loganwrightx's GitHub Portfolio

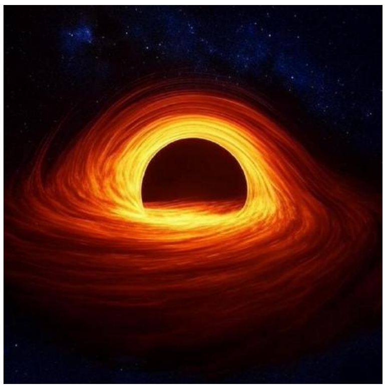
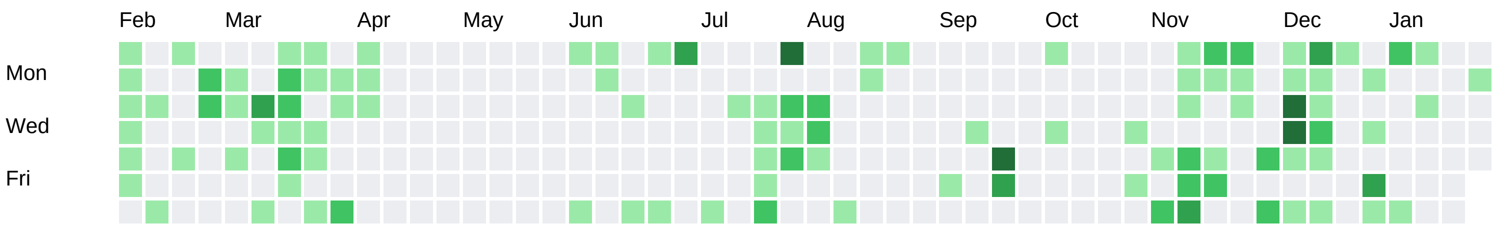


Table of Contents

Contribution Calendar



xrocket_protocol

A header-only protocol project to standardize communication patterns between XRocket and Avionics platforms for HITL testing.

xrocket_protocol

README:

XRocket Protocol

A header-only C++ library for accessing the XRocket Protocol to maintain message types at scale. Use this protocol to interact with the Avionics Server subsystem.

RocketSimulationApp

No description

RocketSimulationApp

README:

Table of Contents

Rocket Simulation Environment

This project is intended to support advanced small-scale spacecraft by generating physically accurate simulations for control system analysis. Will add more details about functionality as the project matures.

Background

The structure of this tool are based on several key ideas, so first, familiarize yourself with them:

* `Element`
datatypes for producing massive rigid body parts

* `ElementDict`
datatypes for constructing argument structures for creating those objects

* `ConfigDict`
which stores the class name of the element and its arguments

* `Builder`
which takes the data dictionary containing the above information to put into a
`Design`

* `Design`
's are used to actually hold the rigid body elements and allow physical operations on them i.e. moments and forces act upon the inertia tensor and mass, respectively, to produce different accelerations on the body

* A
`Design`
is also an efficient controller for modifying objects' positions and orientations in a pre-setting to simulation with the
`.manipulate_element(id: int, displacement: NDArray, attitude: Quaternion)`
method

* The
`.manipulate_element(id: int, displacement: NDArray, attitude: Quaternion)`
method starts with every
`Element`
aligned with their centers of mass at the origin, and after manipulation of the
`Element`
's is complete, a new center of mass vector is computed as well as the inertia tensor for the sum about the center of mass

* Once a

`Design` is complete and all parts have been moved to their respective presets, we call the `.simplify_static_elements()` method, which is also a `.lock()` method and reduces information in the `Design()` by summing the physical effects of the `is_static = True` elements

* When reduction occurs, the `is_static = True` elements are forgotten and can no longer be modified, this is a design choice for efficiency reasons

* If any of the `Element` objects you define have set `is_static = False`, that means that the mass, relative orientation, and position to the `Design` may be modified at any time

NOTE: Calling the `.simplify_static_elements()` method on a `Design` will restrict the user from modifying any parameters about all static `Element`'s, so wait to call this before you want to begin the simulation phase

Assets of the API

Here, I'll keep a list of all of the main `class` elements which are most commonly accessed, their uses, and how to prepare data for each instantiation process. Many of the objects used for this project are singleton instances to minimize memory consumption and to avoid redundant object creation. These will also be specified when listed. As promised:

* `Element` (and its many subclasses) - Represents a massive part of the rocket or general aerospace rigid body. It is an _element_ of a `Design` block

* `Design` - Represents a cluster of `Element`'s and is used to manage the rotations and translations of conslidated reprentations of the inertia tensor and keep track of changing mass for each `DYNAMIC` element. It can be produced automatically by using a `Builder` object

* `Builder`

* `MotorManager`

* `ThrustVectorController`

* `Quaternion`

* `Vector`

Usage

After getting familiar with the ideas above, we can start learning several of the features to access the power of the simulator. This will just be a standard presentation of how a single loop will produce estimations for the next state:

