|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Team n. 17 | Name | Student Number | Name | Student Number |
| Nelson Johnson | 4195787 | Şan Kılkış | 4192028 |
| Utilizing KBE to Frontload Preliminary UAV Design | | | | |
| Describe briefly your design case and make sure to address the following crucial questions:   * What is the design challenge your KBE app is supposed to tackle? * Why the use of KBE is supposed to be a good means to address such challenge? Thus, what are the characteristics of the problem at hand that match the strengths of KBE technology?   Imagine for a second that you work at a company specialized in producing light UAVs (1-20 kg MTOW) for multiple use-cases, and to be competitive, you desire agility and quick project proposals to cope with changing demands of the market. However, the challenge of exploring the design space and analyzing the feasibility of potential designs is a lengthy procedure as experienced from the DSE project. This process requires re-use of corporate knowledge from previous designs and application of repetitive statistical and empirical relations (rules) to aid in generation of preliminary models (geometry) which are then analyzed to obtain actual performance data. However, in the early stages of a design the overall geometry can change tremendously and without automation, the time it takes to analyze the design space will be a product of the time it takes for one preliminary design with the total number of designs to be analyzed. Thus, in a situation where time is limited (such as in the DSE or for project proposals) the depth to which these designs can be explored is constrained and often only qualitative information can be used to assess the strengths and weaknesses of the preliminary designs. Ultimately, without a streamlined process for this analysis, costly mistakes can be made due to the chosen design either not being feasible or not capable of fulfilling the customer’s requirements.    Figure 1: Illustration of How Conceptual and Preliminary Design Allocate Costs (La Rocca, 2018)  Current applications of KBE and KBSs are typically for detail design, yet most funds to a project are allocated during the preliminary design as illustrated by Figure 1. Thus, to be able to frontload this procedure, and free-up resources for more creative work, our KBE app will parametrically generate geometry based on a harmony between Class I and bottoms-up Class II weight estimation, and requirements of different ranges, payload masses and use cases. Due to the tremendous amount of work required to code all the capability modules, the current state of the code integrates these components for the most popular twin-boom aft-tail configuration on the market today. However, the integration procedure can be easily expanded by adding more component modules and combining them in new ways due to the use of primitive class definitions. Furthermore, the code was written with agility and flexibility in mind, thus class methods do not depend on singular object instances, and instead can cope with multiple motors, wings, payloads, etc.  The use cases include changes in the payload mass, payload size and other mission specific requirements such as hand launch-ability. The weight estimations are very repetitive rule-based processes, which can take a lot of time in the conceptual design phase. Especially when an input such as payload mass changes, the entire Class I, Class II, and tail sizing be performed again. KBE is a great way to deal with these repetitive rule-based calculations and to visualize the geometry afterward. This KBE app will allow the generation of all the major components of a UAV as well as its sub-components such as the required battery, motor and propeller sizes. The battery is sized from the range/endurance requirement and an estimation of the drag in cruise from the flight mechanics lectures. Our App will export the current design’s variables in an excel file and the geometry in an .step file. This presents a tremendous advantage to a designer, since they can utilize the output state variables for further detail design, and even run CFD analysis on the output UAV to determine problem areas for aerodynamics.  If the initial fixed-wing demonstrator is successful, due to the object-oriented programming paradigm, this KBE app is open for future enhancement through adding analysis functionalities such as the ability to model alternative configurations such as canards, flying wings, BWB’s, and even multi-rotors. If granted access, we can also connect higher fidelity aerodynamic solvers to calculate the entire aircraft’s aerodynamic characteristics. Finally, to present as a visual aid for the type of designs which the KBE will be valid for, Figure 2a and 2b depict the lower and upper limits of maximum take-off weight respectively.       |  |  | | --- | --- | | Figure 2a: RQ-11 Raven (MTOW 1.9 kg) | Figure 2b: Penguin C (MTOW 22.5 kg) | | | | | |

|  |
| --- |
| Rule based parametric model requirements |
