USER MANUAL

POST

Version 0.1.0

Tibor Völcker

June 1, 2024

Contents

Acronyms **2**

Nomenclature **2**

1 Introduction **5**

2 Overview **5**
2.1 Simulation 5
2.1.1 Coordinate Frames 7
2.1.2 Equation of Motion 9
2.2 Optimization 10
2.3 Further reading 11

3 Usage **11**
3.1 Configuration File 14

4 Discussion **17**
4.1 Differences to the original 17
4.1.1 Missing Features 17
4.1.2 Auto-throttling 18
4.2 The programming language used: Rust 19
4.2.1 The Good 19
4.2.2 The Bad 20
4.2.3 The Verdict 21

5 Development **21**

List of Figures **22**

References **22**

Acronyms

CLI Command Line Interface.

DoF Degree of freedom.

JSON JavaScript Object Notation.

NASA National Aeronautics and Space Administration.

POST Program to Simulate Optimized Trajectories.

Nomenclature

$\dot{\{\}}$	Derivative with respect to time.
$\{\}_b$	Variable in body frame.
$\{\}_I$	Variable in inertial frame.
$\{\}_L$	Variable in launch frame.
$\{\}_R$	Variable in earth-relative frame.
A_{AB}	Acceleration due to aerodynamic forces in body frame.
$A_{E,i}$	Exit area of engine i.
A_{SB}	$= A_{TB} + A_{AB}$. Sensed acceleration in body frame.
A_{TB}	Acceleration due to thrust in body frame.
α	Angle-of-attack.
C_A	Axial aerodynamic force coefficient.
C_D	Aerodynamic drag-force coefficient.
C_L	List drag-force coefficient.
C_N	Normal aerodynamic force coefficient.
C_Y	Aerodynamic side-force coefficient.
g_0	Standard gravity, $= 9.80665 \frac{m}{s^2}$.
G_I	Acceleration due to gravity.
h	Altitude above earth's oblate surface.
$i_{p,i}$	Thrust pitch incidence angle of engine i.
$I_{sp,i}$	Specific impulse of engine i.
$i_{y,i}$	Thrust yaw incidence angle of engine i.
$[IB]$	Transformation from inertial to body frame.
J_i	i-th earth's gravitational harmonics.
m	Mass of the vehicle.
μ	$= J_1$. Earth's gravitational constant.
N_{eng}	Number of engines.
$p(h)$	Atmospheric pressure at altitude h.
$q(h)$	Dynamic pressure at the altitude h.
R_A	Average surface altitude, $= \frac{1}{2}(R_E + R_P)$.
R_E	Equatorial radius.

r_I	Position in inertial frame.
R_P	Polar radius.
S	Vehicle reference area.
T_i	Thrust of engine i.
$T_{vac,i}$	Vaccum thrust of engine i.
v_I	Velocity in inertial frame.
x_X	First unit vector of frame X.
y_X	Second unit vector of frame X.
z_X	Third unit vector of frame X.

1 Introduction

This program is an adaptation of the Program to Simulate Optimized Trajectories (POST). It is a program to optimize a 3 Degrees of Freedom (DoF) trajectory of a vehicle, e.g. a rocket ascending through the atmosphere.

POST was written in 1970 by the Martin Marietta Corporation for NASA. The program is written in Fortran 4 and was mainly developed to simulate the space shuttle, it was later built upon and greatly improved[7]. Its successor, POST2, is still in use today, for example for the Artemis program or for Perseverance[6].

This project tries to use the original user manuals, which are now publicly available (see Section 2.3) to rewrite the program in a more modern programming language: Rust. While I tried to stay close to the original, there are a few differences. Also, only the most important features of the program were incorporated for now, but more features can be added in the future. This is discussed further in Section 4.1.

Special thanks goes to Prof. Dr. Marco Schmidt, which gave me the opportunity to work on this project as a university course.

The manual is structured as follows: In Section 2, an overview of the implementation is given. How to use the project is described in Section 3. The differences to the original and the programming language used is discussed in Section 4. Lastly, notes about the ongoing development are given in Section 5.

2 Overview

This section will give an overview of the program. The program is split into two parts. The simulation and the optimization. The optimization is tweaking the parameters of the simulation and executes it repeatedly to search for the optimal trajectory. This macro logic is shown in Fig. 1.

First, I will show how the simulation works in general, then I will give a few more details on the equations of motion. Next, I will give an idea on how the optimization works. Lastly, I give an overview of the original documentation, where each part is explained in great detail.

2.1 Simulation

The simulation consists of multiple phases, each split by an event. One example is shown in Fig. 2. This actually the example which was used to validate the implementation of this project.

Each phase consists of two sets of variables: the configuration parameters and the simulation variables. The configuration parameters define the vehicle and its environment, while the simulation variables are calculated by the simulation for each time step. To understand the concept better, here are some examples for configuration parameters:

