User Guide to OSCaR and Other MEX Simulations

Masters of Engineering Project

By:

David Hoddinott

Graduate Advisor:

Kurt Anderson

# Purpose:

This guide accompanies the report “Comparison of Control Methods for a Multibody CubeSat”, written in the Fall of 2020 as a Master’s project for Kurt Anderson’s OSCaR project. It was observed that the software written for this project could be useful for other students completing similar projects. This guide has two purposes; to demonstrate how to use the code written for the original project, and to provide an overview of how it could be modified for applying control schemes to alternative models.

Table of Contents

[Purpose: 2](#_Toc62488702)

[Quick Guide to running an OSCaR Simulation 3](#_Toc62488703)

[Modification Guide 5](#_Toc62488704)

[Note on Terminology: Model and Simulation 5](#_Toc62488705)

[Overview 5](#_Toc62488706)

[Explanation of Individual Functions and Subsystems 7](#_Toc62488707)

[Model Input Organizer 7](#_Toc62488708)

[MEX Model 10](#_Toc62488709)

[Control Moment Calculator 10](#_Toc62488710)

# Quick Guide to running an OSCaR Simulation

Before running an OSCaR simulation, ensure that the version of MATLAB being used is up to date (As this was written in 2020a, it is recommended that the version of MATLAB used is the same or newer). Additionally, several MATLAB toolboxes must be installed. These are listed below:

* Simulink
* Control Systems Toolbox
* DSP System Toolbox
* Aerospace Toolbox

Additionally, ensure that you have the relevant file folders from this project.

To run a OSCaR simulation, first decide on which type of control scheme you would like to use (Extended Linear Quadratic Regulator AKA Kalman Filter, Proportional Integral Derivative, or none). Open the relevant file folder, and locate the “input.xlsx” file, found in the same folder as the rest of the code. It should look something like this:

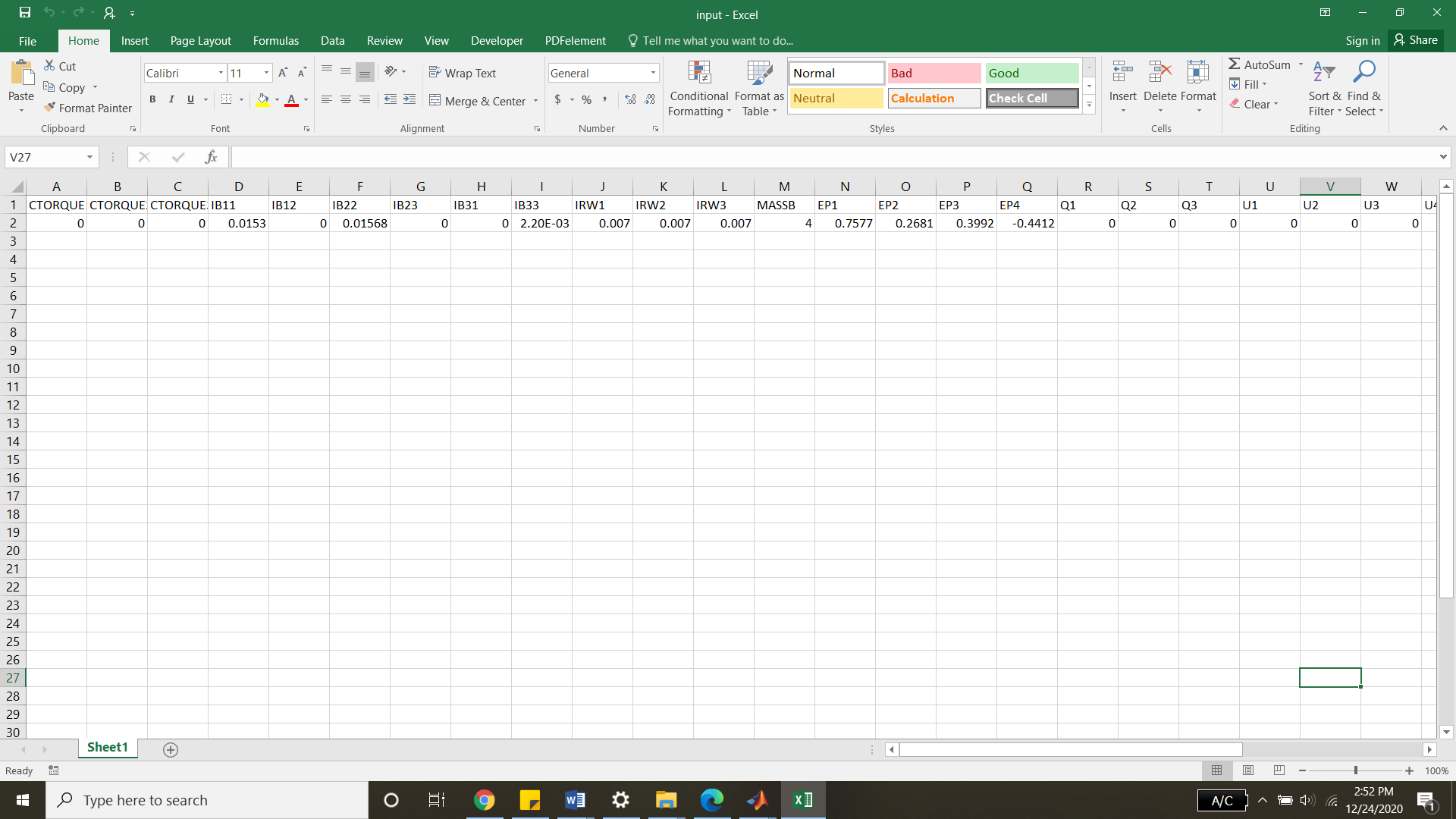


Figure 1, Sample Inputs

Change the parameters to the desired values. These are the initial simulation values. Physical properties (Moments of inertia, mass, etc.) do not vary during the simulation. The attitude and angular velocities will.

