MASR-like tradespace tool developed using Python: Overview of the Program

Motivation

A multirotor sizing and tradespace tool was built by Georgia Tech's Aerospace Systems Design Laboratory (ASDL) in Microsoft Excel. This tool had an interface with the CAD software CATIA to facilitate the export of designs for modeling and subsequent 3D printing. The original choice of building the tool in Excel was driven by a relatively low upfront cost to start the project, since the researchers at ASDL were already familiar with Excel and VBA programming. After the initial implementation, however, it became clear that Excel was not the ideal platform for the application, mainly due to relatively slow execution times and a program structure that was not conducive to additions and/or changes to the application. For those reasons, researchers at the Army Research Laboratory (ARL) Vehicle Technology Directorate (VTD) Vehicle Applied Research Division (VTV) decided to move the tool into a modern programming language, which would ideally solve the problem of slow execution speed and also enable future researchers to expand upon the application more easily.

Choice of language

Once it was determined that the ASDL multirotor sizing and tradespace tool should be rewritten, a language for the new application needed to be chosen. The first requirement identified was that the language needed to support object-oriented programming. An object-oriented approach is a natural choice for this problem because there are many vehicle components (and in fact the vehicles themselves) that have attributes defining their properties. For example, from a very early stage it was clear that a quadrotor (object) could have attributes such as a motor, propeller, battery, etc. all of which could be objects. Candidate languages that supported object-oriented programming were C++, Java, Python, Ruby, and Pearl, and MATLAB. In addition to supporting the object-oriented paradigm, the language needed to be relatively common in order to minimize project-entry overhead for future collaborators and transition partners. This reduced the list to C++, Java, Python, and MATLAB (the others were deemed to be less common among the aerospace and systems engineering communities). The final down-select was based on the criteria that the final language should be as readable as possible for a wide range of developer skill levels and language backgrounds, while maintaining as much inherent language power as possible. All things considered, Python was chosen.

Program structure

Even before Python was selected, it was decided that the result of the effort should be an application that could be run from the command line of any Unix-based or Windows machine. Since Python files can be run from the command line (or a host of other ways) on any machine with an installed Python distribution, this goal was made easy by the choice of language.

Graphical User Interface

The front-end of the tradespace tool is a graphical user interface (GUI) written using Python's standard GUI package *Tkinter* and the Python module *Tkinter.ttk*, which provides access to the *Tk* themed widget set. (Note that *Tkinter* has been renamed *tkinter* in Python 3. This application was written using Python 2.X syntax.) Both the main GUI screen which appears on program launch, and subsequent toplevel windows were written using *Tk*, *Tkinter*, and *ttk*. When available, *ttk* themed widgets were used in favor of their *Tkinter* counterparts, supporting portability between operating systems. The GUI was written using an object-oriented structure. For instance, the main frame held within the root window in which all other widgets are placed was implemented as the *oo_quad_GUI.QuadGUI* class, which is a *ttk.Frame* (i.e., it inherits the *ttk.Frame* class). Theoretically, multiple instances of the application could be opened at the same time simply by creating a new instance of *oo_quad_GUI.QuadGUI* within a new root window. Toplevel windows and most major frames were implemented as classes inheriting *Tkinter.Toplevel* and *ttk.Frame* respectively.

Vehicle component and quadrotor classes

In addition to the GUI, there are several other major aspects of the program. In the original Excel version written by ASDL, vehicle components were described using numerical or string attributes. For example, a propeller is described by a name, weight, diameter, pitch, cost, and number of blades. This type of decomposition lends itself to an object-oriented approach. The following table shows vehicle components, their associated class name, and the Python module the class was written in:

Component	Class Name	Module Name
battery	Battery	battery
motor	Motor	motor
propeller	Propeller	propeller
prop/motor combo	Propmotorcombo	propmotorcombo
3D printer	Printer	printer
printing material	Printingmaterial	printingmaterial
laser cutter	Cutter	cutter
cutting material	Cuttingmaterial	cuttingmaterial

Therefore, the module *battery.py* contains the class *Battery*, which is the blueprint for a battery object. (For reference, Python uses the common "period" notation to denote an object and an attribute object associated with that object, e.g., *battery.Battery.__init__* is the full "pathname" for the Battery class constructor method *__init__*. Further information on Python syntax can be found easily online.)

With component classes defined, a full quadrotor class was written. In this application a quadrotor was defined using the following components: prop/motor combo, motor, battery, printing material, and cutting material. Using the attributes of these components combined with a sizing and performance algorithm, the performance of the entire quadrotor can be computed.

Database management

So far the main GUI, component classes, and the quadrotor class have been described. The next crucial portion of the program is the database management code. In order to save component information from one application run to the next (it would be inconvenient to enter component specifications on each application launch), it was necessary to store component data entered by the user in a database. Python has a built-in module called *shelve* which defines the behavior for a *shelf* object. A *shelf* is a persistent, dictionary-like object that can hold other Python objects that have been serialized using the Python *pickle* module. A shelf object is effectively a database that can store the component objects for retrieval at any time. A database file named *this_filename* is created the first time the *shelve.open(filename=this_filename)* method is called. Since the initial application development involved creation of these databases, subsequent users should not have to create any new databases. An exception to this would be if a later change in the code results in additional component(s) added to the quadrotor.

The built-in *shelve* module was chosen in favor of a more powerful and efficient database management system (e.g., SQLite) because of the ease of implementation (it can store Python objects directly using *pickle*) and the relatively small number of objects to be stored in the databases. Note that *shelve* is not a good method for storing a large number of Python objects.