| Describe here the main functionalities of the rule based parametric model, which will be the core of your KBE application   * What systems components/features will be included in the parametric model? * What are the main parameters used to define and control the various components/features? * What are the main (engineering) rules governing the definition/interface of the various model components? Identify the main sources you will use to capture knowledge * How will your app deal with rules violation? (warnings, automatic corrections, change suggestions)   **------------------------ max 2 pages including explanatory figures ------------------------------**  In the ParaPy GUI tree, all main components of a typical UAV will be present, such as the wing, vertical and horizontal stabilizer, and fuselage along with all subcomponents such as parametrically selected/sized payload, propeller, motor, battery, and connecting booms. The payload shapes are based on reference cameras for drones.  All these component primitives will dynamically upscale to represent the change in requirements based on engineering rules encoded into their respective classes. The main parameters controlling the components are the user input requirements such as range/endurance, MTOW and the choice of hand launched or not. Utilizing this approach, additional aircraft configurations will be possible to implement in the future since they will, for example, use the same wing, fuselage, and tail classes but orient them in different ways and alter the integration methods.  The parameters which size the wing area are from a class I weight estimation, while those for the tailplane area being from the bottom up weight estimation. The bottom up weight estimation uses component surface areas and an assumed number of carbon-fiber prepreg plys (varies per component) to estimate the mass of each component. The fuselage is sized to fit Bounding-Boxes (bboxes) of the parts which go inside the fuselage with frames that scale with the size of the internals.  Furthermore, the majority of the knowledge that will be used was either presented during the bachelor, obtained from literature reports and reference UAV’s, or generated during the DSE for a project on a modular UAV for surveillance. The utilization for this knowledge will be distributed within each primitive. As is the case with KBE systems, the inference mechanism and rules are scattered and hard to centralize. Thus, each individual primitive will encompass its own set of sizing rules and weight estimations. An overview of some of these rules are provided below:  **Global Aircraft**  The global aircraft class (UAV root class) has the responsibility of loading customer requirements either through manual GUI input or through the provided excel sheet and performing the class I weight estimation to obtain the parameters for the rest of the design. These inputs include the thrust and wing loading, MTOW, payload mass, aspect ratio, maximum lift coefficient, and stall speed. The UAV root class will then pass the relevant inputs to the following component class instantiations such as wing, scissor plot (tail sizing), battery, etc. In addition, it calculates the output performance of the UAV with through use of class methods and another capability module ‘Performance’. The procedure is then to use the methods ‘weight\_and\_balance’ that calculates the resultant C.G. as well as weight of the UAV, and ‘sum\_area’ which calculates the total wetted area of the UAV to be used for a zero-lift drag estimation. These are then fed to the ‘Performance” capability module which plots the power required and power available with the actual CD0 to verify our range/endurance requirement.  **LiftingSurface**  This is a primitive class which all flight surfaces (main wing, HT, VT) inherit from. This creates a tapered, swept wing surface from an input wing area, aspect ratio, taper, dihedral, twist, sweep and user airfoil choice.  **Wing**  Once top-level design parameters are defined by the global UAV class, a wing object can be instantiated based on the required wing loading, MTOW and initial guesses from best engineering practice for the airfoil choice, sweep (if necessary), dihedral angle, and taper ratio. If triggered by another class or the end-user, an attribute containing an AVL simulation will return the current performance parameters of the wing. The user may change the wing sizing inputs/parameters in the GUI and then re-evaluate the wing’s performance. The rules used can be found in our knowledge base.  **Horizontal Tail**  At runtime, the HT is instantiated in the main UAV class, but it’s contribution to the CG is not yet accounted for. This causes the HT at runtime to be incorrectly sized because the scissor plot takes the wrong input CG. This is fixed by writing a loop in the UAV class to (re) calculate the CG and (re) size the tail for this updated CG. With the new tail size, the CG is recalculated with and this loop runs until the two CG’s converge. The HT aspect ratio is chosen from statistics.  **Vertical Tails**  Utilizes a simple empirical relation to size the VT. This uses statistical VT data to estimate the tail volume coefficient and thus the VT surface area after supplying a tail arm.  **Fuselage**  This was by far the most difficult class to program due to procedural generation. The idea behind this class is to automatically generate a fuselage based only on the input of components that should be placed inside. The essence of what it does is that it has a forward-looking algorithm, that per “container”, fuses the BBox (bounding-box) of each part inside to obtain a required width and height at each section. It then compares this dimension to the dimension of the section that comes after it to try and detect when the apex or location of maximum thickness or width occurs. These parameters then determine if the frame used for lofting is placed before or after the part. As you can see from the figure below, the largest part of this flying wing fuselage is due to the blue wing bbox. The part of the wing that should be inside the fuselage is cut and then that bbox is used to size the local fuselage frames. The algorithm successfully detects that this is the largest part, thus when looking at the fuselage frames in yellow, it is possible to see that before the wing-section the frame defining the size of the payload (EOIR sensor) is built in front of the sensor. However, after the wing section the fuselage frame is built on the rear section of parts. This ensures that the bounding-boxes of all parts are contained inside the fuselage. Also, a feature that we are most proud of is that a nose or tail cone can be generated that perfectly blends into the curves of the fuselage. Furthermore, a caching system enables remote changes to the fuselage frames that are then re-injected into ParaPy to enable manual input to this automated class. This is especially useful for cases when the automated fuselage generator fails to produce the result the user desires. The output of this fuselage generator is depicted in Figure 3.    Figure 3: Automated Fuselage Generation based on Internal Shapes  To not break the parametric model of ParaPy, a comprehensive value checking algorithm must be placed. These will trigger warnings or error messages if certain values are deemed outside of a suitable range. For example, if the payload mass supplied by the user is a negative value. Furthermore, whenever best engineering practice is violated the user might be alerted by a warning message. An example would be a warning to reduce a high value of sweep for a conventional configuration since it is not needed for the flight regime. Also, we have validators on all of our inputs, which will block this value and return an error in the console if an unacceptable input is supplied. |
| Internal Analysis Capabilities |
| * What analysis modules will be implemented **inside** your KBE application, thus coded in ParaPy. At least one internal analysis module should be present.   **------------------------ max half page ------------------------------**  The following is a list of most internal analysis capabilities within our KBE application:   * Design Parameter Creation and Design Point Selection * Battery Sizing * Stability Parameter Generation (Scissor-Plot) * CG and Tail Size Convergence * Camera selection * Motor Selection * Propeller Selection algorithm * Electronics Selection * Weight and Balance * Wetted Area 🡪 Parasitic Drag Coefficient * Performance Estimation   The first internal analysis capability is taking requirements from the user excel sheet and transforming them into initial design parameters for the sizing of all component parts in the aircraft model. After a weight is established from the Class I weight estimation, the feasible design space is then analyzed in the wing loading and power loading domain and the design point is established based on constraints. These constraints reflect customer requirements such as hand-launchabilty. This then kick-starts the design process by determining a suitable payload and sizing the battery based on range and endurance relations.    Figure 4: Design Point Selection from a Constrained Loading Diagram  The horizontal tail sizing scripts generate a scissor plot based on the current design, yielding the required tail surface area. This has been coded for canards and conventional aft-tail configurations.    Figure 5: Determination of Minimum Required Tail-Surface Area Utilizing a Scissor Plot  Within the KBE app, the center of gravity as well as surface area for various components (wings, fuselage, payloads, batteries, avionics, engines) are calculated using built in ParaPy attributes. These are then used to estimate the C.G. location when combined with component masses from bottoms up weight estimation. Further, this CG determination is used in a loop together with the tail sizing to reach a converged CG and tail size. The external surface areas are also used for a zero-lift drag estimation based on component surface areas.  The speed controller of the electronics module statistically estimates the mass and volume based on the chosen motor’s recommended amperage draw. Also, the number of speed controllers increases with the number of motors instantiated. Based on the estimated power-requirement established from the power-loading diagram, a motor is selected from a database of RimFire motors. Therefore, the main input of this class is target power which represents the maximum shaft-power of the motor. The selection algorithm then selects a suitably powerful motor based on ‘.csv’ input files in the database directory. With this implementation, if need-be, the motor database can be expanded later. The database .csv files also specify a range of allowed propellers as well as the recommended ESC rating in Amps.  