ESP Design Document

Nathan Alday

June 10, 2011

Chapter 1

Philosophy

1.1 Introduction

ESP is intended to be a educational simulation framework that focuses on being easy to both use and understand at the source level for undergraduate level students and more advanced users. ESP is intended to be primarily a teaching tool that also provides a universally available modelling and simulation toolbox that is both effective and accessible for real world applications.

1.2 Objectives

ESP's design objectives are, in order:

Pedagogy The primary purpose of ESP to teach how modelling and simulation works, how to apply good programming and documenting practices to modeling and simulation projects and how to actually write/modify simulations.

Utility/Hackability/Accessibility After pedagogy, utility is ESP next highest goal. To maximize its utility, a modeling and simulation code must usefully address a wide variety real world problems. It must also be easily modified and extended to fit the user's needs. And it must be accessible, both from an ease of use standpoint and easily obtained.

Accuracy The code should be as bug free as possible and the results should be as accurate as possible.

Scalability ESP must be easily scalable to utilize and demonstrate the power of distributed computing as well as teach the principles of designing scalable modelling and simulation software. This includes being scalable across both processors and computers.

Performance ESP should be as performant as possible while still meeting the higher objectives.

To best meet these objectives, ESP must be open source and written primarily in an accessible, powerful language that easily interfaces with other languages. We choose Python for its batteries included approach, growing popularity in the modelling and simulation domain, ease of use and accessibility, cross platform portability and large and well supported open source projects and community.

1.3 Design Overview

For pedagogy, modifiability and scalability, ESP must be modular both at the process level and at the module and function levels. As a result, the framework will consist of three distinct parts, each of which has a corresponding Python module (in parentheses): the numerical and modelling libraries (libs), a process level intermodel coordination program (controller) and a set of data visualization, data references and analysis utilities (utils).

Additionally, clear documentation, particularly of the numerical and modelling libraries, is a central part of the design. The code and its comments should clearly explain and demonstrate each model and numerical method. Furthermore, each model and numerical method as well as each class and function exposed in the libs API should have a corresponding document explaining its functionality and how and why it is used in a pedagogical fashion. Most likely, this will be in a wiki format.

To maximize utility, interfaces for the numerical and modelling libraries will be provided in as many languages as is practical.

1.3.1 The Numerical and Modelling Libraries

The numerical and modelling libraries are the core of ESP. They will implement both numerical routines such as linear algebra routines, integrators, useful numerical types, system solvers and optimizers as well as simulation models such as aerodynamics models, finite element models and logistics models.

The each model will be accessible through at least three submodules, each with a different tradeoff between the pedagogy, accessibility and performance.

standard The standard module is the canonical module. All numerical routines and models in ESP will have an actual implementation in the standard module. Its sole objective is to teach through demonstration both how to implement a routine or model and how it works. Clarity, accesssibility, correctness and modularity take precedence over performance. Routines and models in the standard module should have no dependencies other than the Python interpretter and its default modules. They should also be as type agnostic as possible (i.e. able to operate on multiple classes in the numeric hierarchy such as decimal and float as well as

external types such as mpmath's mp and iv types). The explanation of routines and modules in the documentation should focus on the standard module implementations.

fast The fast module should consist of reimplementations of the routines and models from the standard module with a greater emphasis on performance. Written with an emphasis of performance over clarity, routines and models should still be written in pure Python, but may use outside, easily accessible and open source pure Python dependencies (like mpmath and numpy) while maintaining an API that matches the standard module API, but may contain extra keyword and variable length arguments. Routines and models that haven't been reimplemented should remain accessible through the fast module and simple call their standard model versions.

faster The faster module should consist of reimplemenations of the routines and models from the fast module (and thus the standard module) with an outright emphasis on performance. Routines and modules in faster can call non python code and depend on non python libraries, code and build tools. Like fast, the API of faster must replicate standard, but may contain additional variable length and keyword arguments. When a class or function isn't implemented specifically in faster, it should call or reference its counterpart in fast.

In all cases, reasons for and explanations of any of the optimizations should be explained in the code.

Individual models will be able to import the numerical and modelling libraries, all of which will be threadsafe. All functions will be purely functional to the greatest extent practical. To allow for generic types, single argument numerical functions like square root will attempt to call that relevant method (sqrt()) method.

