degree 3, which is immediately used to instantiate a Curve object that has one extra control point toward the tip. Although the same could be achieved with

```
>>> cpol = ControlPolygon([Point(), Point(y=0.8), Point(y=1.8),
...     Point(y=2.8), Point(y=3)])
>>> T = Curve(cpol, (3,),
...     (np.array([0, 0, 0, 0, 0.8, 1, 1, 1, 1]),))
```

in general it is far more difficult to obtain a good parameterization with this second, more "manual" approach.

3.2 OpenGL Interface

The OpenGL interface of Genair is key to a productive design session. The idea is to quickly inspect or even modify parts of your design by opening and closing OpenGL windows on the fly.

The OpenGL interface is not intended to replace the command-line interface but rather to supplement it. As such, the OpenGL windows should be closed as soon as you are done working with them. Besides, opening too many windows at the same time will reduce the overall responsiveness of the application.

3.2.1 The "`draw`" Function

The "`draw`" function opens a new OpenGL window. For example, type:

```
>>> draw(T)
```

to render the Curve object defined at the end of the previous section.[43]

An OpenGL window is an instance of the Figure class. When you open a new window Genair automatically injects the Figure object in your namespace under the variable name "`figN`", where "`N`" is the number of times you have called the "`draw`" function so far.[44] That same variable is automatically deleted from your namespace when you close the window.

As explained in:

```
>>> draw?
```

no matter what combination of PlotObject objects (see Figure 1) you give to "`draw`", only Point and NURBSObject objects are actually rendered.[45,46] All the points, curves, surfaces, and volumes that a Figure object is aware of are stored in its "`pos`" attribute.

[43] The number displayed in the bottom-left corner of the window is the frame per second (FPS); in general, a window is considered responsive if its FPS is at least 60 Hz. [44] The same name is given to the window's caption, so it is easy to know which variable corresponds to which window. [45] When a Part object is drawn, "`draw`" draws the objects returned by its "`_draw`" method; see Section 4.1.1. [46] For Volume objects, only the control point lattice is rendered, not the underlying NURBS representation.

If you want to add (remove) any PlotObject object to (from) an *existing* OpenGL window, use the "`inject`" ("`deject`") method of the Figure object. For example, say "`srf`", "`wi`", and "`pts`" are a Surface, a Wing, and a *list* of Point objects, respectively; then, one could do:

```
>>> draw(srf, wi)
>>> fig2.inject(*pts)
>>> fig2.deject(wi)
```

A Figure object does not render the same object twice, e.g. in the above example the fourth command:

```
>>> fig2.inject(srf)
```

has no effect.[47] However, the same object can be rendered in two or more OpenGL windows simultaneously. This is useful when modifying an object under different views. For example, try the following with the Curve object defined at the end of the previous section:

```
>>> draw(T); draw(T)
```

Reorient the view in each window as you please and then pick and drag individual control points (the next section explains how). The curve should be updated at the same time in both windows.

3.2.2 Controller

An OpenGL window is controlled via mouse and keyboard input. The controls are listed in the header of "`plot/controller.py`". In particular, use the left, middle, and right buttons of your mouse to respectively rotate, zoom, and translate the view.[48]

It is possible to pick Point and NURBSObject objects inside an OpenGL window. Try picking "`T`" in one of the two windows opened in the previous example. If the pick is successful the curve should turn green. Next, try picking its control points one by one.[49]

Everytime you pick something the Figure object updates three variables in your namespace:[50]

```
>>> last_picked_xyz
>>> last_picked_object
>>> last_picked_objects
```

The first two are self-explanatory; the third is a Python list of the last picked object(s) that gets reset on an unsuccessful pick.

In conjunction with "`last_picked_xyz`" it is sometimes convenient to pick one of the two (four) extremities of a curve (surface). To do so pick any point

[47] In other words, the "`pos`" attribute of the Figure object does not change. [48] Make sure to deactivate the Num Lock key if you are on Linux. [49] Tip: you will need the "`F8`" key. [50] Listed by the magic function "`%whos`".

closest to the extremity but press the right button of your mouse instead of the left one.