- Planet model

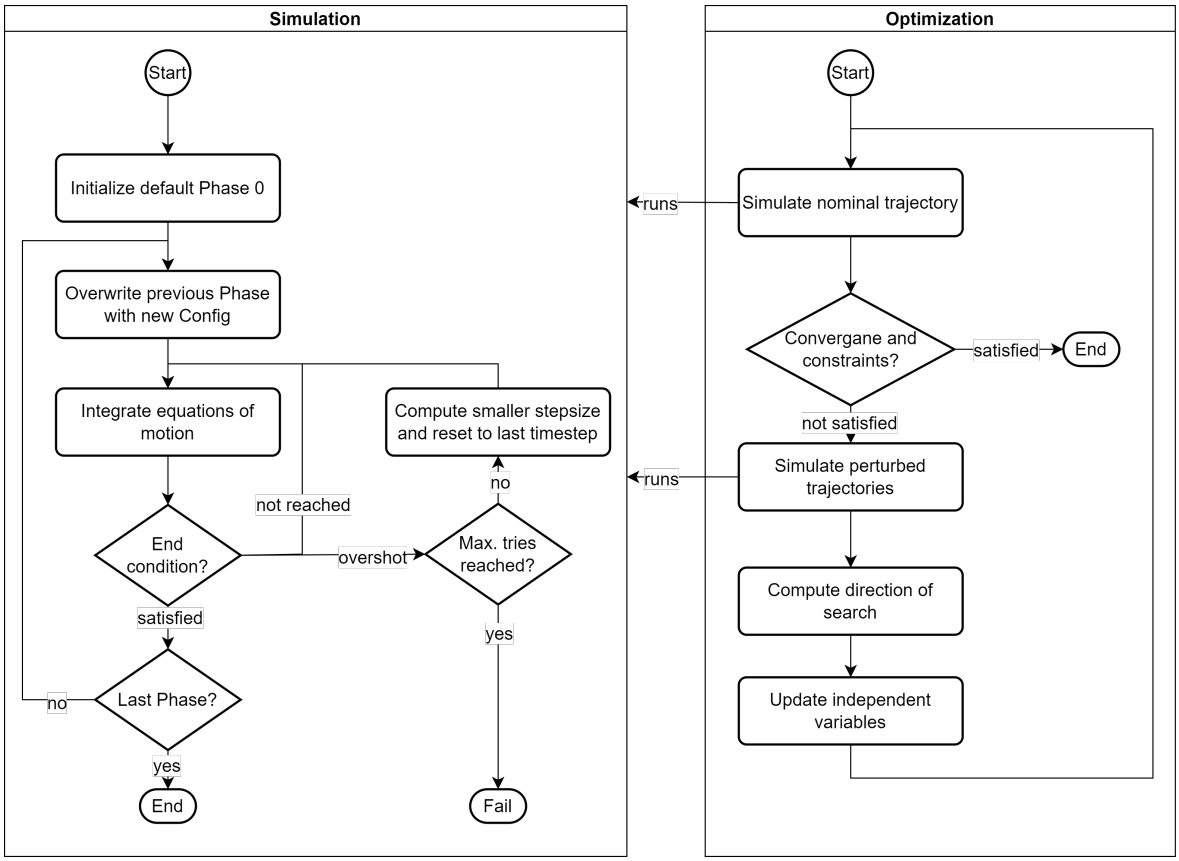

Figure 1: Program macrologic.

- Wind strength / direction
- Launch latitude
- Vehicle structure mass
- Vehicle pitch rate
- Simulation step size
- etc.

And here are some examples for simulation variables:

- Simulation Time
- Position
- Acceleration
- Mass
- Euler Angles
- Aerodynamic Forces
- etc.

These are described in detail in Section 3.

Each event is always some simulation variable reaching a specified value, both of which can be defined by the user in the configuration. At the beginning of each phase, the user can define changes in the configuration.

The simulation executes each phase in the following way:

First, the phase is initialized with the configuration and end state of the previous phase, then it is updated with its own configuration parameters.

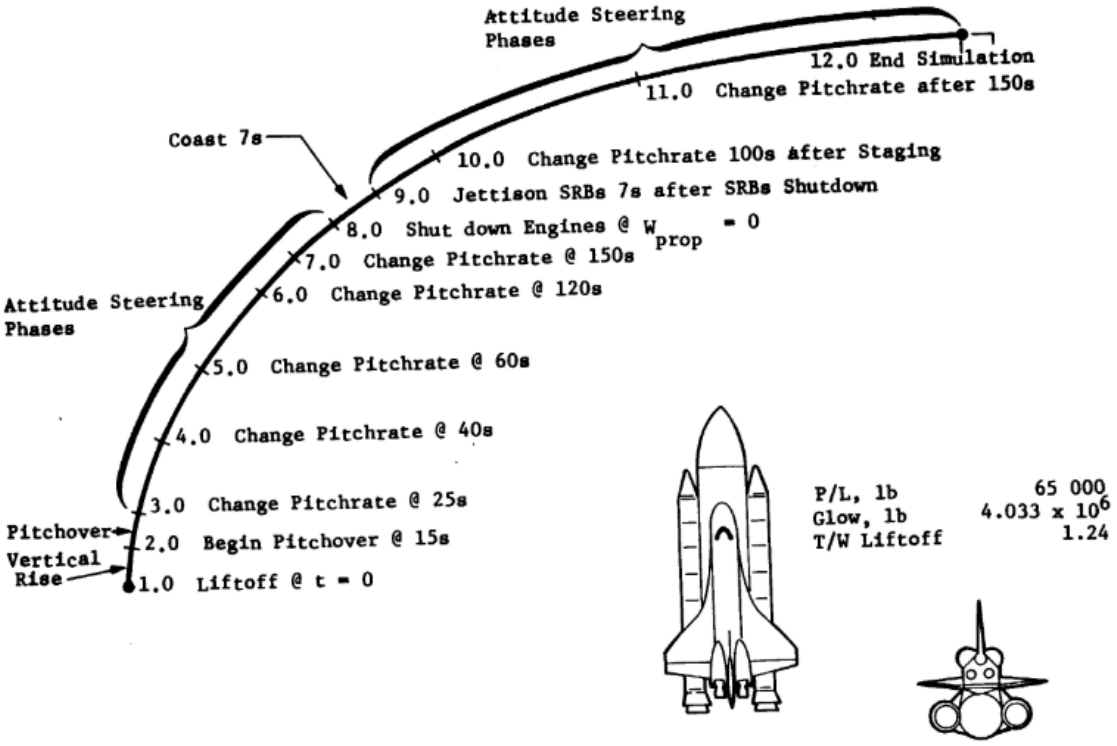

Figure 2: Example simulation split into phases by events from [1, Fig. 28].

