Comparison of Control Methods for a Multibody CubeSat

Masters of Engineering Project

By:

David Hoddinott

Graduate Advisor:

Kurt Anderson

# **Executive Summary**

The motivation behind this project was to compare control algorithms for the attitude and angular velocity of a CubeSat called OSCaR (Obsolete Spacecraft Capture and Recovery). OSCaR’s intended function is to rendezvous with, capture, and de-orbit obsolete spacecraft and debris, in order to combat Kessler Syndrome. PID (Proportional Integral Derivative) and ELQR (Extended Linear Quadratic Regulator) controllers were compared, and it was found that while both controllers were effective, PID control performed better for controlling the attitude, and ELQR performed slightly better for de-tumbling.

Contents

[**Executive Summary** 2](#_Toc62778783)

[**Introduction** 5](#_Toc62778784)

[Kessler Syndrome 5](#_Toc62778785)

[OSCaR 5](#_Toc62778786)

[Motivation for Analysis 5](#_Toc62778787)

[**Analysis Overview** 5](#_Toc62778788)

[System Description 5](#_Toc62778789)

[Control Schemes 6](#_Toc62778790)

[Mathematical Model 8](#_Toc62778791)

[Generalized active Forces: 8](#_Toc62778792)

[Generalized Inertial Forces 8](#_Toc62778793)

[Forming Kane’s Method 9](#_Toc62778794)

[Assumptions and Limitations of mathematical model 9](#_Toc62778795)

[Implementation of the Mathematical Model 9](#_Toc62778796)

[**Results:** 9](#_Toc62778797)

[PID Control 10](#_Toc62778798)

[Overview 10](#_Toc62778799)

[Sample simulations for Angular Errors 10](#_Toc62778800)

[Sample Simulations for Velocity Errors 11](#_Toc62778801)

[ELQR Control 12](#_Toc62778802)

[Overview 12](#_Toc62778803)

[Sample Simulations for Angular Errors 13](#_Toc62778804)

[Sample Simulations for Velocity Errors 14](#_Toc62778805)

[Bang-Bang Control 15](#_Toc62778806)

[Effect of Step Size 15](#_Toc62778807)

[**Conclusions** 18](#_Toc62778808)

[Future work 18](#_Toc62778809)

[Increasing Linearization accuracy 18](#_Toc62778810)

[PID Controller Measuring Angular Velocity 18](#_Toc62778811)

[Decreasing Computational Time via Variable Step Sizing 18](#_Toc62778812)

[Implementing Model-Based Predictive Control 18](#_Toc62778813)

[Dynamically Optimize Gains 19](#_Toc62778814)

[**Citations** 20](#_Toc62778815)

[**Copy of AUTOLEV file** 21](#_Toc62778816)

[Agena-Gemini system 21](#_Toc62778817)

[**Autolev, C and MATLAB tutorial** 23](#_Toc62778818)

[Overview 23](#_Toc62778819)

[Requirements 23](#_Toc62778820)

[Setup 23](#_Toc62778821)

[Run your Autolev code 23](#_Toc62778822)

[Modify Autolev Code for MEX conversion 23](#_Toc62778823)

[Creating a MEX Function from Auto Generated C Code 24](#_Toc62778824)

[Overview: 24](#_Toc62778825)

[Modification: 24](#_Toc62778826)

[Headers 25](#_Toc62778827)

[Main Function Declaration 25](#_Toc62778828)

[Declaring Inputs and Outputs 26](#_Toc62778829)

[Notes on printf and exit(0); 26](#_Toc62778830)

[Replacing inputs 27](#_Toc62778831)

[Replacing Outputs 28](#_Toc62778832)

[Final Words on Converting from C to MEX 31](#_Toc62778833)

[Using MATLAB to Run MEX Code 31](#_Toc62778834)

[MATLAB Setup 31](#_Toc62778835)

[Simple Example 32](#_Toc62778836)

[Basic Implementation Notes 32](#_Toc62778837)

[Simulink 33](#_Toc62778838)

[Introduction 33](#_Toc62778839)

[MEX and MATLAB Interpreted Functions 33](#_Toc62778840)

[Basic Example 34](#_Toc62778841)

[Conclusion 36](#_Toc62778842)

# 

# **Introduction**

## Kessler Syndrome

Kessler syndrome is a theoretical scenario where the density of space debris/objects in Low Earth Orbit (LEO) is sufficiently high that collisions between debris could cause a cascading effect where each collision generates more debris, which cause further collisions. A potential result of this is that the density of the debris field in LEO becomes sufficiently high that the orbit becomes uninhabitable for spacecraft until the debris naturally de-orbit. This is an undesirable outcome, as spacecraft in LEO provide many vital functions, including communication, research stations, and surveillance.