Based on the motor selection defined previously, the Propeller class takes the motor that it is to be attached to as input, as well as the design speed for which it should be optimized. These inputs are used to generate a propeller at the specified motor, and it’s allowed propeller range attribute is used to trigger a selection algorithm which picks the most efficient propeller from over 400 propeller ‘.dat’ files. This is accomplished by parsing the string of the allowed propeller range, and based on these values, looking through only the header of each .dat to create a list of allowed propellers. This preserves the lazy evaluation characteristic of KBE and increases the efficiency of the code by not loading the entire data of all 400 simulation files. Thus, only the range of propellers that are allowed by the motor manufacturer is loaded into memory and the simulation data is used to evaluate per flight velocity, the RPM which corresponds to the highest efficiency. Then, as a second pass, the algorithm selects the highest efficiency at the desired design speed. The result of this process is shown below.    Figure 6: Optimum Propeller Selection Result Based on Simulation Data  Even though the general payload module is supposed to encompass a wide variety in the future, the current version of our KBE only provides the option to instantiate the UAV with an EOIR gimballed sensor (Surveillance UAV assumption). Thus, the payload target mass which trickles down from the main UAV class is utilized to select a suitable EO/IR sensor from a handful of ‘.csv’ files containing the outer-dimensions as well as specifications of each sensor. Currently, this is done by a weight-based algorithm, but in the future, a customer requirement for ID range can also be utilized. Due to this class inheritance, all attributes that are required to define a component are also included here.  As described above, the fuselage class must analyze all internal shapes which are forced to be inside of it, then it must create frames which fit the internals.  The final performance of the UAV is also analyzed. This is done by plotting the required power and available power vs airspeed. The continuous as well as burst power, are multiplied by the propeller efficiency function to produce the available power curves. This allows one to obtain the ideal speeds for flight. As often is the case with drones of this size, the ideal speeds can be too close to the stall-speed, therefore this condition is checked, and a safety buffer is added. Finally, the range and endurance are estimated from the required power in this plot to verify the top level corresponding requirement.    Figure 7: Result of Analyzing the Final Design w/ New Stall Speed and Propeller Data |
| Link(s) with external analysis module(s) |
| * What **external** analysis module will be connected to your KBE app? How will your app interact with such applications? At least one external analysis tool/module should be present.   **------------------------ max half page ------------------------------**  The current wing’s lift gradient and pitching moment are analyzed by AVL and passed to the equations for the scissor plot in HT sizing. With changes to the wing design, this analysis is automatically re-run due to the dependence of the tail and thus causes the tail size to update accordingly. All parameters from the AVL run however, return as an unsorted dictionary. Thus, each run-case is sorted based on the Angle of Attack and is stored nicely for easy access later. In later iterations of the app, this data-set can be used to even enhance the weight estimation method by computing loads and then translating them into a full-fidelity wing model.    Figure 7: AVL Geometry that is Viewable from the GUI |

|  |
| --- |
| Input data handling capabilities |
| * What data sets will be provided as input to your KBE app? * In which format will the input data sets be defined? * Which data (sub)set will be interactively editable in the ParaPy GUI?   **------------------------ max half page ------------------------------**  As mentioned before, an excel file including the mission requirements on Range/Endurance, MTOW or Payload Mass and hand-launchability is the active data set for the KBE app, which can also be set within the GUI.  A set of airfoil ‘.dat’ files will be supplied to the KBE app. There are cambered, reflexed and symmetric airfoils in the database. Symmetric sections will be used for the VT and HTs while a cambered airfoil for the conventional wings. The airfoil can be changed in the GUI.  In a similar fashion to the airfoil ‘.dat’ files but much more complex, the propeller ‘.dat’ files contain simulation information regarding efficiencies at all RPM ranges and velocities in the flight envelope.  Furthermore, the motor and payload components are read into the app through a custom .csv parser. |
| Output data reporting capabilities |
| What output files is the KBE app supposed to generate and in what format?  At least one STEP(or IGES) file and one output file containing results from the analysis modules.  **------------------------ max half page ------------------------------**  The KBE app will output a ‘.step’ file with the aircraft ready for CAD/CFD import. Due to the AVL wrapper used, all wing-strip forces are also output to .json files. Furthermore, an output an excel file is written with the relevant aircraft dimensions and performance will be output. Finally, all plots generated from the GUI are saved to the user directory in .pdf format for easy and beautiful integration in LaTeX documents for reporting to the client. |