Next, locate the s*imulink\_run\_file.m.* It will appear like so:

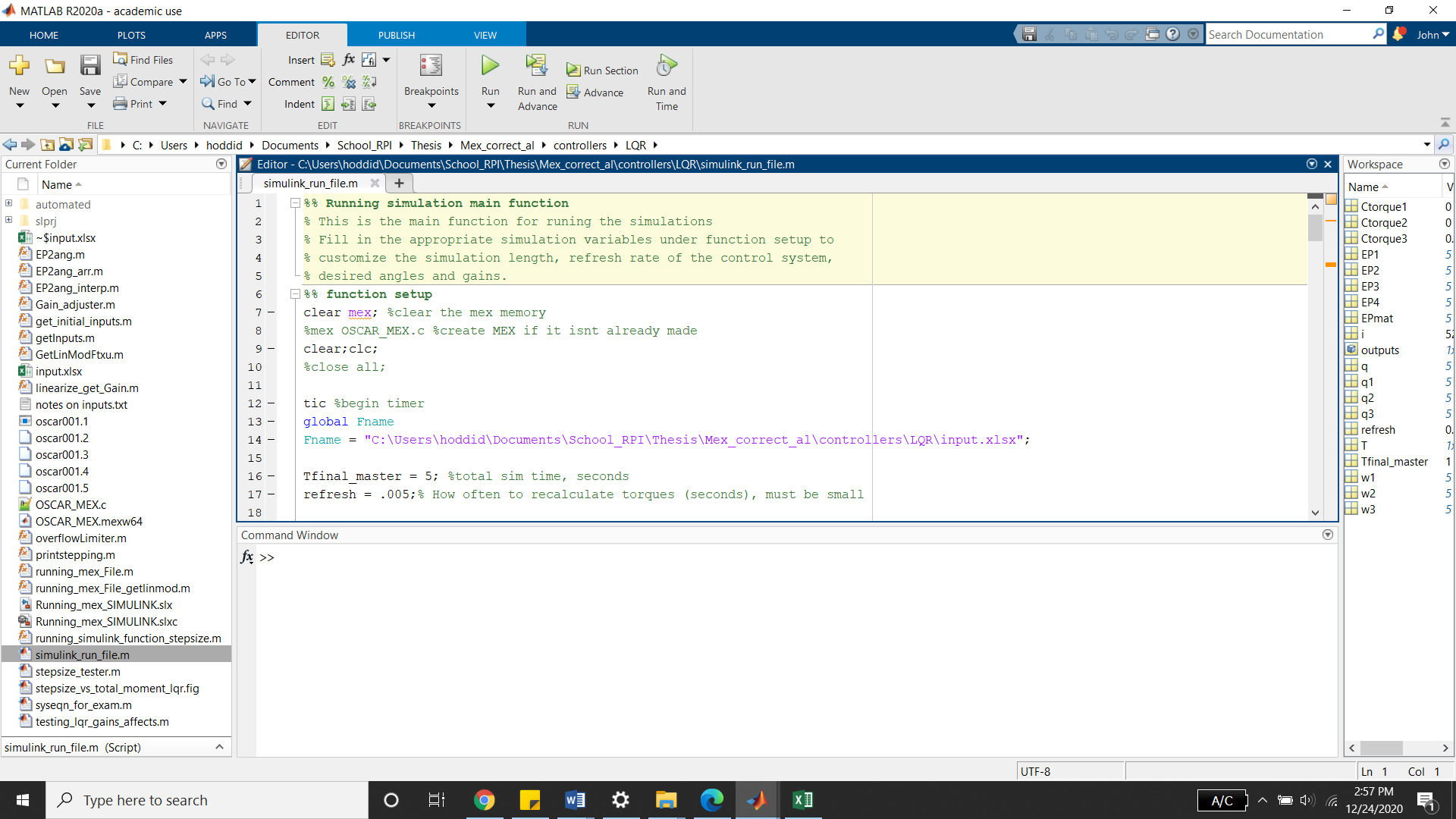


Figure 2, Simulink Run File

Verify that the global variable *Fname* matches the location of the input file, if it does not, copy and paste the file location and name into line 14.

Next, change all of the relevant variables under the section *Function setup*. This includes the Final simulation run time, the refresh rate, any controller gains, the initial torques, the maximum torque output of the system, the desired angles, and the desired angular velocities. Note that most control schemes will fail if the refresh rate (the rate at which the control system updates the control inputs) is too low.

Finally, simply run the function. An approximation of the time that the model will take to run will be printed to the MATLAB console (note, this will vary with model complexity and the computer used to run the simulation). After the simulation has completed, an alarm will sound, and graphs of the generated moments, angular velocities, and Euler angles will be generated.

# Modification Guide

The code written for the original project was designed to control the attitude and angular velocities of a CubeSat. The overall framework for this project could be used to control many systems, and thus this portion of the guide is an overview of the software, given in order to aid in modifying this code to model/control other systems.

## Note on Terminology: Model and Simulation

In this guide, the terms model and simulation are used extensively. This is to distinguish between the MEX model being run, and the MATLAB/Simulink scripts and functions being used to run the MEX model and create control inputs. The term model refers to the MEX function, which contains the dynamics for the system, and the term simulation refers to the MATLAB/Simulink code.

## Overview

The core of this system is a MEX (MATLAB Executable Function), which is run within Simulink. The MEX model used was specific to OSCaR, and it can be replaced with any function. One powerful option is to create the system you wish to model/control in Autolev, and then transform the auto generated C code from this into a MEX function. See the guide “Merging Autolev (C) and MATLAB‘’ for an overview on how to perform this. The remainder of this guide will be an overview of how the Matlab/Simulink code functions.

The MEX model used will have constant control moments applied over the entirety of the time the model is run, which is typically a short time (and is specified by the user). This short amount of time is the Refresh rate, often shortened to ‘Refresh’. The model will run for the specified amount of time, using the constant control inputs specified, then return the new states of the system (in this case, attitude and angular velocity). A control input is then calculated using the control law created by the user. These control inputs, along with the output states of the model are then used as inputs for the next instance of the model, and the cycle repeats until the total amount of time the user has specified the model will run has been reached. This overview can be seen in flowchart form in Figure 3.