## OSCaR

The most obvious solution to preventing or slowing the onset of Kessler syndrome is to prevent space debris creation via including de-orbit capabilities on crafts going into LEO. Indeed, the International Telecommunications Union rules dictate that spacecraft must have a system in place for disposal of spacecraft at the end of their life (either by boosting into a graveyard orbit, or via controlled atmospheric reentry systems [1]), and the United States Federal Communications Commissions have similar rules [2]. However, many defunct satellites still remain in LEO without the ability to safely de-orbit or boost into a graveyard orbit. These satellites remain as possible sources for an influx of debris if they collide.

In order to remove these satellites, Kurt Anderson and his team have been developing OSCaR (Obsolete Spacecraft Capture and Removal), a 3U automated CubeSat which will be capable of maneuvering to, capturing, and de-orbiting these spacecraft (or other debris) by means of an electromagnetic tether. By removing these potential triggers, OSCaR could contribute to the prevention or delaying of Kessler syndrome.

## Motivation for Analysis

In order to function, OSCaR must be able to control both its attitude and angular velocity. The purpose of this project was to use a multibody model of OSCaR to compare two control algorithms, Proportional Integral Derivative (PID), and an Extended Linear Quadratic Regulator (ELQR). The ELQR control is very similar to an extended Kalman filter, which is used on spacecraft. However, because the states (attitude position and angular velocities) are calculated, known values, the control scheme is a regulator rather than an estimator. Both control schemes are tested, and sample simulations are analyzed.

# 

# **Analysis Overview**

## System Description

OSCaR is a 3U CubeSat. This means that the geometric properties of OSCaR are preset to be 30x10x10 centimeters, with a maximum mass of 4 kilograms [3]. The Inertial properties of OSCaR were determined in an undergraduate Capstone project [4]. The products of inertia were assumed to be zero as they have small magnitudes when compared to the moments of inertia about the primary axis. OSCaR is defined in a multibody model. OSCaR itself is assumed to be rigid, with the reaction wheels measured as separate bodies within it.

*Table 1, Moments of Inertia*

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| Moment of Inertia () | 1.53e-2 | 1.568e-2 | 2.28e-3 |

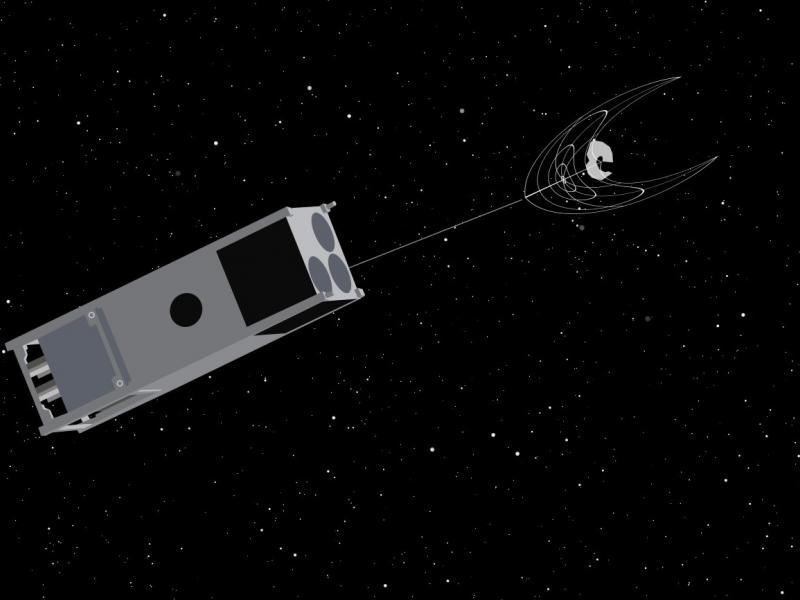


Figure , OSCaR [5]

### Control Schemes

#### PID

Two control schemes are to be analyzed for this project. The first is a PID system, which has a control law of the form:

Where is the control effort (in this case, the moment applied by the reaction wheels) and is the error between the current controlled state and the desired position of that state. are the gains for the proportion (size) of the error, the integral of the error, and the derivative of the error. These gains are chosen by the user. For this project, was always the error between the desired angular position, and the current angular position.