Next, an integrator is integrating the equations of motion and the mass flow for a fixed time step. The next chapter will detail the equations of motion. In the equations of motion, the simulation state, consisting of all simulation variables, is slowly build step by step.

After each time step, the simulation checks the termination condition for the current phase. If the specified value is reached (within a margin), the phase is terminated. If the simulation overshot the termination condition, it will compute a smaller time step and redo the simulation step.

2.1.1 Coordinate Frames

There are 4 coordinate frames used. These are the inertial frame, earth-relative frame, launch frame and body frame.

Inertial Frame The inertial frame is located at earth's center and is fixed in inertial frame. z_I points towards the North Pole and the x_I, z_I plane intersects Greenwich at time zero.

Earth-relative Frame The earth-relative frame is the same as the inertial frame at time zero, but rotates together with the earth. This means the difference of two velocity vectors in both frames is earth's velocity at that point.

Launch frame The launch frame is initialized at the launch location and planar to the oblate earth's surface with z_L pointing towards the North Pole, except it is rotated by the `"azimuth"` parameter specified by the user. The frame is shown in Fig. 3.

The launch frame is an inertial frame, meaning it does not rotate with earth. It is used as a fixed reference point for the Euler angles.

Body frame The body frame is located at the vehicles center of mass. x_B points in the direction of flight, rotation around this axis corresponds to roll. Rotations around y_B corresponds to pitch and rotations around z_B corresponds to yaw. The frame is defined arbitrarily, only the user needs to be consistent with its usage, as the user defines the thrust direction and aerodynamic coefficients.

Attention: The orientation using the euler angles are in the order roll-yaw-pitch. Read more about the coordinate systems and transformations in [3, Sec. III].

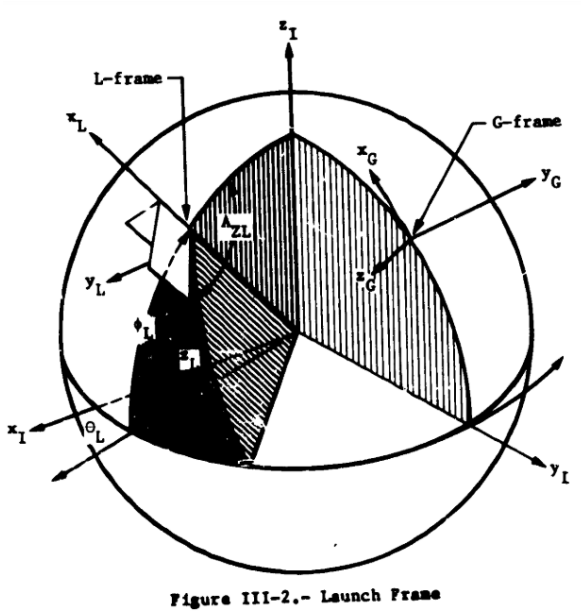

Figure 3: The Inertial Launch Frame (G-Frame not used) from [3, Fig. III-2].

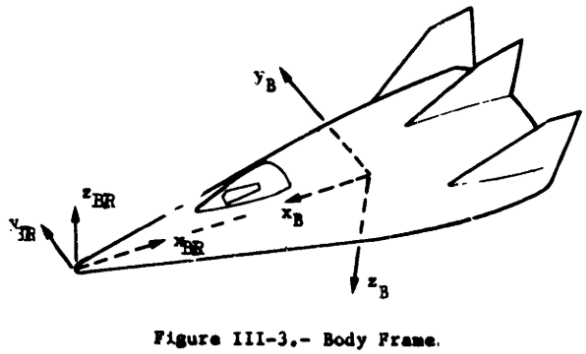

Figure 4: The Body Frame (BR-Frame not used) from [3, Fig. III-3].

The original program also includes some other coordinate frames which are not used by this implementation.

2.1.2 Equation of Motion

The equations of motion are the heart of the simulation. They are simple differential equations describing the motion of the vehicle. As the program only uses 3 DoF, they only describe the translational movement:

$$\begin{aligned} \dot{r}_I &= v_I \\ \dot{v}_I &= [IB]^{-1}(A_{TB} + A_{AB}) + G_I \end{aligned}$$

The calculation of each part is outlined in the following paragraphs to give a rough understanding of the simulation, but many details are omitted. More details can be found in [3, Sec. VI-4].

Orientation The transformation from the inertial to the body frame $[IB]$ depends on the inertial Euler angles and the launch coordinate frame (L-Frame). The Euler angles are defined with respect to the launch frame.

The Euler angles are each computed by three cubic polynomials

$$\alpha = c_0 + c_1 y + c_0 y^2 + c_1 y^3.$$

The coefficients c_i can be defined by the user. The variable y is some simulation variable, which can be also picked by the user. Each angle α (roll, yaw & pitch) has a separate polynomial and can be configured separately.

Acceleration due to Thrust The thrust of each engine T_i is calculated with

$$T_i = T_{vac,i} - A_{E,i} \cdot p(h),$$

where $T_{vac,i}$ and $A_{E,i}$ are specified by the user, and $p(h)$ is calculated with the atmospheric model. The acceleration due to thrust in the body frame A_{TB} is then calculated with

$$A_{TB} = \frac{1}{m} \sum_i^{N_{eng}} T_i \begin{bmatrix} \cos i_p \cos i_p \\ \sin i_y \\ \cos i_y \sin i_p \end{bmatrix}_i,$$

$i_{p,i}$ and $i_{y,i}$ are specified by the user for each engine, which define the angles between the thrust vector and the body frame.