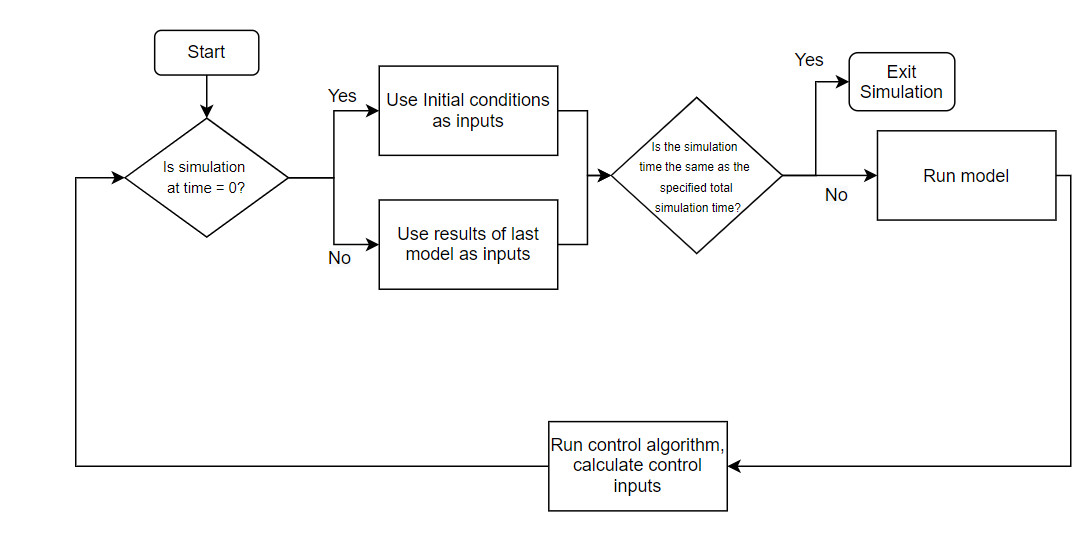


Figure 3, Model Pseudocode Flowchart

This flowchart is recreated in Simulink, using Simulink built-in blocks as well as custom MATLAB functions. The Simulink Equivalent of this flowchart can be seen in Figure 4. This Simulink diagram, shown below in Figure 4, uses an Extended Linear Quadratic Regulator.

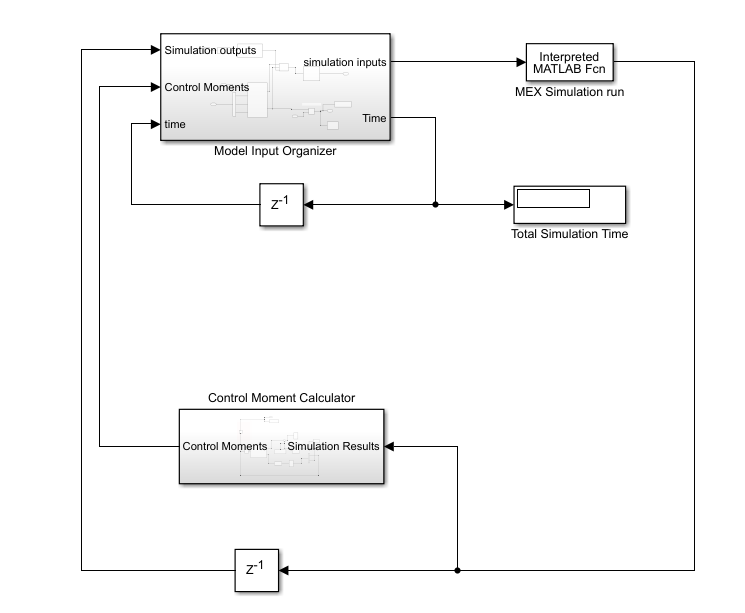


Figure 4, Simulink Diagram

## Explanation of Individual Functions and Subsystems

### Model Input Organizer

This subsystem is used to fetch the initial function values, as well as combine the calculated moments and results from the previous simulation into a form that the MEX function is able to use as an input. It also checks if the total simulation instance time has reached the total time specified by the user, and exits the simulation if it has. The expanded subsystem can be seen in Figure 5.