Similar to the Vi text editor, an OpenGL window can be in one of several "modes". A window responds differently depending on which mode is active. The default mode, "Translate Point", allows you to not only pick Point objects but also to drag them around.[51,52] Similarly, the "Translate NURBS" mode allows you to translate NURBSObject objects.[53]

The mode manager is aware of the "glue/unglue" mechanism described in Section 4.1. For example, while in the "Translate NURBS" mode, try translating an Airfoil object, before and after gluing it.

Finally, the mode manager implements a very basic "undo" mechanism. Despite its simplicity it is actually quite useful, but keep in mind that *closing a window loses all the undos associated with that window!*

3.3 Transforms

A useful property of NURBS is that an affine transformation is achieved by applying the transformation to the control points.

Let A be a 4×4 transformation matrix and P^w be a (reshaped) object matrix, then the product $A \cdot P^w$ achieves the desired transformation.

The "`translate`", "`rotate`", "`mirror`", "`scale`", and "`shear`" functions defined in the "`nurbs.transform`" module each performs $A \cdot P^w$ relative to any arbitrary point, line, or plane in Euclidean space.[54] For example, say "`c`" is a Curve object, then the command

```
>>> nurbs.transform.scale(c.cobj.Pw, 2, L=(1,1,0))
```

scales "`c`" by a factor of 2 in the given direction.

All five transformation functions are implemented as methods on the Point and NURBSObject classes. For example, the command:

```
>>> c.scale(2, L=(1,1,0))
```

is equivalent to the previous one. That being said, using the class methods over the module functions is still preferable. First, they take less effort to type. Second, they will correctly update the OpenGL window(s), if any. Third, analogous to the modes of an OpenGL window, they are aware of the "glue/unglue" mechanism described in Section 4.1. So, say "`af`" is an Airfoil object, the commands:

```
>>> af.glue()
>>> af.nurbs.rotate(90)
```

[51] The translation occurs in the plane perpendicular to the view; this is also true for the "Translate NURBS" mode. [52] If the point is a control point then the ControlObject object of the NURBSObject object will be automatically updated. [53] You can toggle between the two modes by simultaneously pressing the Control, Alt, and N keys. [54] Have a look at their docstring, e.g. "`nurbs.transform.scale?`"

also rotate the lower and upper halves of the airfoil (“`af.halves`”) as well as its camber line (“`af.CL`”).[55,56]

3.4 Toolbox

Section 3.1 explains how to instantiate a NURBBObject object, either directly through class constructors or indirectly through class methods. A third way is through the “toolbox” of the NURBS library.

The toolbox helps you generate and manipulate NURBS curves, surfaces, and volumes. It is accessible from the “`nurbs.tb`” (virtual) module, which is a single point of access to a set of functions collected from “`curve.py`”, “`surface.py`”, “`volume.py`”, “`conics.py`”, and “`fit.py`”.[57]

For example, the commands:

```
>>> O, X, Y = (0, 1, 0), (1, 0, 0), (0, 1, 0)
>>> a = nurbs.tb.make_ellipse(O, X, Y, 3, 1, 0, 3 * np.pi / 2)
```

generate three-quarter of an ellipse. And the commands:

```
>>> a1 = a.copy(); a1.mirror()
>>> a2 = nurbs.tb.make_composite_curve([a, a1])
```

generate a single curve out of two.

I suggest that you read about and experiment with the remaining tools of the toolbox. Doing so will help you understand how the “`geom`” package works. For example, “`make_swept_surface`” is at the core of the “`Wing`” class. Other utilities that you may find especially useful are “`arc_length_to_param`”, “`param_to_arc_length`”, and “`reparam_arc_length_curve`”.

4 Component-Based Aircraft Design

The only package in Genair that knows anything about aircraft design is “`geom`”. Its main challenge is to translate the vocabulary of an aircraft designer to the classes and functions of the “`nurbs`” package.

Like most tools of its kind, Genair takes a component-based approach to aircraft design. For example, a wing-body configuration is generated from a wing and a fuselage component. In Genair, each component is assigned a different class and each class is derived from the same base class: Part (see Figure 1).

4.1 `Part` Class

The Part class is defined in the “`geom.part`” module. It is to the Airfoil, Wing, etc., classes what the NURBSObject class is to the Curve, Surface, and Volume classes, i.e. it implements functionality common to all aicraft components.