1.3.2 The Controller

The controller program coordinates and controls the order and rates at which models are run as well as maintains, shares and collects interprocess and global state, all at the process level. The prime purpose of the controller program is to structure model programs together to create a simulation. It will also be able to communicate across networked computers. The controller program will communicate with the model programs via a set of function calls that allow the programs to register with the controller, respond to controller commands to run for a cycle, shutdown, copy in new global state and copy out local state. These functions will be located in the controller module.

Furthermore, the controller program will accept user input from standard in and output its status to standard out, allowing it to be controlled via the commandline, through an input file driven batch mode and possibly be driven by a GUI.

The controller program instances will define and enforce the intermodel topology by acting as nodes on a tree. True model programs will be leaves (childless nodes) on the tree. One controller program instance will be the root node, while other controller program instances may be child nodes with further controller programs and/or models as their children. Controller programs that are child nodes will respond to run commands from their parents as if they were models and will accept and return state as if they were models.

The controller/model interface will be treated as purely functional. The model programs should (in a particular execution) return the same output for a given input at all times.

The controller program will be able to operate in at least 4 modes:

Generic The generic mode will connect with various models across both local and networked machines, and run them a specified number of times in a specified order. Each model will have "run order number" – an integer that tells the controller in what order to run the registered models. Before the first run command, the controller will set the inputs of the group 0 models, which will then run. The group 1 models will follow after the controller updates their inputs with the global state, including the outputs of the group 0 models. Then the group 2 models do the same, etc. Negative group numbers will correspond to indexing the models last to first, like Python indices. So, group -1 will run last, group -2 will run second to last, etc.

The controller will repeate the cycle either a fixed number of times as commanded or until it receives a stop request from a model, its parent controller or standard in. It will then command the models to shutdown and then exit itself.

Time step The time step mode is identical to the generic model except that it will track and increment a time state variable. It will be able vary how much the time variable is incremented by and will be able to rerun a timestep to accomodate variable timestep integrators. The time step mode will be able to run for a designated time period in addition to the generic mode's run conditions.

Solver The solver mode will treat the models as a system of equations to be solved and will be able to calculate derivatives and rerun models. The solver mode will run until it solves the system within some tolerance. The solver will assume that the inputs are continuous, real variables. Otherwise, the solver mode will act identically to the generic mode.

Optimizer The optimizer mode will be similar to the solver mode, but instead of solving a system equations, will instead optimize an objective function state variable according to a set of constraint state variables.

All of the modes other than the generic model can be implemented by calling one or two models that run at the end of the run ordering. For instance, time mode can be implemented simply by having a single model

in group -1 that increments the time state and stores the previous state in case the time step changes.

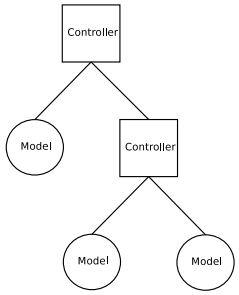

Figure 1.1: Example controller/model topology

1.3.3 The Utilities

The utilities module will provide interfaces for data analysis and visualization tools, such as matplotlib, as well as a database on publically available data for commonly modelled componets such as engines and structural beams. It's a catch all module for anything that is useful to modelling and simulation that is not included in the other modules.

Chapter 2

Detailed Design

The propsed general design will be tested by implementing a prototype (ESP version 0.1.0, using semantic version numbering), that solves a particular example problem, specifically a 3 degree of freedom (3dof) rocket simulation. The first few iterations of ESP will utilitize this approach.

2.1 Version 0.1 Design

Version 0.1 will implement all of the components of the ESP design necessary to simulate a multistage rocket launch to orbit using a 3 degree of simulation (X,Y,Z position). These components include the following models:

Dynamics Model A simple 3dof simulation of the rocket state and its component parts will intialize at a velocity and position dictated by the earth's rotation and the rocket's starting lattitude, longitude and altitude. The rocket's position and velocity will then be integrated over time responding to applied forces from gravitational, aerodynamic and engine forces. The rocket's mass will be calculated from a starting mass and a fuel mass that is integrated over time according the the mass flowrate dictated by the engine model.

Aerodynamics Model An aerodynamics model will calculate drag on the rocket as a function of Mach number and atmospheric density.

Atmospheric Model An atmosphere model will calculate air density according to the rocket's position.