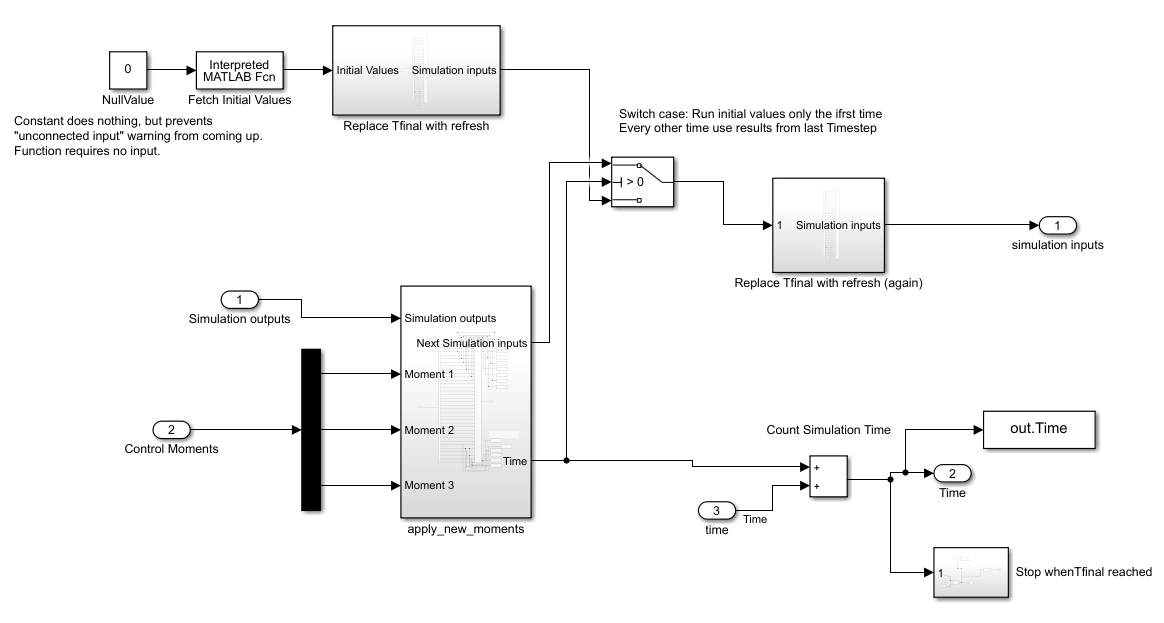


Figure 5, Model Input Organizer

The section seen in Figure 6 is responsible for fetching the initial values. This is done via a MATLAB function, which parses an excel file. This is run in a MATLAB interpreted function because Simulink is not capable of parsing excel files via the normal MATLAB function command. This function needs the file location for the excel file, however Simulink lacks the capabilities to use a string as an input. Thus, this is set as a global variable in the workspace.

Another minor complication is that the MATLAB interpreted functions always have an input port. To avoid the warning message that nothing is attached to this input, a value of zero is used as an input, but never used. The final simulation time is then replaced with the refresh rate.

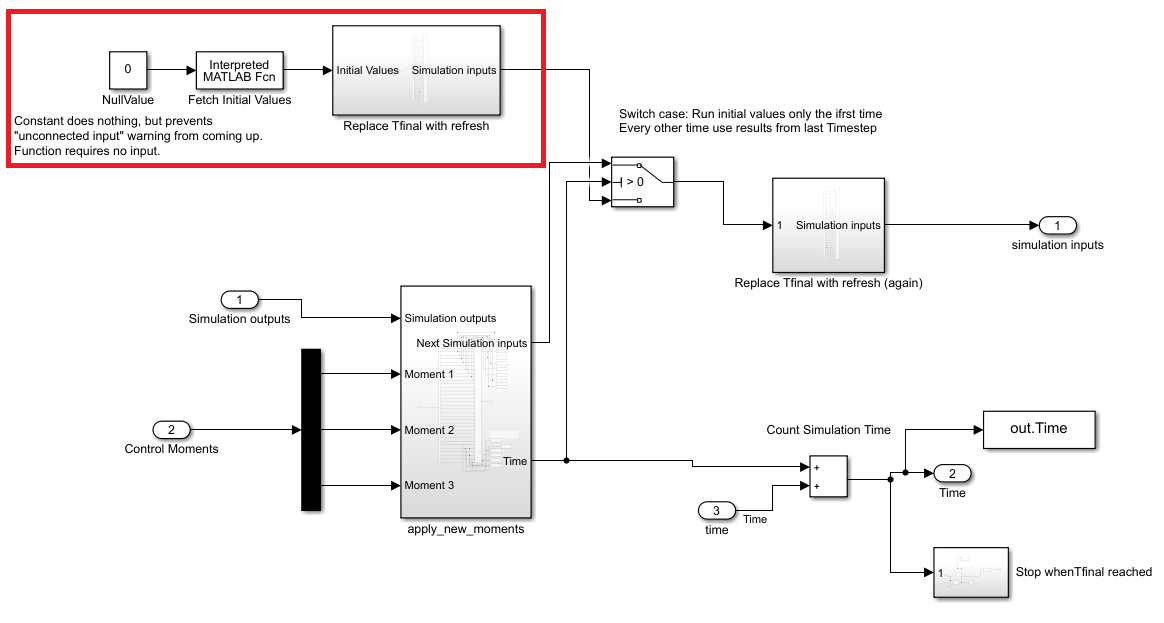


Figure 6, Initial value fetching in Model Input Organizer

The section seen in Figure 7 is responsible for fetching the results of the last simulation and combining it with the calculated control inputs to a form an array that can be used as an input to the MEX model.

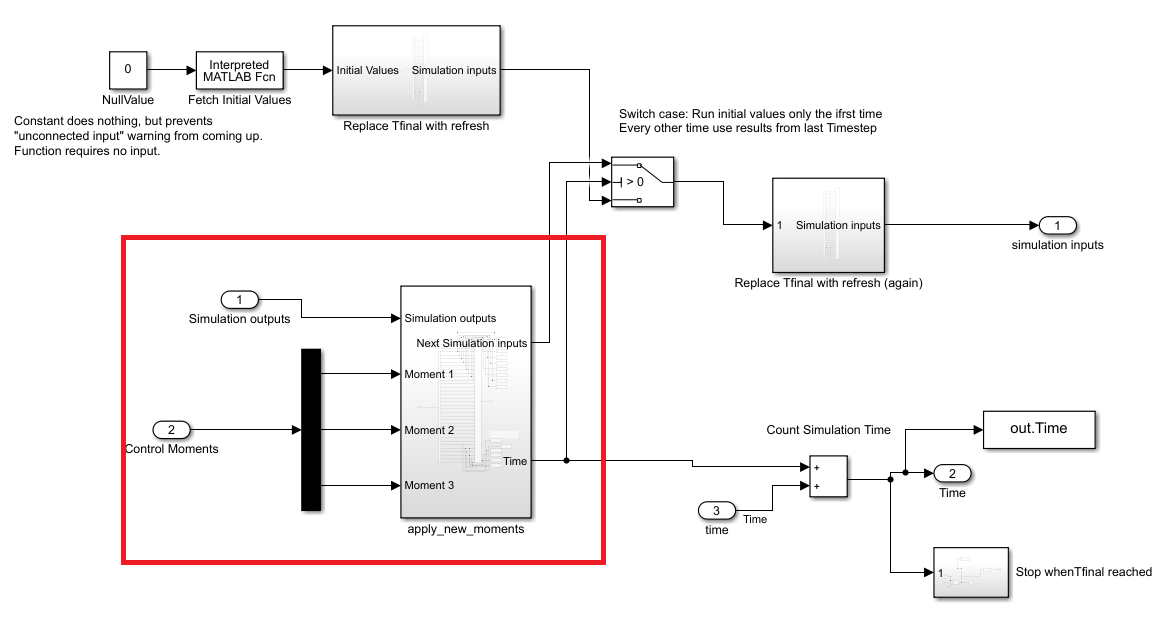


Figure 7, Results of the Last Model in Model Input Organizer

The sections in Figure 6 and Figure 7 are then fed into a switch. If the simulation time is equal to zero, the initial simulation parameters are used as inputs to the model. Else, the results of the last model outputs are used as inputs. The final model time parameter is then set to be the refresh rate, and these parameters are then sent to the model.