[55] The attributes of the Airfoil class are explained in Section 4.1.2. [56] This example is for demonstration purposes only; the second command is more intuitively written as “`af.rotate(90)`”. [57] “`fit.py`” defines many other useful functions related to the interpolation and approximation of curves and surfaces; have a look!

4.1.1 Common Attributes

The Part class implements the following basic properties and methods: "`bounds`", "`symmetrize`", "`colorize`", "`clamp`", and "`copy`". Refer to their docstring and source code for more information.[58]

Analogous to the Point and NURBSObject classes, the Part class implements all five transformation functions ("`translate`", "`rotate`", "`mirror`", "`scale`", and "`shear`") as methods. For example, assuming "`w`" is a Wing object, one could sweep it back like so:[59]

```
>>> w.glue()
>>> w.shear(30)
```

The "`glue`" and "`unglue`" methods of a Part object control which objects actually transform when the Part object is transformed. In the previous example, these objects include the trajectory curve ("`w.T`"), the wing surface ("`w.nurbs`"), etc., but *exclude* the orientation curve ("`w.Bv`").[60]

More specifically, the "`glue/unglue`" mechanism keeps track of a list of Point and NURBSObject objects assigned to a Part object.[61] Every object in that list is also given a pointer to the list. Hence, transforming any object in the list, either through one of the two "translate" modes of the OpenGL interface (Section 3.2) or one of the five class methods (Section 3.3), has the same effect than transforming the Part object itself. For example, the commands:

```
>>> w.glue()
>>> w.nurbs.shear(30)
```

are equivalent to the previous ones.

The "`glue/unglue`" mechanism is also responsible for updating the "family tree" of a Part object. In the previous example, if "`w`" had a Wingtip object assigned to it, then the wingtip surfaces would have also been sheared.

Note that "gluing" a component also unbinds it from its parent, if any. So, continuing the previous example, the commands

```
>>> w.tip.glue()
>>> w.tip.shear(30)
```

shear only the wingtip. Analogously, the command

```
>>> w.tip.unglue()
```

unglues only the wingtip, but the command

```
>>> w.unglue()
```

[58] Again, use the "`?`" and "`??`" operators of IPython. [59] A more convenient alternative would be to use the "`sweep`" attribute of the Wing class; see Section 4.1.3. [60] Have a look at the "`_glue`" method of a Part object to see which objects it controls, e.g. "`w._glue??`". [61] FYI the list is a Python list called "`glued`".

unglues both the wing and its wingtip.

Finally, each Part class implements its own "`_draw`" method. Similar to "`_glue`", this method returns a list of objects, the difference being that the list only contains objects that the designer is most likely to be interested in at a given stage of the design process. For example, once a Wing object has been fitted, drawing it should no longer display its trajectory curve ("`w.T`") but rather its lower and upper surfaces ("`w.halves`").[62] This information is used by a number of functions other than "`draw`", including those in the "`geom.io`" module.

4.1.2 `Airfoil` Class

The Airfoil class is defined in the "`geom.airfoil`" module. As explained in

```
>>> Airfoil?
```

it creates a B-spline approximation to an airfoil from a set of data points.

The Airfoil class is well documented; please refer to the "Intended section" of its docstring and make sure to read the docstring of each one of its methods as well. You may also want to revisit Section 2.2.

When fitting an airfoil for the first time make sure to find the right selection of parameters that will give you the best possible fit.[63] Generally speaking, a good fit is one that results in few control points (say less than 40) and smooth curvature plots. Unfortunately, depending on the number and quality of the sampled data points, you may need to try several combinations of parameters before you achieve a good fit. For example, a reasonable fit of the NASA SC(2) 0614 airfoil is achieved with the parameter "`E`" set to 0.0015:[64,65,66]

```
>>> a = Airfoil('sc20614.dat')
>>> a.fit(E=0.0015)
```

Once you have found the right selection of parameters make sure to save your airfoil to avoid repeating the same process.