#### Extended Linear Quadratic Regulator

The second control system to be analyzed is an extended Linear Quadratic Regulator. The ELQR works via linearizing the model about the current state by placing it into state space form. This simplifies the equations to the form:

Where is the current state vector (angular position and velocity), is the most recent control vector, and is the derivative of the state vector. In this case, these are the control moments. In OSCaR’s case, is the angular velocities and accelerations. The Autolev model returned the angles and angular velocities, but not the accelerations. For simplicities sake, the angular acceleration was calculated in MATLAB via a numerical differentiation of the angular velocity. However, in future models, including the angular acceleration as an Autolev output could be useful for decreasing computation time and increasing accuracy. and are the matrices of the partial derivatives, with being the partial derivatives of the states, and being the partial derivatives of the control efforts.

Once these matrices are calculated, they are used as inputs to MATLAB’s included *LQR* function. Below is a brief overview of how this calculates a gain matrix.

First, the following cost function is defined:

*N* is set by the user (usually zero). and are supplied as inputs in the forms of gains. is a scalar, and is a matrix of the following form:

The gain will affect the amount of control effort the system uses. Each part of the matrix corresponds to a specific part of the state, and will affect the amount that the controller will prioritize that part of the state.

The *LQR* function then assumes a state feedback law , and then calculates a gain matrix, *K* which will minimize the cost function. *K* is then multiplied by the state vector to create a control input, which is then applied to the model.

## Mathematical Model

To model the system, the following representation was created, where the body in modeled in the inertial reference frame, and the three reaction wheels are within the body. Each Reaction wheel spins independently, and they are aligned with orthogonally with the OSCaR body directions

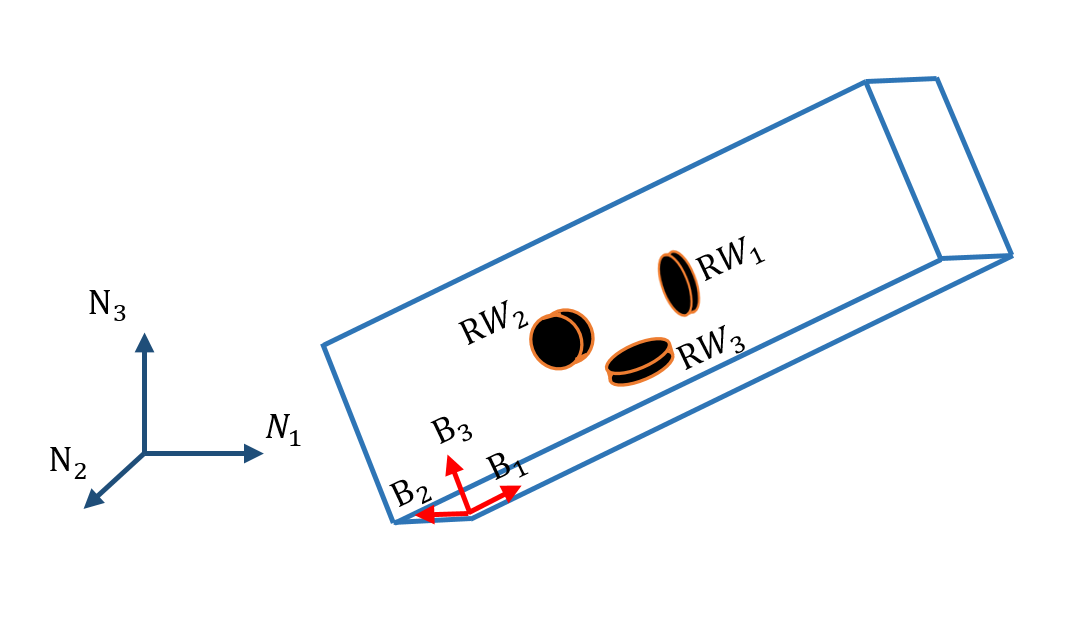


Figure , Diagram of OSCaR Multibody System

### Generalized active Forces:

The reaction wheels produce the following control moments:

With OSCaR assumed to be a rigid body, these are the only generalized active moments on the body.