* Before loops begin, call `.simplify_static_elements()` on the `Design`

* Loop begins and we set/call `mass, cg, inertia_tensor = design.get_temporary_properties()` to collect information about the current physical state of the vehicle. These are gathered automatically in the inertial world frame, not the body frame so no rotations are required

* Compute the inverse of the inertia tensor with `inertia_tensor_inv = np.linalg.inv(inertia_tensor)`

* Get the contributed force and torque from the rocket motor using the `ThrustVectorController` object by calling `.getThrustVector()`

. These vectors are in the
BODY FRAME
 and must be manually transformed to the world frame using the process shown in the demonstration below

*	Use the sum of the forces and moments acting on the body in the world (inertial) frame to approximate the next state - convert to linear and angular accelerations with the following repective formulae:
`a = force / mass`
,
`alpha = np.matmul(inertia_tensor_inv, torque.v - np.cross(a=omega.v, b=np.matmul(inertia_tensor, omega.v)))`

*	Convert
`angular_acceleration (alpha)`
 and
`linear_acceleration (a)`
 to
`Vector`
 objects with
`alpha = Vector(elements=(alpha[0], alpha[1], alpha[2]))`
 and
`a = Vector(elements=(a[0], a[1], a[2]))`
, respectively for proper typing in the
`KinematicData`
 named dictionary

*	Use
`q, omega = solver(omega=omega, alpha=alpha, q=q, dt=dt, display=True)`
 to get the new quaternion representing attitude and omega representing the angular velocity vector

*	The last thing we need to do is create a
`KinematicData`
 named dictionary with the updates for the
`Design`
, so we finalize the loop by calling:

```
design += KinematicData(
  R=v * dt,
  V=a * dt,
  Q=q,
  OMEGA=omega
)

design.step(dt=dt)
```

NOTE

Make it clear that when we add this
`KinematicData`

`TypedDict` to the design, we are adding the physical values in terms of the world reference frame, and **NOT** in the body (or equivalently the `Design`) frame of reference. This is done to reduce matrix multiplications and increase code efficiency.

Example Code

Below is a demo of working code for this project to see it in use.

```
from typing import Any
import numpy as np

from Design import *
from Builder import *
from MotorManager import *
from Element import *
from ElementTypes import *
from SerialManager import *
from ThrustVectorController import *
from SimulationLoop import simulationLoop
from PhysicsAPI import *

if __name__ == "__main__":
  api = PhysicsAPI()
  api.postMotor({"motor": "E12"})
  api.postMakeTVC()
  api.postAddElement({
    "flight_computer": ConfigDict(
      Type=Cylinder,
      Args=CylinderDictS(radius=0.072 / 2, height=0.12, mass=0.18, is_static=True)
    )
  })
  api.postAddElement({
    "nose_cone": ConfigDict(
      Type=Cone,
      Args=ConeDictS(radius=0.074 / 2, height=0.2, mass=0.12, is_static=True)
    )
  })
  api.postAddElement({
    "body_tube": ConfigDict(
      Type=Tube,
          Args=TubeDictS(inner_radius=0.072 / 2, outer_radius=0.074 / 2, height=0.8, mass=0.3,
is_static=True)
    )
  })

  api.postBuildDesign()

  api.postMotorAdjustment({"translation": np.array([0, 0, -0.4])})
  api.postElementAdjustment({"id": "flight_computer", "translation": np.array([0, 0, 0.15])})
  api.postElementAdjustment({"id": "nose_cone", "translation": np.array([0, 0, 0.4])})
  api.postElementAdjustment({"id": "body_tube", "translation": np.array([0.0, 0.0, -0.05])})

  api.postLockStaticElements()
  api.postSetTVC({"x": 0.0, "y": 0.0})

  res = api.postConnectSerial({"port": None, "baud_rate": 115200})
  if res["res"]:
    api.postStartListening()

  api.getSimulationResults()

  print("Done!")
```

Common Problems and Solutions

N/A

Report a Problem

To report a problem, simply start an issue on this repository and be specific. If you have the skillset to make fixes, fork the repository to your local editor, commit changes, and submit a pull request for us to review. If there are concerns about copyright issues or if you'd like to use this project for entrprising uses, contact me directly through GitHub messaging.

export-github-as-portfolio

A python tool to export your GitHub profile as either a PDF, HTML, or Markdown portfolio. The portfolio will only display public repositories and user has control over what parts are displayed.

export-github-as-portfolio

README:

export-github-as-portfolio

A python tool to export your GitHub profile as either a PDF, HTML, or Markdown portfolio. The portfolio will only display public repositories and user has control over what parts are displayed.

GroundControlApp

An app to communicate with the ground station for remote launch pad control.

GroundControlApp

README:

GroundControlApp

An app to communicate with the ground station for remote launch pad control.

PH336RocketDynamics

A RK4 solver for a rocket subject to aerodynamic loads and nonlinear forces.