The "`nurbs`" and "`halves`" attributes of an Airfoil object store the airfoil curve in two *equivalent* forms. "`nurbs`" is a single curve that loops around the airfoil counterclockwise (looking toward the negative y axis) starting from the trailing edge. It is parameterized on $[0, 1]$, where the parameter 0.5 corresponds to the leading edge. "`halves`" is a list of two curves, each starting at the leading edge, representing the lower and upper portions of the airfoil. They are also both parameterized on $[0, 1]$.

The direct analogs of the "`nurbs`" and "`halves`" attributes are implemented on the Wing class.

[62] See it for for yourself: "`Wing._draw??`". [63] The default parameters were chosen based on the NACA 0012 airfoil. [64] The "`sc20614.dat`" file should be located in your "`airfoils/`" directory. [65] See what happens if you set "`E`" to its default value of 0.0001. [66] Note that this airfoil is a good example where you won't be able to use "`sharpen`".

4.1.3 `Wing` Class

The Wing class is defined in the "`geom.wing`" module. As explained in

```
>>> Wing?
```

it creates a wing by sweeping an airfoil along a trajectory curve.

Like the Airfoil class, the Wing class is fairly well documented. Unlike the Airfoil class, however, it is fairly complex. Indeed, the Wing class trades a lot of user-friendliness for flexibility. This level of flexibility is certainly not required when generating trapezoidal wings. For such wings it is in fact more convenient to automate the generation process with scripts; see Appendix B.1.

Rather than describing each and every properties and methods of the Wing class let us go over an example where you will generate a wing with linear twist and nonlinear taper and dihedral.

Start off by loading two previously fitted airfoils, say the NACA 0012 and the NASA SC(2) 0614 airfoils:[67,68]

```
>>> a0, = load('n0012.p')
>>> a1, = load('sc20614.p')
```

and use them to instantiate a Wing object:[69]

```
>>> w0 = Wing(a0, a1)
```

Next, generate a cubic B-spline curve to act as trajectory curve:

```
>>> T = nurbs.tb.make_linear_curve(Point(), Point(y=3))
>>> T = T.elevate(2)
```

and give it some nonlinear dihedral by translating some or all of its control points in an OpenGL window.[70] Then, create a linear B-spline function, e.g.[71]

```
>>> tw = BSplineFunctionCreator1(end=(0, 30)).fit()
```

that shall "orient" your wing with linear twist:[72]

```
>>> w0.orient(T, Tw=tw)
```

Next, create another B-spline function to specify taper. This time, since we want nonlinear taper, use say, a quadratic B-spline function with 3 control points:

```
>>> Sc = BSplineFunctionCreator1(end=(2, 1), p=2, n=3)
>>> Sc.design()
>>> sc = Sc.fit()
```

[67] See the previous section for a suggestion of parameters that will give you a reasonable fit of the NASA SC(2) 0614 airfoil. [68] Make sure that the "`transform`" method has been called on both Airfoil objects. [69] The wing profile will be a linear interpolation of these two tip airfoils. [70] Do that from a yz view to avoid giving it nonlinear sweep. [71] Currently, Genair defines two other BSplineFunctionCreator* classes; example usage for each one of them are given in Section 4.2. [72] As a rule of thumb, the more nonlinear the twisting function is, the larger "`m`" (the third argument of "`orient`") should be; here, the twisting function is linear so the choice of "`m`" is irrelevant.

The second command will open an OpenGL window where you will be given the opportunity to reshape the default “representation” curve. Pick and translate the middle control point to about $(x, y) = (0.5, 0.5)$. (Alternatively, pick the control point without translating it and type

```
>>> last_picked_object.xyzw = 0.5, 0.5, None, None
```

at the prompt.) At this point you can finally fit (sweep) the wing likewise:

```
>>> w0.fit(scs=(sc,None,sc))
```

As explained in “`w0.fit?`”, the shape of the wing will only approximate the true design intent. As a rule of thumb, the more nonlinear a wing is the less acccurate the approximation will be. The approximation can be improved by increasing the value of the argument “`K`”, but for all intents and purposes this is seldom necessary.