Additionally, the mass flow is calculated with

$$\dot{m} = \sum_i^{N_{eng}} T_{vac,i} \cdot I_{sp,i} \cdot g_0.$$

Acceleration due to Aerodynamic Forces The acceleration due to aerodynamic forces in the body frame A_{TB} is calculated with

$$A_{AB} = \frac{1}{m} q(h) S \begin{bmatrix} -C_A \\ C_Y \\ -C_N \end{bmatrix},$$

where $q(h)$ is calculated with the atmospheric model. S is specified by the user.

C_A and C_N are calculated with

$$A_{AB} = \frac{1}{m} q(h) S \begin{bmatrix} C_A \\ C_N \end{bmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} C_D \\ C_L \end{bmatrix},$$

where C_D, C_L and C_Y are specified by the user using tables.

The tables can be 1D, 2D or 3D, which are then interpolated using a simulation variable, which is also specified by the user.

Acceleration due to Gravity The acceleration due to gravity is calculated with the gravity model. There are three different gravity models available:

- Spherical Model
- 1960 Fisher Model
- Smithsonian Model

Each model calculates using simple gravitational harmonics J_i, where $J_1 = \mu$, the earth's gravitational constant. The spherical model uses a spherical earth model and therefore only the gravitational constant mu. The 1969 Fisher model uses an oblate spheroid and harmonics up to J_2, while the Smithsonian Model includes harmonics up to J_4.

Atmospheric Model The atmospheric model used is the 1962 U.S. standard atmosphere model with a few changes for above 90 km.

2.2 Optimization

The optimization algorithm is detailed in the formulation manual. A rough sketch of the algorithm is as follows:

1. First, the nominal trajectories are simulated.
2. Now, convergence is tested. If the problem converged, we found the optimal solution.
3. If not, we simulate a slightly perturbed trajectory for each independent variable.
4. With these small perturbations, the dependency of the cost function (our optimization goal) regarding the independent variables can be calculated. This is called the cost gradient.
5. The same can be done for the constraints. This is called the sensitivity matrix.

6. Next we use the cost gradient to calculate the optimization step size and direction, which is how the independent variables should be changed to optimize our problem.
7. We use the calculated step size and direction to update our independent variables.
8. Next we calculate the constraint step size and direction, which is how the independent variables should be changed to honor the constraints.
9. We use the calculated step size and direction to update our independent variables again.
10. Then, we repeat the process.

2.3 Further reading

The program is based on the manual published in April 1975. It includes three volumes:

1. Volume 1: Formulation manual[3]
2. Volume 2: Utilization manual[4]
3. Volume 3: Programmer's manual[5]

Volume 1 is the most important one explaining the working principle and formulas used. Volume 2 explains the usage of the original program, which includes a few bits of extra information and an example which was important for this project. Volume 3 explains the program's structure and was only used a little to get an overview.

For further explanations of this project, look into these manuals. For convenience, I included the manuals here.

There also exists a Program summary document[1] published in 1977.

Additionally, there exists a bit of documentation about the various additions, like

- 6 DoF capabilities[2]
- Parallelization[9]
- and interplanetary missions (Volume 1[11], Volume 2[12], Volume 3[13] and Volume 4[14]).

These were not used for this project.

3 Usage

The tool's usage is very basic. It consists of one executable, which needs to be called with the file path to a configuration file, which is detailed below.

The easiest way to run the program is to download the pre-built executables. These can be found in the latest release. Choose the respective executable for your operating system (Windows, macOS, or Linux as well as x86_64 or AMD64), and unpack the directory.

Build it yourself If there is no pre-built executable for your system, you need to build it yourself. For this, download the project files and make sure the rust compiler and the cargo tool is installed[1].

To build the tool, run

```
$ cargo build --release
```

The executable should now be located at `target/release/post.exe`.

Running the simulation You can run the simulation with:

```
$ post.exe --config <config filepath>
```

This will write the time, position, velocity, altitude, and propellant mass for each time step to the standard output, as seen below:

```
Starting Phase 1
Time: 1
Position:
  ⎡   915482 ⎤
  ⎢ -5529976 ⎥
  ⎣  3043398 ⎦


Velocity:
  ⎡ 404 ⎤
  ⎢  65 ⎥
  ⎣   1 ⎦


Altitude: 1
Prop mass: 1014475

Time: 2
Position:
  ...
```

Alternatively, the tool can be build and executed in one command with the `cargo run` command if the rust compiler and cargo tool are installed.

Plotting tool The project also includes a small plotting tool to plot the simulated trajectory. For this, you will need Python installed. Python version 3.12 is recommended.

[1] https://www.rust-lang.org/tools/install

Note: If you built the project yourself, the plotting tool is located at `utils/plot`.

You can install the requirements with

```
$ pip install -r plotting-tool\requirements.txt
```

Then, the simulated trajectory can be piped to the plotting tool with:

```
$ post.exe --config <config filepath> | python
   ↪ plotting-tool\plot.py
```

You should see a window open to show the simulated trajectory like in Fig. 5.