The final task that this subsystem completes is keep a count of the total simulation time, store this time to the workspace, and exit the simulation when the final time has been reached. This section is shown in Figure 8

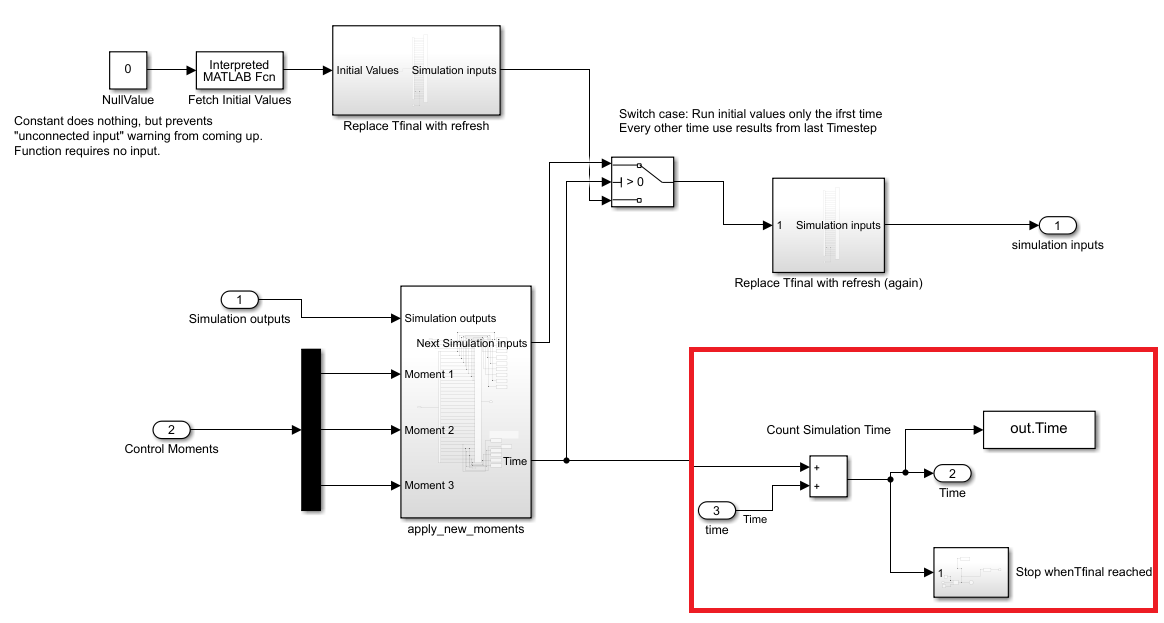


Figure 8, Exiting the Simulation When Final Time is Reached

### MEX Model

After the input parameters have been set by the Model Input Organizer subsystem, an Interpreted MATLAB Function runs the MEX model. For more detailed information about how MEX models work, and how they can be created for Multibody systems, see the guide, “Merging Autolev (C) and MATLAB”.

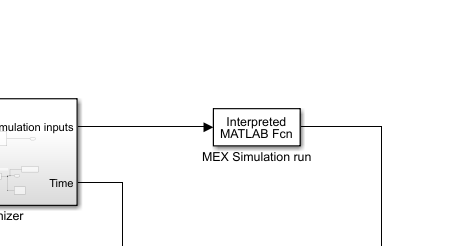


Figure 9, MATLAB Function Which Runs the MEX Model

### Control Moment Calculator

The final subsystem to be examined is the Control Moment Calculator, highlighted in Figure 10. This function is responsible for creating the control inputs that bring the system towards the desired states.

This subsystem will be different for different the control laws. Two example control laws will be shown, an Extended Linear Quadratic Regulator, as well as a Proportional Integral Derivative law.

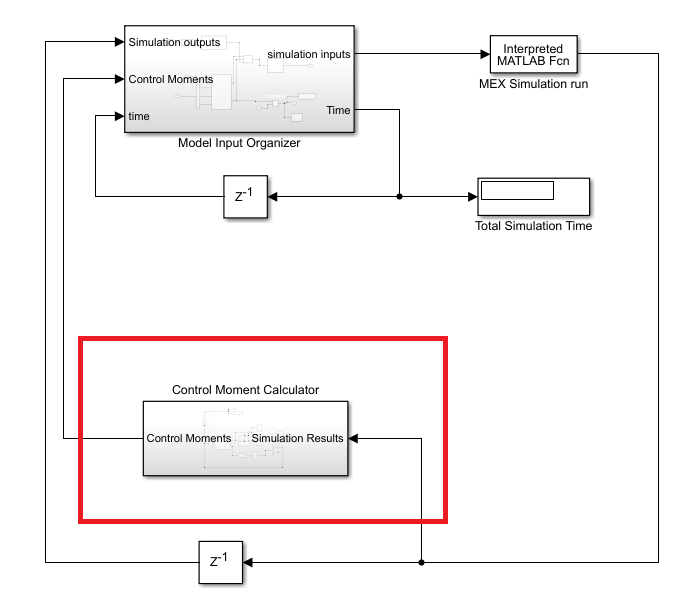


Figure 10, Control Moment Calculator

#### Extended Linear Quadratic Regulator Control Subsystem (ELQR)

The first control law subsystem to be examined is the Extended Linear Quadratic Regulator. For the specifics on how this control law functions, see the report this guide accompanies, “Comparison of Control Methods for a Multibody CubeSat”. The expanded subsystem can be seen in Figure 11.