Notice how the trajectory curve (“`w0.T`”) is almost the same as the quarter-chord curve (“`w0.QC`”). This is because the quarter-chord point of the airfoils stored on the Wing object each coincides with the global origin. If you want the trajectory curve to match say, the trailing edge curve, then you must translate the airfoils accordingly:[73]

```
>>> for a in w0.airfoils:
...     xyz = a.nurbs.eval_point(0)
...     a.glue(); a.translate(-xyz)
```

You may change the sweep and dihedral of a Wing object through its “`sweep`” and “`dihedral`” attributes, respectively. These attributes are simple wrappers built around the “`shear`” method of the Part class (Section 4.1.1). Note that the shearing origin is by default the root of the wing’s trajectory curve. Therefore, if you want sweep with respect to say, the trailing edge curve, then you must reassign the “`T`” attribute first, e.g.

```
>>> w0.T = w0.TE
>>> w0.glue(); w0.sweep = 20
```

At any point of the design process you may also reassign the “`half`” attribute of a Wing object. For example, if you are designing a vertical tail then use a value of 0 or 1 to only show the lower or upper portion of the wing surface, respectively.[74]

4.1.4 `WingMerger` Class

The WingMerger class is defined in the “`geom.wing`” module. As explained in

```
>>> WingMerger?
```

[73] Of course, this step must be performed prior to fitting the wing. [74] The attribute merely modifies the list returned by “`_draw`”, not the internal representation of the Wing object (nor of its “child” object(s), if any).

it merges wings one after the other.

The wings can be arbitrarily positioned and oriented in space prior to the merge process. The only requirements are that the connecting tip(s) must share the same airfoil as well as the same chord and twist.

Building on the example of the previous section, first generate a Wing object that is merge-compatible with "`w0`", e.g.

```
>>> w1 = Wing(a1, a0)
>>> w1.orient(T, Tw=tw.reverse(), show=False)
>>> w1.fit(scs=(sc.reverse(),None,sc.reverse()), show=False)
>>> w1.glue(); w1.dihedral = 45
```

Now use the following commands to merge "`w0`" and "`w1`":

```
>>> wm = WingMerger(w0, w1)
>>> wm.merge()
```

and use the "`merged_wings`" attribute of "`wm`" to retrieve the new, merged Wings objects.

4.1.5 `Wingtip` Class

The Wingtip class is defined in the "`geom.wingtip`" module. As explained in

```
>>> geom.wingtip.Wingtip?
```

it creates a wingtip to fill the gap located at the tip of a wing.

Use a Wing object to instantiate a Wingtip object. Continuing the example above:

```
>>> wm0, wm1 = wm.merged_wings
>>> tip = wm1.generate_tip()
>>> tip.fill()
```

See the docstring of the "`fill`" method to learn how to control the shape and length of the wingtip extension.

Note, a Wingtip object is stored by a Wing object on its "`tip`" attribute.

4.1.6 `Fuselage` Class

The Fuselage class is defined in the "`geom.fuselage`" module. As explained in

```
>>> Fuselage?
```

it creates the outer mold line of a fuselage from a bullet-shaped cylinder by successively molding the latter with FFD volume(s).

Unlike most other classes discussed in this chapter, the Fuselage class heavily relies on the interactive features of the OpenGL interface (Section 3.2).

Please refer to the “Intended usage” section of the Fuselage class’ docstring to get pointers on how to use the FuselageMolder and FairingMolder classes.[75] Both of these classes are derived from the FFDVolume class (see Figure 1), so refer to the latter for more information on how to “embed” and “unembed” Point objects, including the control points of NURBSObject objects.

Use the “`nurbs`” attribute of a *Molder object to revert a Fuselage object back to a previous state. For example, the commands:

```
>>> m = fus.molders.pop()
>>> fus.nurbs = m.nurbs
```

discard the changes introduced by the last *Molder object on a Fuselage object called “`fus`”.

Don’t forget to call the “`finalize`” method of your Fuselage object at the end of the molding process!

4.1.7 `Aircraft` Class

The Aircraft class is defined in the “`geom.aircraft`” module. As explained in

```
>>> Aircraft?
```

it is a customized Python dictionary that acts as root for all the Part objects it is composed of.