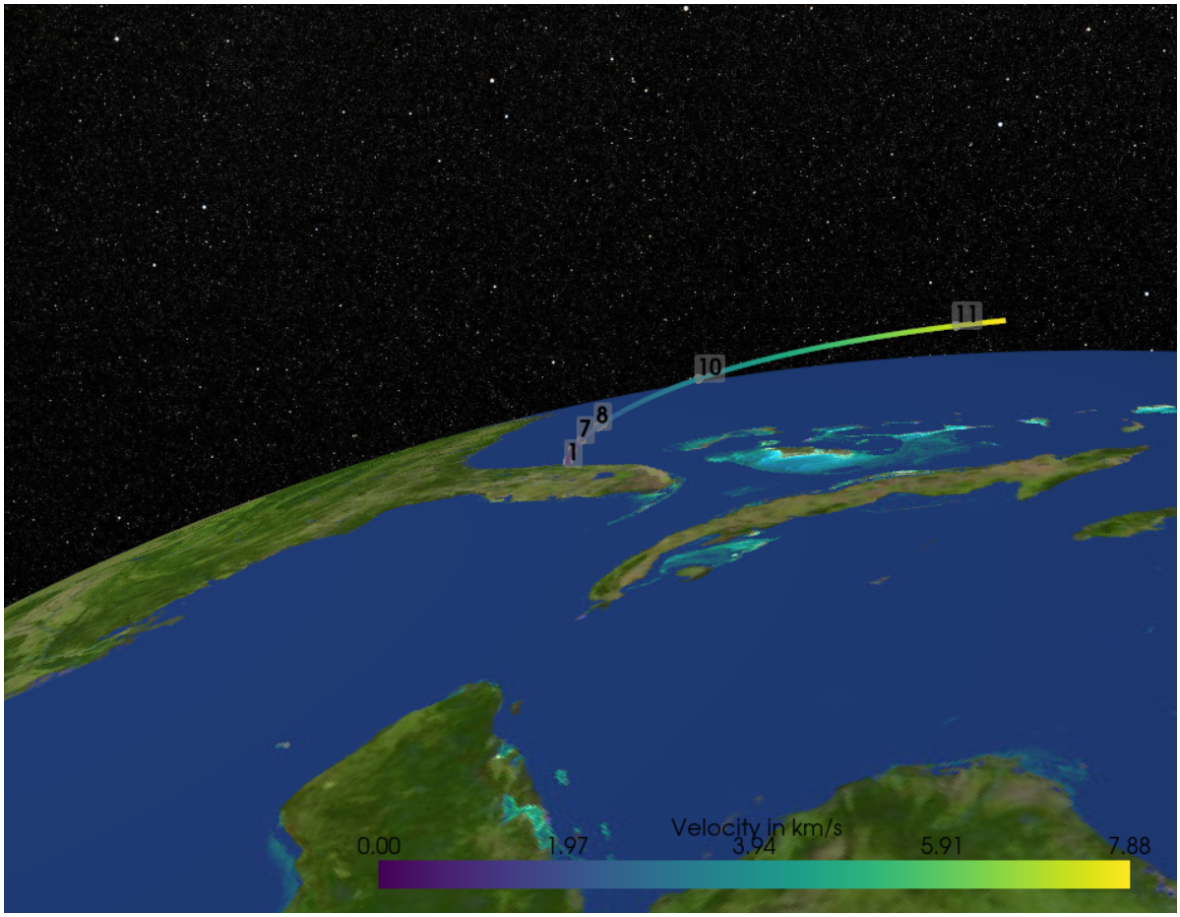

Figure 5: The example trajectory shown with the plotting tool. The color indicates the velocity of the vehicle while the labels indicate the start of a new phase.

Example The original documentation also included an example trajectory for the ascent of a space shuttle type vehicle. This example is used to validate this project.

It is also a good baseline for custom configurations.

Note: If you built the project yourself, the example is located at `utils/example.json`.

To execute the example, run:

```
$ post.exe --config example.json
```

3.1 Configuration File

The JSON configuration file consists of an array of configurations, one for each phase. Each phase is inheriting the configuration of the previous phase. Then the specified parameters are overwritten.

As the first phase does not have a previous phase, there is a default phase to inherit from:

```
[
  {
    "planet_model": "spherical",
    "atmosphere": {
      "enabled": false,
      "wind": [ 0, 0, 0 ] // in m/s
    },
    "init": {
      "latitude": 0, // in °
      "longitude": 0, // in °
      "azimuth": 0, // in °
      "altitude": 0 // in m
    },
    "vehicle": {
      "structure_mass": 0, // in kg
      "propellant_mass": 0, // in kg
      "reference_area": 0, // in m^2
      "drag_coeff": {"x": ["time", []], "data": []},
      "lift_coeff":  {"x": ["time", []], "data": []},
      "side_force_coeff":  {"x": ["time", []], "data": []},
      "engines": [
          {
            "incidence": [ 0, 0 ], // in rad
            "thrust_vac": 0, // in N
            "isp_vac": 0, // in s
            "exit_area": 0 // in m^2
          }
        ]
      },
      "max_acceleration": -1, // in m/s^2, -1 for disabling
      "steering": {
      "roll": [ "time", [ 0, 0, 0 ] ], // in °
      "yaw": [ "time", [ 0, 0, 0 ] ], // in °
      "pitch": [ "time", [ 0, 0, 0 ] ]  // in °
    },
    "stepsize": 0, // in s
    "end_criterion": [ "time", 0 ]
  },
  { ... },
  ...
]
```

Overwriting the propellant mass will also reset the consumed propellant.

Setting a parameter to `null` is the same as not specifying it, so it will not be overwritten. Remove a table by setting it to `x": ["time", []], "data": []"x": ["time", []], "data": []`. To disable thrust, remove the engines by setting them to `[]`.