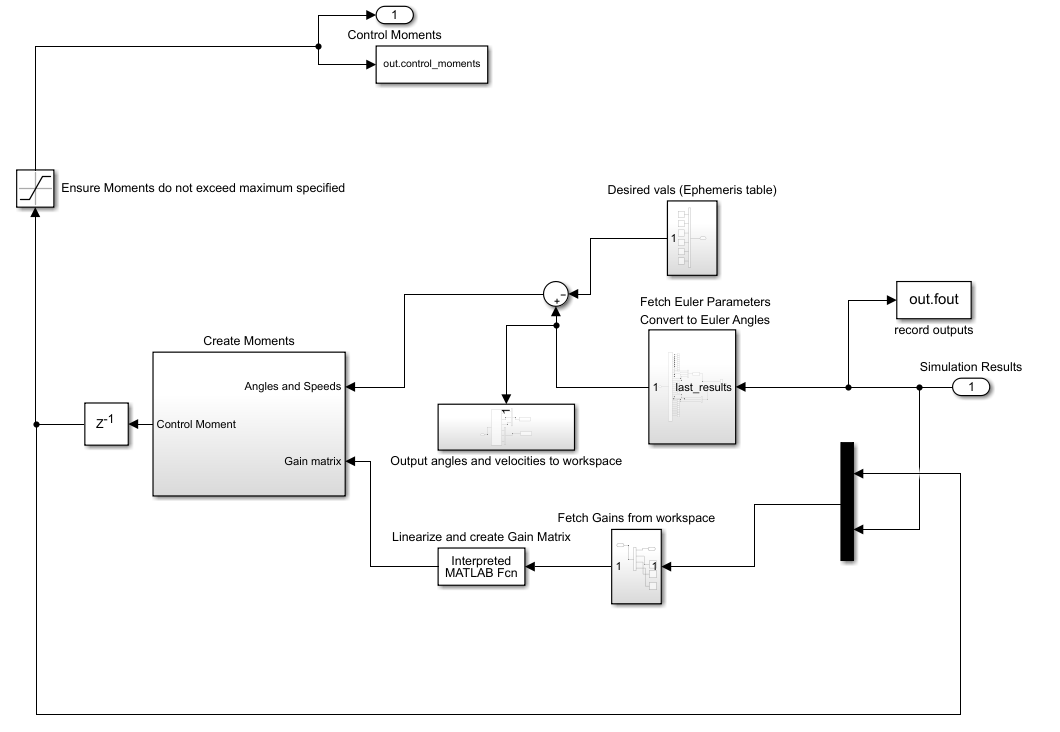


Figure 11, Creating Control Moments Subsystem

Note that this subsystem has inputs on the right, and outputs on the left (compared to the Model Input Organizer subsystem, which is the reverse). The subsystem takes in the results of the last model, and then separates it into two paths. The top path (highlighted in Figure 12) fetches the Euler Parameters and angular velocities from the model results. The Euler parameters are then transformed into Euler angles, and then the desired angles/angular velocities are subtracted from these to find the error of the states.

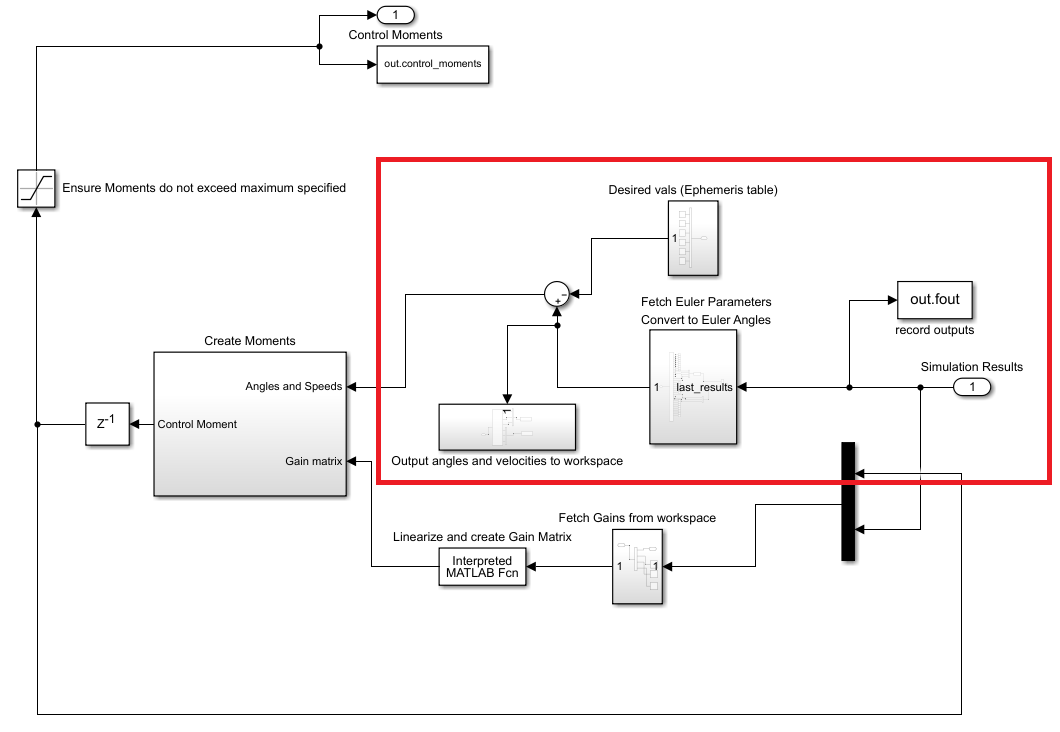


Figure 12, Top section of Control Moment Calculation subsystem

The lower part of this subsystem uses the results of the last model and the previous control moments to create a gain matrix which, when multiplied by the states error vector, will create a vector of control moment. This is performed in the Interpreted MATLAB Function block ‘Linearize and Create Gain Matrix’. This function first linearizes about the current state. Then, MATLAB’s built in Linear Quadratic Regulator function is used, along with user defined gains in order to create a gain matrix. These control moments then go into the Model Input Organizer subsystem, and the simulation continues.