More specifically, the Aircraft class inherits the properties and methods of both the “`dict`” and Part classes. Hence, an Aircraft object may still be drawn, transformed, clamped, etc. However, its main purpose is to store other Part objects along with their assigned name, e.g.

```
>>> ac = Aircraft(wing=my_wing, fuselage=my_fuselage)
>>> ac.items()
```

Use the “`blowup`” method to quickly resume work from a previously saved Aircraft object.

4.1.8 Miscellanea

Genair can help you define a few more aircraft components other than the ones presented so far. For example, the “`geom.misc`” module defines the Nacelle and WingStructure classes.

Consider, for example, the WingStructure class. As you can see from

```
>>> geom.misc.WingStructure??
```

it defines only two public methods: “`generate_spars`” and “`generate_ribs`”. The first extract u-directional curves from the lower and upper halves of a wing

[75] Tip: when translating the “pilot points” of an FFD volume make sure to be in either one of the xy, yz, or zx views.

before linearly interpolating them.[76] The second takes a similar approach but with v-directional curves.

Note that, similar to a Wingtip object, a WingStructure object can be instantiated directly from a Wing object through the "`generate_structure`" method.

4.2 Examples

The aircraft components described in the previous section are, for the most part, highly flexible. This section gives examples of how this flexibility can be exploited in the context of unconventional aicraft design.

The "`play/examples/`" directory of Genair contains a conventional tube-wing ("`ctw.p`"), a box-wing ("`bw.p`"), a blended-wing-body ("`bwb.p`"), and a strut-braced wing ("`sbw.p`") configuration. All four configurations are saved as Aircraft objects. Make sure to inspect each one of them, e.g.

```
>>> ctw, = load('ctw.p')
>>> ctw.colorize()
>>> draw(ctw, *ctw.symmetrize())
```

and try to pay attention to things like: what trajectory curve was used for each wing segment? which airfoils were used? how many FFD applications did the fuselages have?

You may notice that none of the surfaces are trimmed where components intersect. This is good practice. Indeed, because the geometry produced by Genair is likely to be meshed in an external application anyway, and because the IGES file format is ill-suited to export trimmed geometry, it is best to let the external application do the hard work, i.e. clean up and trim the geometry for you.[77]

You may also notice that the first wing segment of the blended wing-body has a very specific and well delineated planform, and that the B-spline function used to achieve this nonlinear taper is different than the one used to scale the tip airfoils in the vertical direction. In this case both B-spline functions were created from the same "`BSplineFunctionCreator2`" object. For your convenience this object has been stored on the "`Sc`" attribute of the Aircraft object;[78] so, assuming that "`bwb`" is the Aicraft object, you may inspect it like so:

```
>>> bwb.Sc.design()
```

and you may retrieve the B-spline functions with something like:

```
>>> r, n = 1000, 200
>>> scx = bwb.Sc.fit(r, n)
>>> scz = bwb.Sc.fit(r, n, di=2)
```

[76] The "`make_ruled_surface`" function is part of the NURBS toolbox discussed in Section 3.4.
[77] In ANSYS ICEM CFD look for the "Repair Geometry" tab. [78] BSplineFunctionCreator* objects are normally saved as separate pickles.

Finally, both the blended wing-body and the box wing feature corner fillets at wing-wing transitions. Those are actual Wing objects generated with the help of the BSplineFunctionCreator3 class.

A Workflow with Jetstream

Genair can *help* you set up the geometry control system (GCS) implemented in Jetstream. There are in fact many common features between the two: similar to the trajectory curve of the Wing class (Section 4.1.3), the GCS uses axial curves, and similar to the WingMerger class (Section 4.1.4), the GCS uses joints to connect axial curves one after the other.

Note that despite the flexibility of the GCS it is unlikely that the default implementation will satisfy your own needs. For example, the use of axial curves to control FFD volumes is not necessarily recommended. So, as a first warning, *be prepared to tweak the code.*

Also note that the material discussed in this section is for your convenience only. In particular, the implementation of the GCS in Jetstream is capable of more than what you can set up with the utilities provided by Genair. Again, *be prepared to tweak the code.*

One more note: keep in mind that the GCS is ill-suited to handle surface-surface intersections such as wing-body junctions. It can, however, support wings of arbitrary topology, including boxed and braced configurations.