### Generalized Inertial Forces

For simplicity, *o* is used to represent OSCaR, and refers to a reaction wheel (where *b* can be 1-3). The formulation of the generalized inertia forces can be seen below.

The expanded versions of this equation are not expressly written, as they are very large.

### Forming Kane’s Method

To form Kane’s method, the sum of the Generalized active and inertial forces are set to equal to zero. The equations can now be numerically integrated, and are done so using AUTOLEV version 3.4.

## Assumptions and Limitations of mathematical model

It is important to note that this model assumes OSCaR to be a rigid body. Generally, one can assume this to be reasonably accurate at low angular velocities, as OSCaR is small and stoutly constructed. This assumption will create inaccuracies in the model if the angular velocities or accelerations become large. Additionally, this model assumes that OSCaR has no propellant slosh and is not in the processing of capturing a piece of debris. Depending on the size of the debris and the length of the tether used to capture it, the effects of this debris on the dynamics of the system could be very significant, or indeed become the dominant part of the dynamics.

## Implementation of the Mathematical Model

The mathematical model described above was created in Autolev by Professor Anderson. This Autolev code was executed and the results verified. Autolev is capable of generating several different types of code. C code was generated, and then converted into a MEX (MATLAB Executable) function. As few resources exist online to aid in this, a tutorial was written, and can be found at the end of this report. This MEX function was then run using MATLAB, and verified against the original Autolev code.

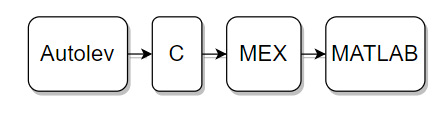


Figure , Autolev code to MATLAB flowchart

# 

# **Results:**

Both the PID and ELQR control systems were tested a variety of situations including tumbles and attitude adjustment maneuvers. It was found that both control schemes, once properly tuned, are reasonably effective for many situations. They are both perturbed by the multibody effects, and these effects worsens as the angular velocities increase. In general, PID works best for angular offsets which have small or no initial velocities. The ELQR system works better than PID when there are initial velocities in several axes (such as in a tumble), as it is able to stabilize the system with slightly less control effort and smaller oscillations.

Both control schemes fail if the angular velocities are sufficiently high that the inertial terms of Euler’s equations strongly dominate the system behavior.