PH336RocketDynamics

README:

Project Title: Dynamic Modeling of a Propulsive Vehicle

Logan Wright, Abram Bell, Wes Crapo, Matthew Boling

Experiment Description

This project is an RK4 solver for a rocket subject to aerodynamic loads and nonlinear forces. The models simulated in this environment are to support real experiments over the course of several weeks.

The goal of the experiments are to prove/improve our model of a rocket in an aerodynamic environment with statistical methods, Monte Carlo algorithms, and numerical integrators. Data is to be collected from the vehicle in flight and the characteristics of its performance will be compared to the results of the simulations. Iterative methods will be used to estimate an average behavior of the system and approximate uncertainties of those behaviors. Since RK4 is fast, computational simulation time can be very efficiently used to estimate system performance according to the model, and data collected will have a greater impact on validating/disproving our model.

The Physical Model

The analysis of a rocket trajectory requires a precise description of the forces acting on the body. The exhaustive list for the purpose of this project is shown below:

* Air resistance, acts opposite to the direction of velocity and is proportional to the square of the vehicle speed. The quadratic air drag model will be used:

$
F_drag = 1/2 r v^2 C A
$

* Gravity, always points toward earth's center which is the local negative y coordinate

* Thrust force, point in the same direction as velocity but its magnitude is time and motor dependent. Thrust curve data will either be derived from experimentation or obtained from publicly available data.

* When thrust is non-zero, mass is reduced in time from the vehicle also contributing to a change in momentum. For simplicity, linear mass reduction will be assumed. The mass of an expired rocket will be measured to determine the endpoint.

* And finally, aerodynamic forces on the body from the sides induced by wind are also accounted for, and operate under normally distributed random processes. The quadratic air model will again be used.

Using Newton's second law, it follows that

$$
sum \vec{F} = \vec{T(t)} - \vec{F}_{\text{Drag}} - \vec{F}_{g} - \vec{F}_{\text{Aero}} = m(t) \cdot \ddot{\vec{r}}
$$

This is the differential equation the simulator aims to solve.

Motor Model

The rocket uses an Estes A8 solid propellant rocket motor. Thrust curve data is publicly available and free for use. Our team did not collect the data, but we use to it model the performance of the motors in our experiments. We use linear interpolation to create a continuous representation of the thrust curve and incorporate small deviations from the data with Monte Carlo methods according to the uncertainties in the motor design. For our purposes, simulations use $d{T} = +/- 1$% up to $d{T} = +/- 5$%.

If time permits, and if we need to optimize our computational model, we can generate our own thrust curve data by positioning the engine in a wood block on top of a force sensor. We can measure the force as a function of time and take the average of several acquisitions.

Atmospheric Model

We use the NASA tropospheric model. Since we are operating at low altitudes, this will be sufficiently accurate for the case of modeling air density, and thus drag forces.

Wind Model

Our wind model consists of a simple normally distributed wind gust model and a small continuous form breeze in a constant direction. Monte Carlo will also be used for computing a random initial direction for the wind. Quadratic air drag will once again be implemented.

Uncertainty Propagation

Uncertainty will be computed using iterative methods. Since RK4 is so fast and flight times will likely last less than 10 seconds in real time, computation time is extremely small per iteration. To approximate our expected performance parameters (max altitude, flight time, landing zone, etc), the simulation will be run hundreds of times and averages over them all with standard deviations representing the associated uncertainties. Uncertainties of the following variables will be included in the simulation:

* Initial wind speed and direction
* Density of air
* Cross-sectional area of the rocket (across lateral and y axes)
* Thrust force
* Parachute deployement time
* Mass of the rocket (prior to and after fuel expenditure)
* Initial angle of launch

Materials List

* Model rocket
* Estes A8 motors, ~10 of them
* Computer with Python 3.13 installed
* Timer
* Accelerometer sensor, barometric pressure sensor, Arduino Nano, and SD card logger
* Soldering iron
* Launch supplies (launch pad, igniter for motors)
* Lab notebooks and pens

* Wood block and force sensor

* Protractor

* GPS for measuring horizontal travel distance (phones might suffice)

Expected Results

We expect our experimental results for flight time, maximum altitude, and horizontal distance traveled to pass a Chi-Squared Test (
a = .05
) when compared with our Monte-Carlo simulation. Further refinement with our wind model may be necessary if time permits.

Timeline

We expect to have completed our computational model by the 20th of March, after which we will collect our computational data. We will then, if time permits, acquire thrust curve data using a force sensor. During the following week, we will monitor weather and select an optimal launch day. We expect the setup, launch, and data acquisition to take no more than three hours. The next week will be dedicated to data analysis, model refinement, and re-acquiring data if necessary.

ThrustVectorControlSim

Dynamics problem with a discrete PID controller to balance a uniform beam.

ThrustVectorControlSim

README:

Inverted Pendulum Problem

This project aims to implement a PID controller to solve the inverted pendulum problem for variable mass systems.