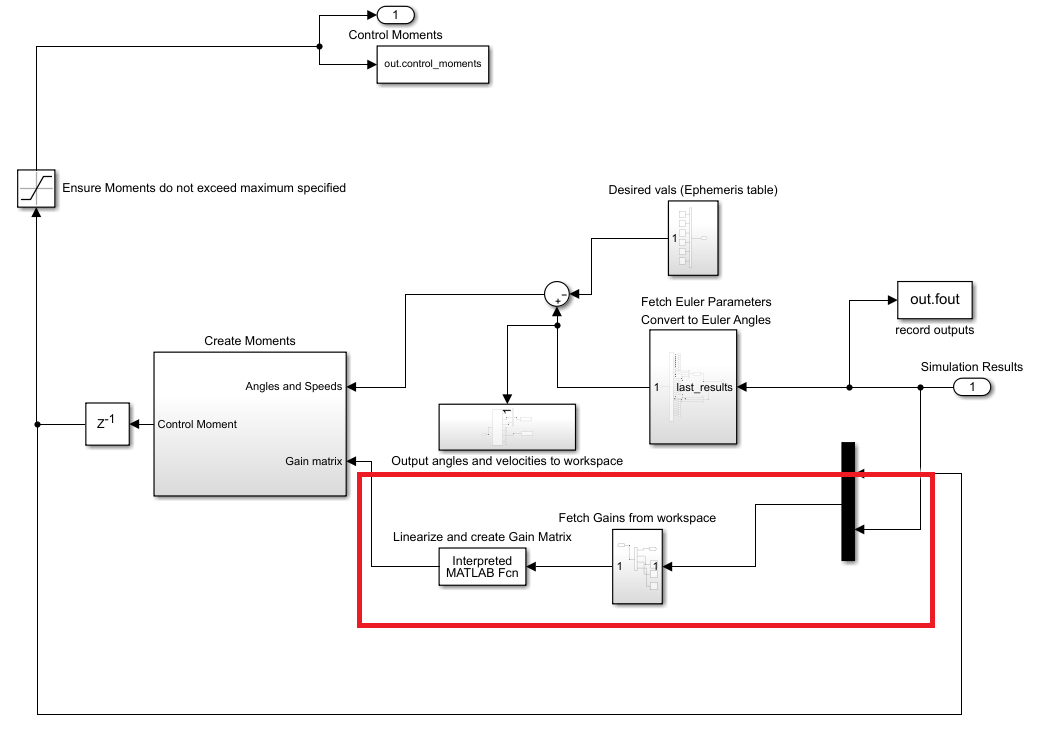


Figure 13, Lower Section of Control Moment Calculation Subsystem

The gain matrix previously created is then multiplied by the state error vector in the subsystem ‘Create Moments’. This creates the control moment vector. The moment vector is then set such that it does not exceed the maximum moment that can be applied to the system. The control moments are saved to the workspace and then exit the subsystem to be used in the Model Input Organizer subsystem.

#### Proportional Integral and Derivative Control System (PID)

The second control law subsystem to be examined is the PID controller. For the specifics on how this control law functions, see the report this guide accompanies, “Comparison of Control Methods for a Multibody CubeSat”. The PID control subsystem replaces the E ELQR Control Moment Calculation subsystem, and the rest of the model remains the same, with the exception that the PID controller requires the simulation time as an input as well. This is because the integral part of PID control needs to calculate the integral of the error over time. This change can be seen below in Figure 14.

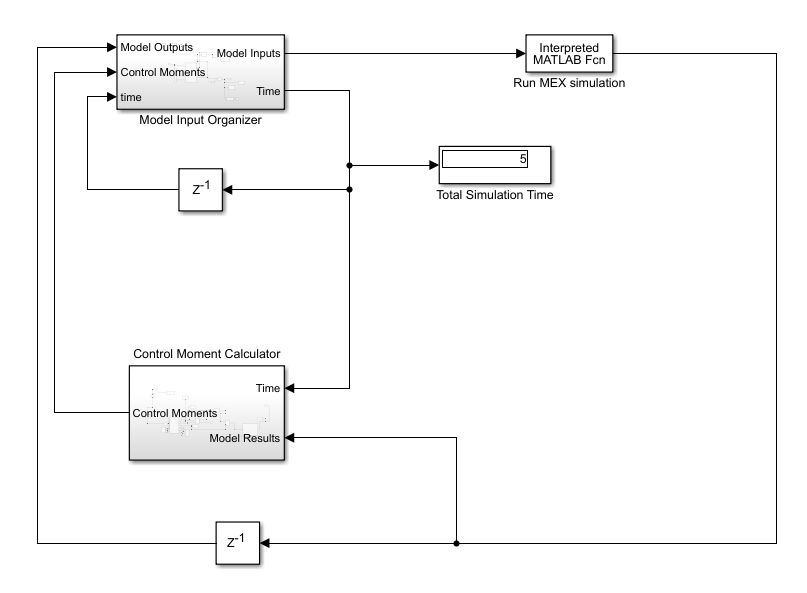


Figure 14, Full Simulink Diagram for PID System

The Control Moment Calculator subsystem was modified such that a PID controller was used. First, inputs to the PID controller are created/organized. The system states are fetched from the model outputs. The Euler Parameters are then converted to Euler angles, which are recorded to the workspace, along with the angular velocities. The desired angles are then brought in from the workspace. The integral of the error is calculated via a buffer block, which stores a certain number of time and angular error values. These values are integrated to find the integral of the angular errors.