Now that you have better idea of what you are getting into, here is the recommended workflow between Genair and Jetstream:[79]

1. generate the geometry using Genair,
2. generate the grid using ANSYS ICEM CFD,
3. map (fit) the grid using Jetstream,
4. set up the GCS using Genair, and
5. optimize the geometry using Jetstream.

The remainder of this section addresses Step 4.

A.1 Fortran Interface

First off, you need to compile the Fortran I/O interface of the "`opti`" package. Simply navigate to "`opti/io/`" and type "`make`" in your shell. Make sure that the "`FC`" variable in the Makefile points to a valid Fortran compiler.[80]

A.2 Plot3D and Connectivity Files

Following Step 3, you will need these four files: "`newgrid.g`" (typically renamed as "`grid.g`"), "`grid.map`", "`grid.con`", and "`patch.con`".[81]

A good place to put these files is in a subdirectory of "`play/`", and a typical usage session goes like:

[79] Of course, you don't have to use Genair in Step 1, but doing so will *greatly* facilitate Step 4. [80] Tip: the Unix command "`which`" can help you locate the executable. [81] To be precise, "`grid.g`" is not strictly required to set up the GCS, i.e. you could just copy and rename "`grid.map`" to "`grid.g`".

```
>>> gd = Grid()
>>> gd.read_grid()
>>> gd.read_connectivity()
>>> gd.read_map()
>>> gd.read_patch()
>>> save(gd, fn='gd.p')
```

The Grid class is defined in the "`opti.grid`" module. It can be thought of as a visual interface to the grid/connectivity files of Jetstream.

A Grid object has five attributes: "`blk`", "`iface`", "`bcface`", "`ptch`", and "`stch`". These are all Python lists that store Block, Interface, Boundary, Patch, and Stitch objects, respectively.[82] A sixth type of object, Map objects, are stored directly on the "`map`" attribute of the Block objects. Note that all six object types are derived from the NURBSObject classes, so they can all be, for example, rendered in OpenGL window(s). Moreover, each object type implements a "`print_info`" method which is useful to verify information such as boundary conditions and connectivity between patches.[83]

A.3 Geometry Control System

Essentially, a Grid object holds the surface control points that go inside the FFD volumes of the GCS. The next step is thus to define those FFD volumes and, if necessary, their assigned axial curve. For wings, Genair can automate much of this step, *provided that the wings were generated by Genair in the first place* (as suggested in Step 1 above).

Take, for example, the "`wing`" component of the Aircraft object saved in the "`ctw.p`" file of the "`play/examples/`" directory:

```
>>> ctw, = load('ctw.p')
>>> ctw.blowup()
>>> w0, w1 = wing.wings
```

You may easily generate the FFD volume and axial curve for each wing with the "`make_axial_from_wing`" function of the "`opti.axial`" module, e.g.

```
>>> a0 = opti.axial.make_axial_from_wing(w0, 'LE', 2, 1,
...     (10, 4, 2), (3, 3, 1),
...     offsets=[(0.01, 0.01), (0, 0), (0.01, 0.01)])
>>> a1 = opti.axial.make_axial_from_wing(w1, 'LE', 2, 1,
...     (10, 8, 2), (3, 3, 1),
...     offsets=[(0.01, 0.01), (0, 0.07), (0.01, 0.01)])
>>> draw(wing, a0, a1)
```

Next, you should check if the wings fit inside the FFD volumes. Normally, you would do that by embedding the control points of the *patches* stored on a Grid object, i.e.

[82] The order of the objects stored in each list is implied from the connectivity files. [83] Tip: if you don't know the index of an object in a list then pick it from an OpenGL window and type "`last_picked_object.print_info()`" in the prompt.

```
>>> draw(*(gd.ptch+axials))
>>> for a in axials:
...     a.ffd.embed(*gd.ptch)
```

but for this example try embedding the surface control points of the Wing objects instead. Make sure that all the control points are successfully embedded, including those at the tip and at the symmetry plane.[84,85]

Next, proceed with extracting "joints" (called "points" in Jetstream) from the axial curves:

```
>>> axials = [a0, a1]
>>> joints = opti.axial.make_joints_from_axials(axials)
```

after which point is it customary to check the whole setup by visualizing the effect of translating joints on the shape of the axial curves, FFD volumes, and wings:[86]

```
>>> opti.axial.set_module_variables(axials, joints)
>>> draw(*(gd.ptch+axials+joints))
```

(Again, to follow with this example use the Wing objects instead of the grid patches.)