Planet model The planet model has 4 valid parameters: `"spherical"`,`"fisher_1960"`, `"smithsonian"`, or a custom model. A custom planet model must be specified with:

```
{
  "custom": {
    "equatorial_radius": 6.3781658568e6, // in m
    "polar_radius": 6.356783832e6, // in m
    // [mu, J_2, J_3, J_4]
    "gravitational_parameters": [
        1.3077457350581376e15, // in m^3/s^2
        1.082639e-3,
        -2.565e-6,
        -1.608e-6,
    ],
    "rotation_rate": 7.29211e-5, // in rad/s
  }
}
```

Here, the Smithsonian model implementation is shown.

Steering The steering configuration are the coefficients c_i for a cubic polynomial of the form

$$\alpha = c_0 + c_1 y + c_0 y^2 + c_1 y^3.$$

The first coefficient c_0 is initialized as zero for the first phase. For later phases, it is initialized as the last Euler angle of the previous phase. This is done to avoid jumps in the orientation.

The variable y is any simulation variable. Following variables are defined:

Simulation Variables	
`"time"`	Simulation time
`"time_since_event"`	Time since the last event
`"position1"`	Inertial position
`"position2"`	
`"position3"`	
`"position_norm"`	Distance from earth center
`"position_planet1"`	Earth-relative position
`"position_planet2"`	
`"position_planet3"`	
`"altitude"`	Height above surface
`"altitude_geopotential"`	Used for the atmospheric model[2]
`"velocity1"`	Inertial velocity

[2] Calculated with $\frac{R_A h}{R_A + h}$.

`"velocity2"`	
`"velocity3"`	
`"velocity_norm"`	Total inertial velocity
`"velocity_planet1"`	Earth-relative velocity
`"velocity_planet2"`	
`"velocity_planet3"`	
`"velocity_planet_norm"`	Total earth-relative velocity
`"velocity_atmosphere1"`	Atmosphere-relative velocity
`"velocity_atmosphere2"`	
`"velocity_atmosphere3"`	
`"velocity_atmosphere_norm"`	Total atmosphere-relative velocity
`"gravity_acceleration1"`	Acceleration due to gravity
`"gravity_acceleration2"`	
`"gravity_acceleration3"`	
`"gravity_acceleration_norm"`	Total acceleration due to gravity
`"mass"`	Total vehicle mass
`"propellant_mass"`	Propellant mass
`"temperature"`	Atmosphere temperature
`"pressure"`	Atmosphere pressure
`"density"`	Atmosphere density
`"mach_number"`	Vehicle mach number
`"dynamic_pressure"`	Dynamic pressure
`"alpha"`	Angle of attack
`"euler_angles_roll"`	Roll angle with respect to launch frame
`"euler_angles_yaw"`	Yaw angle with respect to launch frame
`"euler_angles_pitch"`	Pitch angle with respect to launch frame

Important: The values `"alpha"` and `"euler_angles_*"` are not available for steering, as they are not computed yet. If chosen, they will return zero (see Section 5).

Aerodynamic coefficients Aerodynamic coefficients are input as tables. There are 1D, 2D and 3D tables, which are structured as follows:

1D:

```
{
  "x": ["<state var>", [ ... ]],
  "data": [ ... ]
}
```

2D:

```
{
  "x": ["<state var>", [ ... ]],
  "y": ["<state var>", [ ... ]],
  "data": [ [ ... ] ]
}
```

3D:

```
{
  "x": ["<state var>", [ ... ]],
  "y": ["<state var>", [ ... ]],
  "z": ["<state var>", [ ... ]],
  "data": [ [ [...] ] ]
}
```

Note that the `"data"` parameter changes: A 1D table uses an array, a 2D table an array of arrays and a 3D table an array of arrays of arrays.

For the `"x"`, `"y"` and `"z"` variables, any simulation variable (defined above) can be specified. This will then be used to interpolate between the data points.

4 Discussion

In this chapter, I will discuss the differences of this project to the original. Also, I will outline my thoughts about the programming language of my choice for this project, Rust.

4.1 Differences to the original

While I tried to closely follow the original, some things differ. Mostly, not all features were implemented. This is further detailed in this chapter.

4.1.1 Missing Features

The original program includes a lot of features and multiple options for many of them. As time is limited, this project only implemented the most important ones (the ones needed to simulate the example space shuttle trajectory). Some of these missing features might be implemented in the future, but most of them are deemed unnecessary for now.

Initialization While this project only includes initialization using geodetic latitude, longitude and altitude, the original program also included initialization with inertial position and velocity, with orbit parameters and a few others. The most important of which are planned to be implemented in the future.

Steering In the original program, the vehicle could be steered using angular rates, aerodynamic angles, relative Euler angles and inertial aerodynamic angles. They could be calculated with (cubic) polynomials, tables, or even by implementing a custom feedback controller. While most of these are not considered to be implemented, a 6 DoF version would be interesting which would come with big changes to the steering model.

Additional calculations The original program would calculate a bunch of additional variables for each time step, which partly could be used as inputs for steering or tables. These included a heating model, orbit parameters, range calculations, ground station visibility, sun and shadow calculations and others. These are not essential, but the most important of them are planned to be implemented sometime in the future.

Atmosphere models In the original program, one could also use a custom atmosphere model specified with tables and the 1963 Patrick AFB model. These will most likely never be implemented, but rather a more modern model.

Jet engines The original program could also simulate jet engines. As I am mostly interested in space travel, they will most likely not be implemented in the future.

Events The original program included some more options to specify events, like optional or repeating events. As these are only needed for more complex missions, they are in the backlog for now.

Integrators The original program included more options for integrators like the Krogh Integrator, the Laplace method or Encke's method. These will be implemented only if necessary.

4.1.2 Auto-throttling

The original documentation is very detailed regarding the simulation and can be followed without big difficulty. There is, however, one part which I did not find any explanation about, which is the auto-throttling.