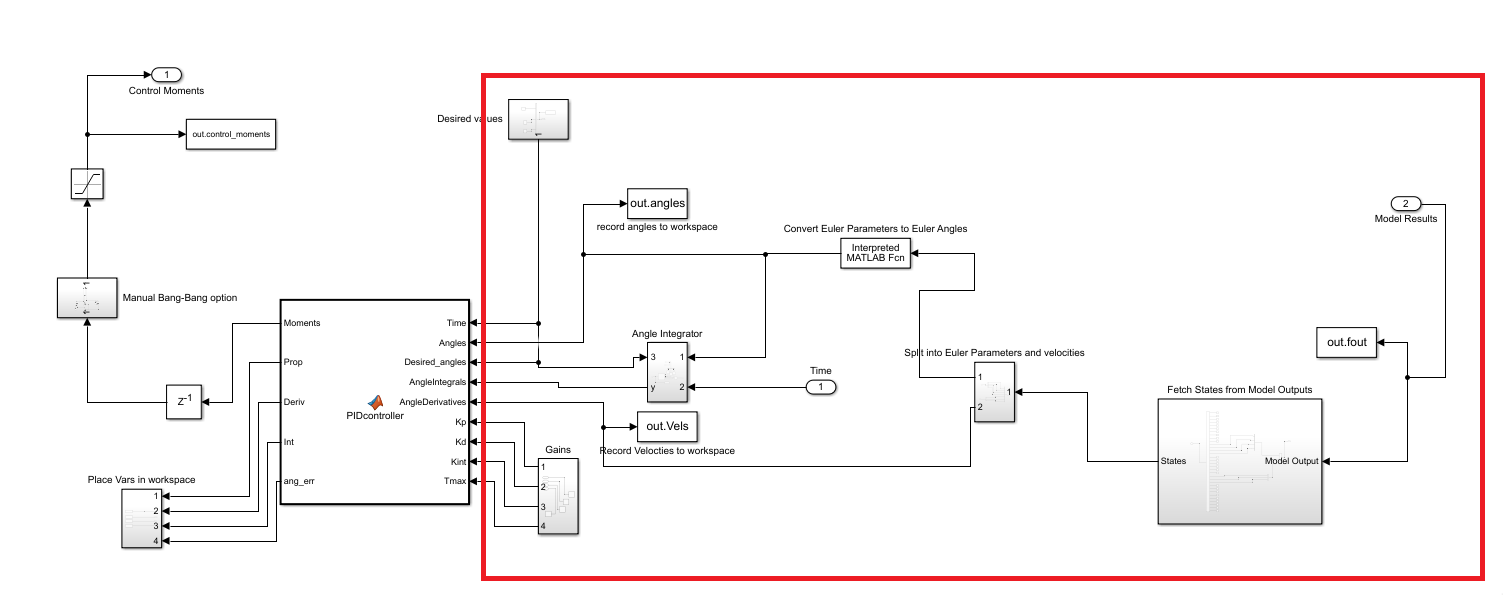


Figure 15, Organizing Inputs to the PID Controller

The PID controller then takes in the Euler angles, the desired Euler Angles, the integral of these angles, the derivative of the Euler Anglers (the angular velocities), as well as the Proportional, Integral, and Derivative gains.

The controller then creates a control moment. This moment is limited such that it does not exceed the maximum output torque set by the user. Additionally, the individual parts of the control moment are saved (e.g. how much of the total torque came from proportional vs integral vs derivative parts).

A manual Bang-Bang control subsystem is also included. This allows the user to switch on and off the control moment in different axis, to see how the system responds, as well as to experiment with the required velocity cutoffs that an automatic Bang-Bang control system would require. The control moments are then sent to the Model Input Organizer subsystem, and the simulation continues.

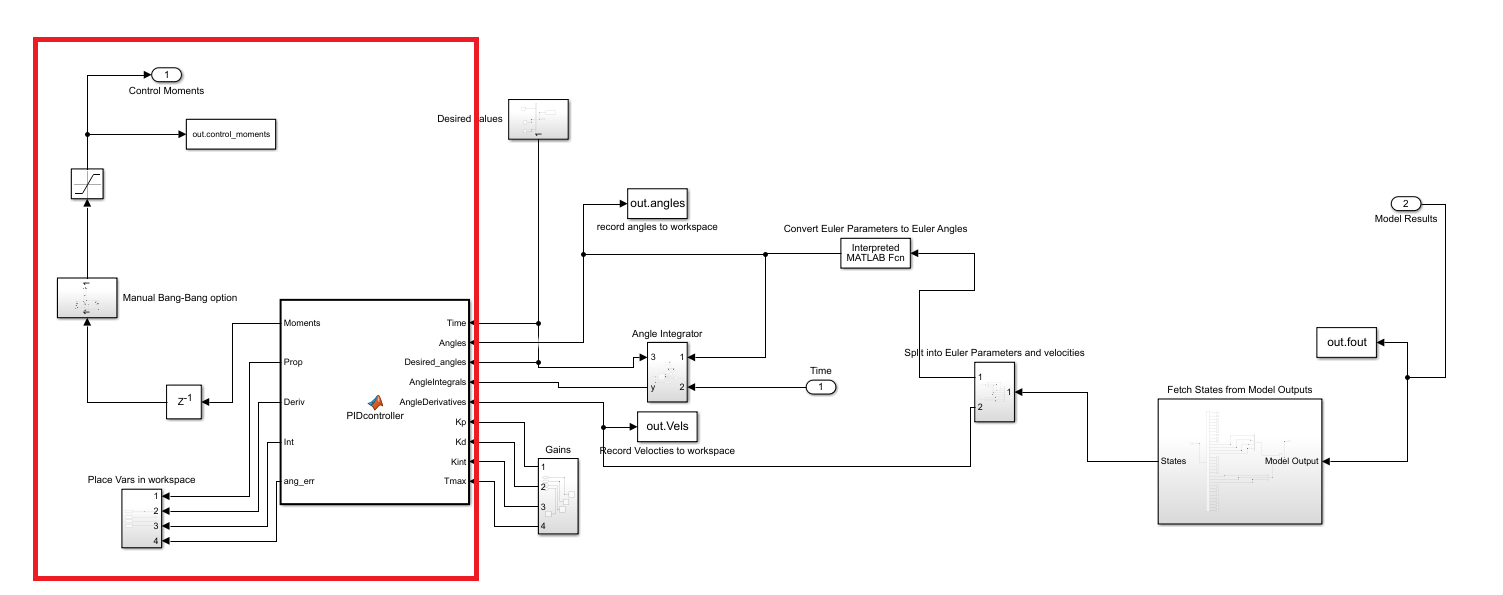


Figure 16, PID controller and Bang-Bang option