If everything looks fine then the last step is to save the files required to run Jetstream. This can be done with the following commands:

```
>>> opti.axial.write_axial_connectivity()
```

and

```
>>> ffds = [a.ffd for a in axials]
>>> gd.write_patch(ffds)
```

The first command saves the FFD volumes and axial curves in two separate "B-spline" files (".b"). It also saves connectivity information in a format that is analogous to the "`patch.con`" file. As for the "`write_patch`" method of the Grid object, it saves a new "`patch.con`" file that contains additional information such as which embedded surface control point belongs to which FFD volume.

The above commands are decoupled; i.e. you don't need to embed control points prior to calling "`write_axial_connectivity`" and, conversely, you don't need to call "`set_module_variables`" prior to calling "`write_patch`".

In the connectivity file, the "type" of a joint refers to: 0 for a normal joint, 1 for a joint that connects two or more axial curves at their common tip, and 2 for a joint that attaches the tip of an axial curve anywhere along another axial curve. The "`make_joints_from_axials`" utility can automatically detect Type 0 and 1 joints; Type 2 joints must be specified manually.

[84] Here, because "`w0`" has dihedral, you must use `a0.snap_ffd()` to snap the first row of FFD volume control points on the $y = 0$ plane prior to embedding the surface control points. [85] A Point object is successfully embedded if it turns green, except if it is a control point falling on the origin of its NURBSObject object (in which case it won't turn green even though it was successfully embedded). [86] Tip: use the "undo" mechanism (Section 3.2.2) to come back to the initial state.

B Suggested Scripts

B.1 Multi-Segmented Wing

The following script generates and merges an arbitrary number of trapezoidal wing segments. The user must provide the Airfoil objects at each break point. The variable “REF” refers to the curve about which sweep is applied; permissible values are “LE”, “QC”, and “TE”.

```
AIRFOIL = [a0, a1, a2]
TWIST = [0, 0, 0]
CHORD = [5.82, 3.40, 0.78]

SPAN = [4.66, 8.44]
DIHEDRAL = [2, 2]
SWEEP = [30, 30]

REF = ‘LE’

###

class TrapezoidalWing(Wing):

    def __init__(self, a0, a1, t0, t1, c0, c1, s, d, p):

        super(TrapezoidalWing, self).__init__(a0, a1)
        tw = BSplineFunctionCreator1(end=(t0,t1)).fit()
        sc = BSplineFunctionCreator1(end=(c0,c1)).fit()
        T = nurbs.tb.make_linear_curve(Point(), Point(y=s))
        self.orient(T, Tw=tw, show=False)
        self.fit(scs=(sc,None,sc), show=False)
        if REF is not ‘QC’:
            self.T = eval(‘self.’ + REF)
        self.glue(); self.dihedral = d; self.sweep = p

ws = []
for i in xrange(len(AIRFOIL) - 1):
    w = TrapezoidalWing(AIRFOIL[i], AIRFOIL[i+1],
                        TWIST[i], TWIST[i+1],
                        CHORD[i], CHORD[i+1],
                        SPAN[i], DIHEDRAL[i], SWEEP[i])
    ws.append(w)

multi_segmented_wing = WingMerger(*ws)
multi_segmented_wing.merge()
```

References

[1] L. Piegl and W. Tiller, *The NURBS Book*, Springer, 2nd ed., 1997.

[2] L. Piegl, *On NURBS: a survey*, Computer Graphics and Applications, IEEE **11** (1991), no. 1, 55–71.

[3] T. Sederberg and S. Parry, *Free-form deformation of solid geometric models*, ACM SIGGRAPH Computer Graphics, **20** (1986), no. 4, 151–160.

[4] https://www.python.org

[5] http://www.numpy.org

[6] http://www.scipy.org

[7] http://ipython.org

[8] http://pyopengl.sourceforge.net

[9] http://www.pyglet.org

[10] http://www.pythonware.com/products/pil/

[11] http://continuum.io