## 

## PID Control

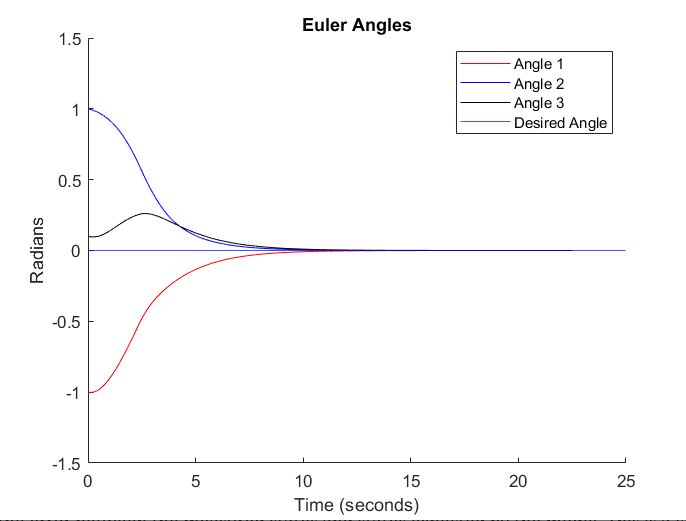
### Overview

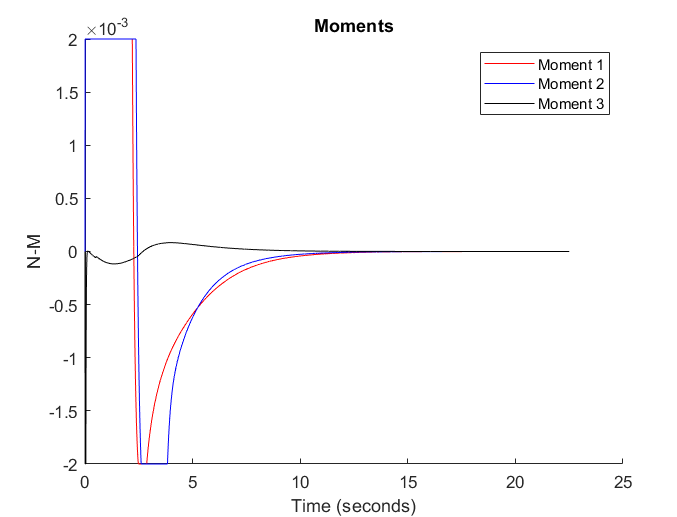
The PID system was able to stabilize a variety of initial conditions. Because the dampening in this system is negligible, it was found that the derivative term must be kept relatively high, and the inclusion of an integral term was generally not necessary. Indeed, if the system stability was marginal, the inclusion of such a term could de-stabilize the system, as the integral of the error could, in the short term, not match the correct direction that the system needed to move in order to reach the desired position. In order to stabilize systems with any significant angular velocity, the proportional term had to be lowered such that the derivative term strongly dominated the controller. If the system had little or no angular velocity, then the proportional term could be increased to prevent extremely overdamped behavior.

### Sample simulations for Angular Errors

The first simulation example is one where OSCaR is initially set to have an attitude of -1 radian in the direction, 1 radian in the direction, and 0.1 radian in the direction. There was no initial angular velocity, and the desired states were set to be the zero state (all angles are all zero). This would be similar to a small attitude change that might occur before a burn is initiated, and allows for easy graphical visualization. The gains were tuned such that the system converged to the desired state as quickly as possible.

The controller was successfully able to critically damp the and axes, but the axis remained underdamped, and no combination of gains could be found that was able to critically damp the axis without disrupting the critically damped and axes. The maneuver was fully completed in 15 seconds.



Figure , PID Attitude Maneuver (Angles)

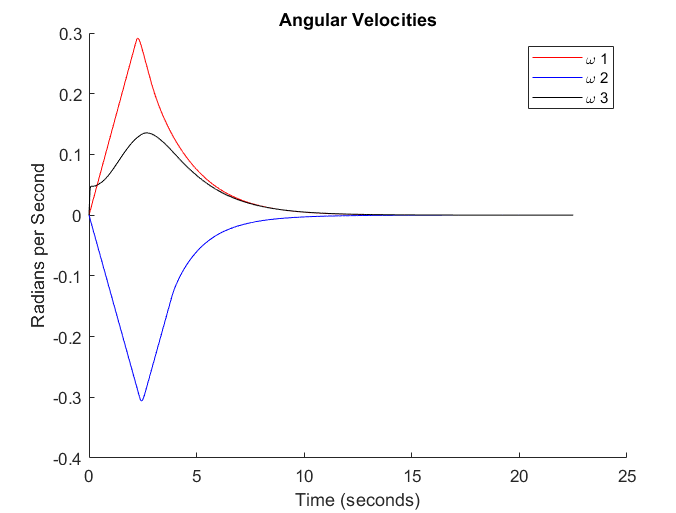


Figure , PID Attitude Maneuver (Angular Velocities) Figure , PID Attitude Maneuver (Applied Moments)

### 

### Sample Simulations for Velocity Errors

The second example simulation is a tumble. OSCaR has an initial angular velocity of 1.5 radians per second in each axis. These values are sufficiently high that the inertial terms of Euler’s equations noticeably come into effect, but not so high that the system is uncontrollable. In this scenario, the controller should stop the tumble without any regard for the final angular position. However, the PID controller only measures the angular position error, and the velocity cannot be directly controlled. Thus the desired angles are set to be zero. The PID controller was tuned for this scenario.

The PID controller was able to control the tumble in approximately 17-20 seconds, and settles to the desired states after 35.