All code relating to database management is contained within the *dbmanagement* module and the *oo_quad_GUI.DataMgtFrame* class. An instance of *oo_quad_GUI.DataMgtFrame* is contained in the main GUI (located in the upper right hand corner upon launch). It contains a list of components from which the user can select one component type. On clicking the *View Database* button below the list, the appropriate database is displayed using code contained in the *dbmanagement* module. Various tasks can be performed once a component database is opened, such as adding a component instance to the database, deleting an existing component from the database, and editing components that have been previously defined and saved in the database. Multiple database widows may be open at the same time.

Generating alternatives

The final major part of the program is the process of generating vehicle alternatives, calculating the performance of those alternatives, and scoring feasible alternatives (i.e., alternatives that do not violate any sizing or performance constraints defined by the user). Code having to do with this process is contained within the *alternatives* module and the *oo_quad_GUI.AlternativesFrame* class, which has several slave widgets during execution including instances of *oo_quad_GUI.VertScrolledFrame*, *oo_quad_GUI.AlternativesSheet*, and *oo_quad_GUI.ViewQuadDetails*. The alternatives frame is located in the lower half of the main GUI window.

The number of components currently entered into the databases for use in vehicle alternative generation is relatively small, therefore it is reasonable to generate all possible combinations of parts. Currently a call to *oo_quad_GUI.AlternativesFrame.find_alternatives* with zero constraints (i.e., maximum number of combinations) takes approximately 0.05 seconds. It seems improbable that there would come a point where the number of available component combinations would make a full-factorial computation impractical, however, if this were to occur a more sophisticated design of experiments could be performed. The current flow of alternatives generation is as follows (and can be found in the code within the *oo_quad_GUI.AlternativesFrame.find_alternatives* method):

- The current user-defined constraints are retrieved from the GUI entries
- Constraints are passed to the *alternatives.generate_alternatives* function
    - *alternatives.generate_alternatives* loops through all possible alternatives using combinations of the parts currently found in the component databases
    - Each quadrotor alternative is passed to the *alternatives.is_feasible* function, which is the sizing and performance algorithm
        - *alternatives.is_feasible* determines whether the quadrotor alternative is feasible. If it is feasible it returns the alternative's performance metrics.
    - All alternatives are returned to *oo_quad_GUI.AlternativesFrame.find_alternatives*, whether feasible or not.
- Feasible alternatives (i.e., alternatives for which *quadrotor.feasible* is True) are filtered out of the full alternatives list.
- The list of feasible alternatives is passed to *alternatives.score_alternatives*.
    - *alternatives.score_alternatives* scores all of the feasible alternatives using their performance metrics and user-defined performance metric importance ratings.
- The list of scored feasible alternatives is displayed for the user in the main GUI window.

Note that all of the alternatives are returned from *alternatives.generate_alternatives*, whether feasible or not. Therefore, more advanced statistical analysis can be performed on both infeasible and feasible alternatives. For developer convenience, the reason for an infeasible alternative's failure of *alternatives.is_feasible* is stored in the quadrotor attribute *quadrotor.feasible* (see the quadrotor class for more information on the feasible attribute).

The *displayable.Displayable* class

An important feature of the GUI code is the ability to display vehicle components (i.e., their specifications and performance) and quadrotors. Due to the fact that all vehicle components were implemented as objects containing attributes, a general method of displaying objects was developed. This method is defined within the *displayable.Displayable* class. All quadrotor components, as well as quadrotors themselves, inherit the *displayable.Displayable* class. Therefore a *ttk.Frame* containing entries describing a component's attributes can be obtained simply by calling the component class's inherited *display_frame* method (generally inherited from *displayable.Displayable*, although occasionally overridden). This additional inherited class provides a significant improvement in code readability within the *oo_quad_GUI* and *dbmanagement* modules. For more information on the *displayable.Displayable* class refer to the code.

The *unitconversion* module

At different points in the code it was necessary to perform various unit conversions. To make this as simple as possible, the *unitconversion* module was created. The *unitconversion.convert_unit* function takes in a starting value, a starting unit, and an ending unit, and returns an ending value. Currently this

function accepts a select number of units, those relevant to the application. This function can be updated at a later time if more units need to be added.

Future work

While the current Python tradespace tool has most of the functionality of the original MASR tool, not everything has been added. For instance, the Python tool does not have a way to interact with a CAD package such as CATIA. If the ability to push quadrotor designs to a CAD package for visual confirmation of design, or for the purpose of creating a file to 3D print the part, is desired, it would need to be added to the program. A second potential area for future work is to incorporate an analytic quadrotor hover and forward flight model into the sizing and performance algorithm. This would allow the user to input more complicated mission profiles. Currently the tradespace tool can only calculate the required thrust for hover based on the weight of the vehicle plus some margin to account for a desired level of maneuverability.

There are also many ways to add tradespace exploration functionality to the tool. For example, one could add the capability to start the analysis with a desired set of performance metrics and a quadrotor with one free variable (e.g., a battery with an arbitrary capacity). From there it would be possible to run the sizing and performance code “in reverse” and obtain the capacity needed to achieve the desired performance. In many situations no solution would be possible using this method, since changing the capacity (all other things being equal) would not improve metrics like payload capacity or maximum speed. On the other hand, the specifications of the theoretical battery required to achieve a specific vehicle endurance could be obtained in this way.