The user can specify a maximum sensed acceleration, which should not be exceeded. To ensure this, the thrust is throttled automatically. The algorithm to calculate the throttle had to be designed by myself, which works as follows.

Problem statement A_{TB} and A_{AB}, together with the resulting A_{SB} form a triangle in 3D space, as seen in Fig. 6. A_{AB} can be calculated with v_I, while the the magnitude of A_{SB} should not exceed a maximum value. Also, the direction of A_{TB} is known from the thrust incidence angles $i_{y,i}$ and $i_{p,i}$.

Solution Using the direction of A_{TB} and A_{AB}, the angle between the two vectors α can be calculated. Now the magnitude of A_{AB} and A_{SB} is given together with an angle. This means we can calculate all values in the triangle with the side-side-angle method.

Using the law of sines with α, A_{AB} and A_{SB}, a second angle β can be calculated, which is opposite of A_{AB}. The third angle γ trivially follows, as all angles add up to 180° or π.

Finally, the law of sines can be used again to calculate A_{TB}, using A_{AB}, β and γ.

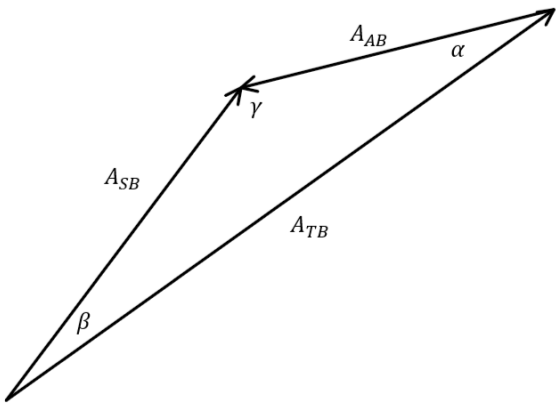

Figure 6: The auto-throttling problem.

Lastly, one has to catch all the edge cases, and make sure the resulting A_{TB} is not negative or bigger than the maximum possible A_{TB} using full thrust.

4.2 The programming language used: Rust

Rust was chosen for this project for a few simple reasons.

First, Rust is fast[10, 8]. Its speed is easily comparable with C and C++. As this project is about an optimization of a complex simulation, speed is very likely of importance.

Also, Rust is safe. Its unique borrow checker and strong type system allows you to write reliable, safe and less buggy code more easily. This takes away the nightmare of having to debug memory related bugs, like with C or C++. Basically, it is much harder to shoot yourself in the foot.

But most importantly, I wanted to try out and learn Rust. Rust is the consecutive most loved programming language for 8 years in a row, according to [15], which piqued my interest. Also, as I mostly work with Python, I wanted to add a fast, low-level programming language to my arsenal.

4.2.1 The Good

This chapter is inherently opinion based. This chapter is only about how I experienced working with and learning the language and are exaggerated a bit to underline my points. Some issues can very much boil down to me not having enough experience with the language, or with low-level languages in general.

First things first, I think Rust is doing a lot of things right. There are a bunch of things great about the language. Like, as already mentioned, the safety does free you from some annoying bugs. But sometimes overlooked, I believe the environment of a programming language is sometimes more important than the language itself. And here, Rust really shines.

There exists a central tool for downloading, installing and updating the Rust language, compiler, package manager and tools, called “rustup”.

The best part is the package manager, called "Cargo". It is used to run all the tools that already come with Rust. It can be used to create a new project, build and run your code, test or benchmark it, download packages and add it to your dependencies. It is used for linting and formatting, to create automatic documentation and even publish it to Rust's package index.

Languages like Python also have all of these features, but in Rust, most of them are already coming with your installation, ready to use. Or more importantly, there is one "official" tool for all of these tasks. I mostly do not care how to format my code, I only want it to be easy and uniformly across all projects. The exact formatting rules don't matter, as long as they are officially declared.

There is also one way to declare dependencies, give additional information about the project and publish your code and documentation. Speaking of documentation, automatic documentation from code comments is already built-in, even with examples which are automatically tested! And the documentation can then be published on an official site which is linked with the package index.

4.2.2 The Bad

In my experience with this project, Rust can be very clunky and writing it is slow. While this certainly gets better with more experience, I believe it will always make your life hard in certain scenarios. Also, I believe the learning experience should be considered when evaluating a programming language, which can feel like an uphill battle in Rust. While there were many instances where the language was standing in the way of how I wanted to solve a problem, a few examples are detailed here.

Rust's speed can be quickly compromised (and probably is in many parts of this project) by an exhausted developer who does not want to fight the borrow checker anymore and simple clones a rather big structure.

Also, dealing with slightly complex structures is a nightmare. One example for this are the tables used for the aerodynamic coefficients. At first, these were implemented recursively. A 3D table consisted of a number of 2D tables, and so on. Working out how build the structure was already complex, but manageable. The problems began, once these tables needed to be cloneable: The tables were owned by the vehicle struct. Because the type must be known, the tables were a generic type, restricted by a shared trait. Because it was infeasible to copy the type restriction of the generic (meaning the specification of the trait) across the entire project, the trait was "dynamically dispatched". This then in turn restricts the trait to be 'object safe', which is what being cloneable is not[3].

[3] This can be circumvented with the `dyn_clone` crate. The implementation fell apart, though, when the tables needed to be deserializable. Again, deserialization is not object safe, but can be again be circumvented with `erased_serde`. But I spend about a week to figure out how to deserialize a recursive structure, without knowing which type it is beforehand. So I figured it is not worth it and abandoned the (in my eyes more elegant) implementation of the tables.