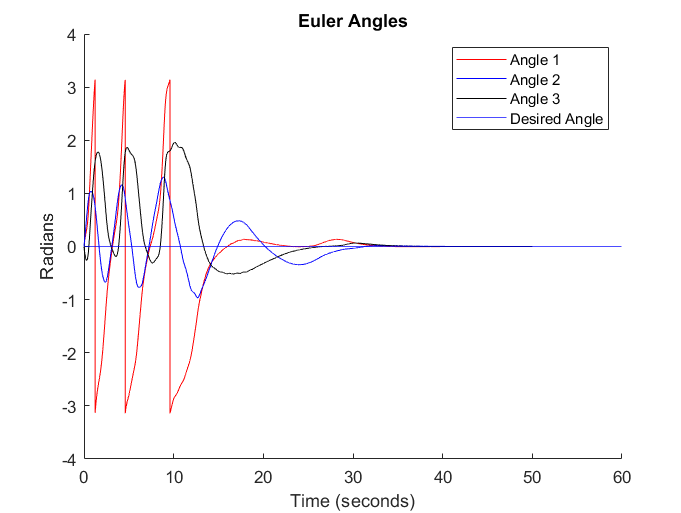


Figure , PID Stabilizing Tumble (Angles)

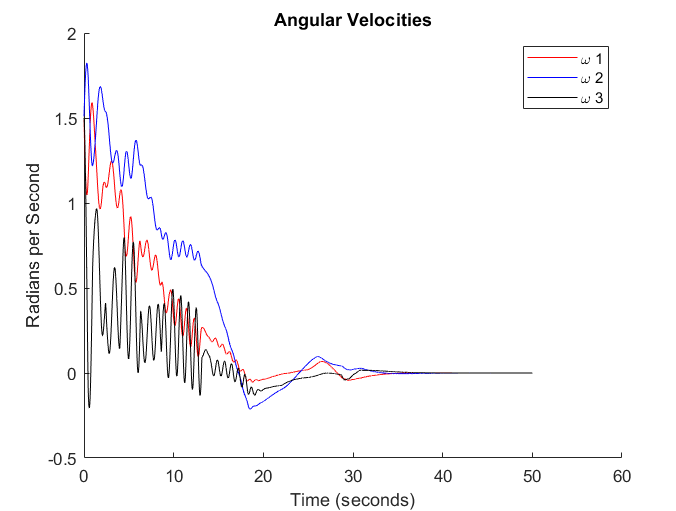


Figure , PID Stabilizing Tumble (Angular Velocities)

## ELQR Control

### Overview

The ELQR control system was also able to stabilize a variety of initial conditions. By controlling both the velocity and attitude, this was able to stabilize the tumble more efficiently than the PID system, however it was less effective at performing the angular maneuvers.

### Sample Simulations for Angular Errors

A simulation matching the angular error scenario above for the PID system was explored with the ELQR controller (OSCaR is initially set to have an attitude of -1 radian in the direction, 0.1 radians in the direction, and 1 radian in the direction, with no initial angular velocity, and the desired states were set to be the zero state). The ELQR system gains were tuned for this scenario, and the system was able to perform the maneuver successfully. The system converged to the desired states quickly in the two axes, however it was underdamped in one, and no combination of system gains could be found that were able to eliminate this behavior. It therefore was not as effective as the PID system in this scenario.

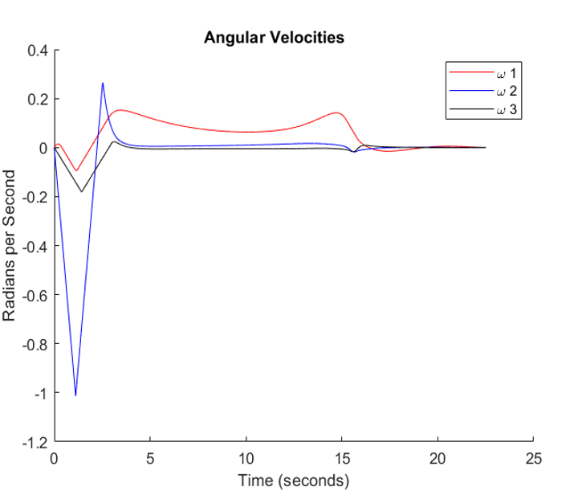
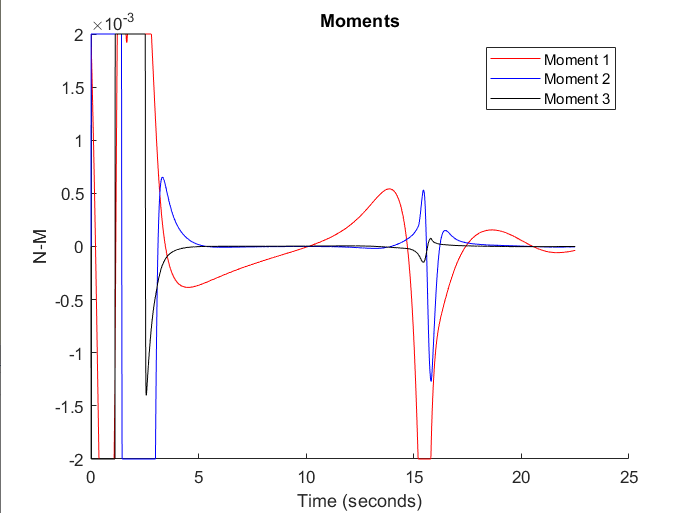