Gravity Model A model of the earth's gravitational field and its effects on the rocket.

Engine Model A model that calculates thrust from the engines and the direction of that thrust applied to the rocket.

Engine Controller Model A model containing the navigation commands (as engine thrust angles) to the rocket. The model will consist of a set of engine thrust angles at a discrete set of time intervals with cubic splines in between. (Unless theres a better way to do this)

To implement the these models, the following standard module numerical routines and typeswill be implemented:

Vector Type A generic vector type with addition, substraction, dot and cross products and scaling will be implemented. A norm method will also be implemented.

ODE Solver An ODE solver will be implemented with interfaces that both return a single time step increment as well as a list of values given a list of time steps. The solver will use the vector type for vectors and generic number types for numbers.

Unconstrained optimizer An uncontrained optimizer will be implemented that can optimize the thrust angle nodes.

Horner's method Horner's method of evaluating polynomials for quick calculation of the spline values.

Linear and Quadratic Interpolation For modelling the engine specific impulse as a function of altitude as well as modelling the atmosphere.

Cubic splines For modelling thrust direction.

Look up tables For the atmosphere model.

To coordinate and control these models, the generic, optimization and time modes of the controller program will be implemented. The optimization and time modes will be implemented using models that run last and second to last in a controller cycle. The time mode will be used to advance the time in the dynamics model.

The simulation will be laid out as in Figure 2.1. Figure 2.2 shows the layout of the simulation with the timer, objective function and optimizer models connected to the generic controller programs, explicitly showing how the controller programs will be implemented with respect to the rest of the sim. The numbers are the order numbers which determine the order in which the executables will run.

2.2 Version 0.1 API

class standard.**Vector**
    Generic Vector class built on top of lists.

    row

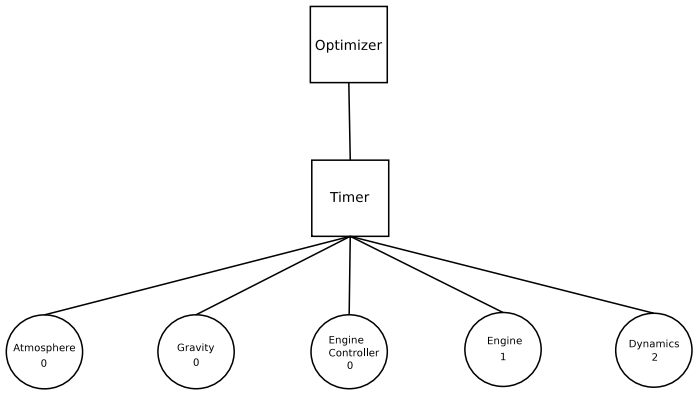

Figure 2.1: Version 0.1 3dof sim topology

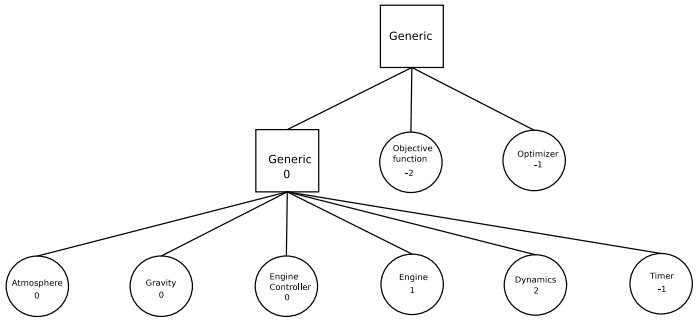

Figure 2.2: Alternate view of 3dof sim topology

```
    Boolean set to true if vector is a row vector
    and false if vector is a column vector

elements
     List of the vector's elements. All the elements
     should be of a type with  __add__(), __sub__()
     and __mul__() defined.

__add__(  self, other )
     Returns a vector that is the sum of self and other.

__sub__(  self, other )
     Returns a vector that is the difference of self and other.

__mul__(  self, other )
     Returns a vector that is the product of self and other.
```

If other is a scalar, multiply it piecewise with self.

__radd__(*self*, *other*)
__rsub__(*self*, *other*)
__rmul__(*self*, *other*)
The right hand versions of __add__, __sub__ and __mul__.

norm(*self[,p]*)
The norm of the vector (2 norm by default). *p* is
the p-norm value. No support of infinity yet.