4.2.3 The Verdict

I believe Rust is a great language, if applied to the right problems.

Before considering Rust, it needs to be clear that its speed is really needed. Also, the problem should not require too many complex, nested or recursive data structures. I think Rust really shines for low-level high-intensity tasks and complemented by high-level languages which handle the encompassing logic.

5 Development

The project is still in development. Currently, the simulation part of the program is functional, but the optimization part is still missing.

Other than that, there still exists some known issues and some changes are planned. While these are tracked in the issues, the most important ones are outlined below.

To read more about the code of the project, visit the code documentation.

Existing Issues

- Program panics when reaching max. acceleration
- Unspecified behavior for negative altitudes
- Unspecified behavior or crashing for invalid configuration (e.g. zero mass)
- Unlimited simulation duration if end condition is never reached (or was reached before event started)
- Thrust is generated even when propellant was consumed
- Bad performance due to writing to stdout, unnecessary cloning or passing by value
- The state variable inputs allow any state variable. But when steering or aerodynamic forces are calculated, not all state variables are calculated yet.

Planned Changes and Features

- Output timesteps as a table
- Let output be configurable
- Improve logging using the `log` crate
- Improve code documentation with comments and examples
- Include plotting script into the Command Line Interface (CLI)
- Add figures for e.g. orientation in the plotting script
- Add safety against unit (rad vs °) or frame differences (inertial vs relative)

List of Figures

1 Program macrologic. 6
2 Example simulation split into phases by events from [1, Fig. 28]. 7
3 The Inertial Launch Frame (G-Frame not used) from [3, Fig. III-2]. 8
4 The Body Frame (BR-Frame not used) from [3, Fig. III-3]. 8
5 The example trajectory shown with the plotting tool. The color indicates the velocity of the vehicle while the labels indicate the start of a new phase. 13
6 The auto-throttling problem. 19

References

[1] G. L. Brauer, D. E. Cornick, and R. Stevenson. *Capabilities and applications of the Program to Optimize Simulated Trajectories (POST). Program summary document*. NASA. Feb. 1, 1977. URL: https://ntrs.nasa.gov/api/citations/19770012832/downloads/19770012832.pdf (visited on 05/18/2024).

[2] G. L. Brauer, A. R. Habeger, and R. Stevenson. *Six-degree-of-freedom program to optimize simulated trajectories (6D POST). Formulation manual*. NASA. Nov. 1, 1974. URL: https://ntrs.nasa.gov/api/citations/19760006045/downloads/19760006045.pdf (visited on 05/18/2024).

[3] G. L. Brauer et al. *Program to Optimize Simulated Trajectories (POST). Formulation manual*. NASA. Apr. 1, 1975. URL: https://ntrs.nasa.gov/api/citations/19750024073/downloads/19750024073.pdf (visited on 05/18/2024).

[4] G. L. Brauer et al. *Program to Optimize Simulated Trajectories (POST). Utilization manual*. NASA. Apr. 1, 1975. URL: https://ntrs.nasa.gov/api/citations/19750024074/downloads/19750024074.pdf (visited on 05/18/2024).

[5] G. L. Brauer et al. *Program to Optimize Simulated Trajectories (POST). Programmer's manual*. NASA. Apr. 1, 1975. URL: https://ntrs.nasa.gov/api/citations/19750024075/downloads/19750024075.pdf (visited on 05/18/2024).

[6] William T. Colson. *POST2 Applications*. NASA. Oct. 31, 2023. URL: https://www.nasa.gov/post2/applications/ (visited on 05/18/2024).

[7] William T. Colson. *POST2 History*. NASA. Oct. 30, 2023. URL: https://www.nasa.gov/post2/history/ (visited on 05/18/2024).

[8] Isaac Gouy. *Measure "Which programming language is fastest?"*. Apr. 5, 2024. URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/ (visited on 05/18/2024).

[9] Data P. Hammond and John J. Korte. *Parallelization of Program to Optimize Simulated Trajectories (POST3D)*. NASA. Nov. 1, 2001. URL: https://ntrs.nasa.gov/api/citations/20020004355/downloads/20020004355.pdf (visited on 05/18/2024).

[10] hanabi1224. *C VS Rust benchmarks*. Feb. 1, 2024. URL: https://programming-language-benchmarks.vercel.app/c-vs-rust (visited on 05/18/2024).

[11] P. E. Hong et al. *Interplanetary Program to Optimize Simulated Trajectories (IPOST). User's guide*. NASA. Oct. 1, 1992. URL: https://ntrs.nasa.gov/api/citations/19930005605/downloads/19930005605.pdf (visited on 05/18/2024).

[12] P. E. Hong et al. *Interplanetary Program to Optimize Simulated Trajectories (IPOST). Analytic manual*. NASA. Oct. 1, 1992. URL: https://ntrs.nasa.gov/api/citations/19930010932/downloads/19930010932.pdf (visited on 05/18/2024).

[13] P. E. Hong et al. *Interplanetary Program to Optimize Simulated Trajectories (IPOST). Programmer's manual*. NASA. Oct. 1, 1992. URL: https://ntrs.nasa.gov/api/citations/19930005541/downloads/19930005541.pdf (visited on 05/18/2024).

[14] P. E. Hong et al. *Interplanetary Program to Optimize Simulated Trajectories (IPOST). Sample cases*. NASA. Oct. 1, 1992. URL: https://ntrs.nasa.gov/api/citations/19930005579/downloads/19930005579.pdf (visited on 05/18/2024).

[15] Stack Overflow. *Stack Overflow Developer Survey 2023*. May 2023. URL: https://survey.stackoverflow.co/2023/#technology-admired-and-desired (visited on 05/02/2021).