Figure , ELQR Attitude Maneuver (Angular Velocities)



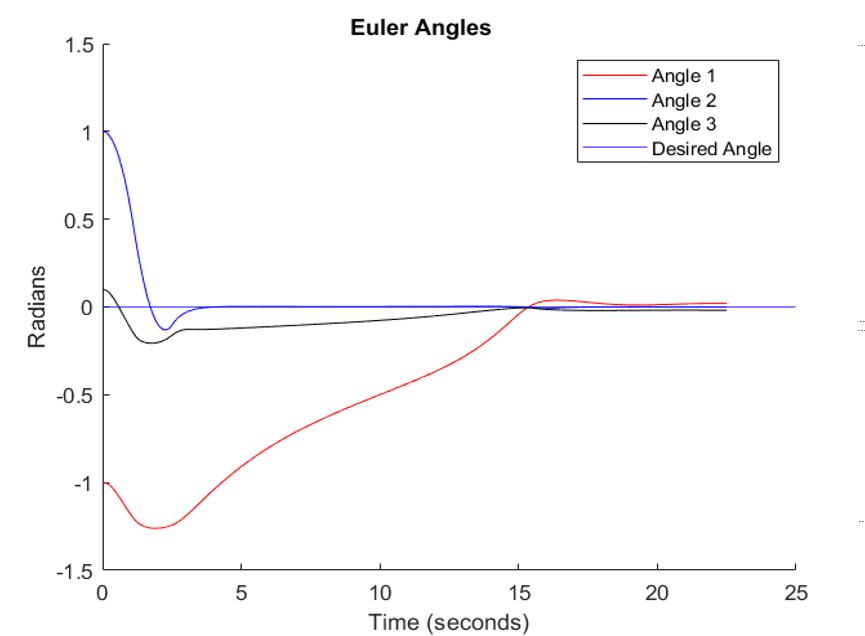


Figure , ELQR Attitude Maneuver (Angles) Figure , ELQR Attitude Maneuver (Control Moments)

### Sample Simulations for Velocity Errors

The same tumble as the PID example above was applied to the ELQR system (OSCaR has an initial angular velocity of 1.5 radians per second in each axis). Because the velocities could be controlled directly, the ELQR was able to stabilize the system more quickly than the PID controller. The angles do not converge to zero because the angular gains are set to very small values.

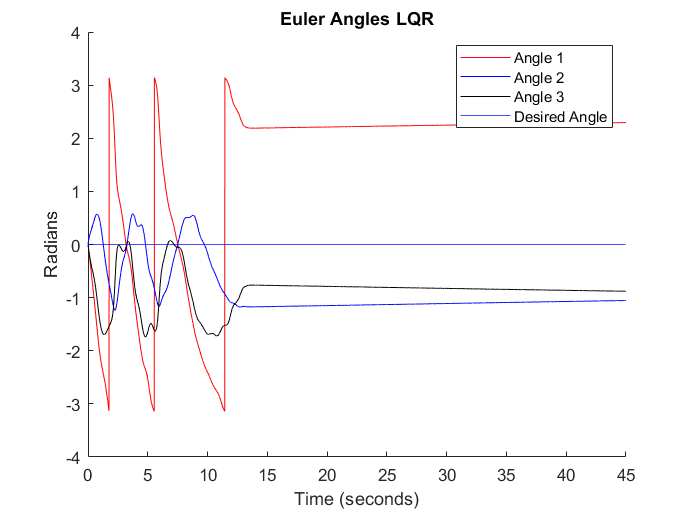


Figure , ELQR Stabilizing Tumble (Angles)

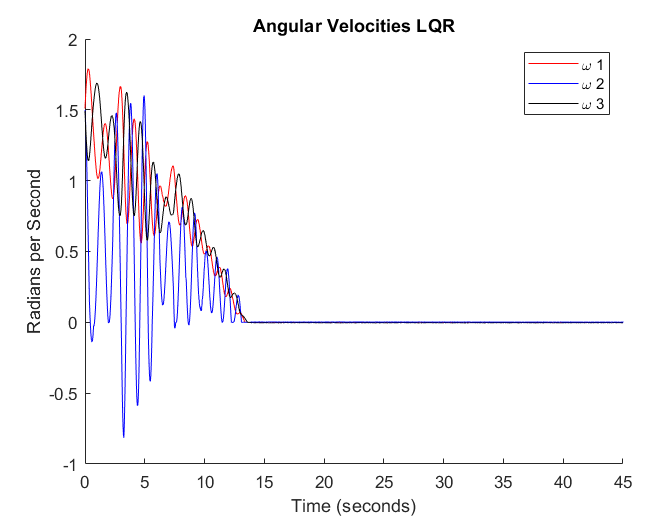


Figure , ELQR Stabilizing Tumble (Angular Velocities)

## Bang-Bang Control

One of the simplest ways to avoid the control issues stemming from the inertial terms of Euler’s equations is to only rotate the spacecraft in one axis at a time. This method is known as Bang-Bang Control, and by eliminating the inertial terms of Euler’s equations, it allows the control law to more directly control the system, albeit only one direction at a time. This results in behavior that is slower to reach the desired states, but generally exhibits less oscillation than controlling all three axes at a time. An example of the OSCaR system being controlled using Bang-Bang control can be seen below. PID control was used in this scenario.

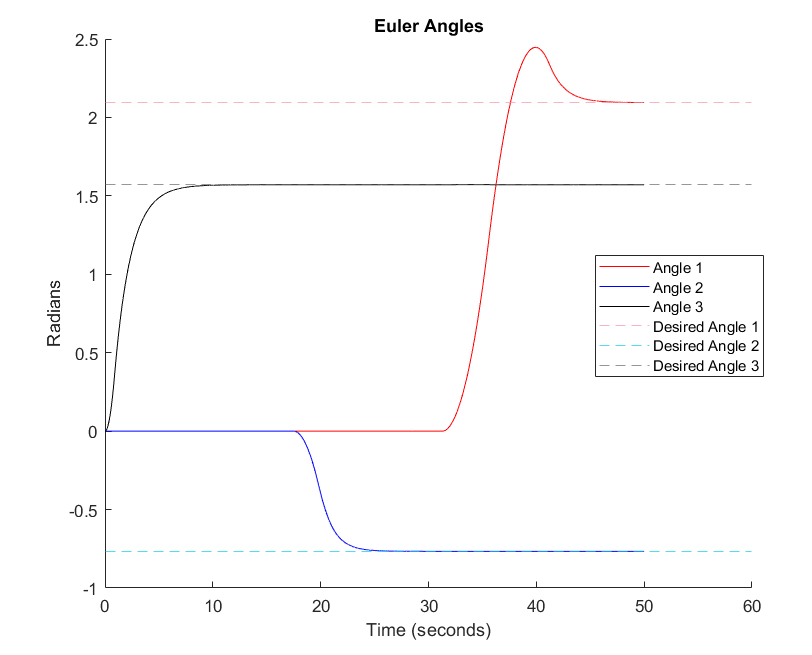


Figure , Euler Angles Resulting from Bang-Bang Control

## Effect of Step Size

A convergence study was performed to find an optimal step size. A step size that is too small will result in longer computational times without any increased performance, and a step size that is too large will result in the control moments updating too infrequently to properly control the craft. In a physical CubeSat, this refresh rate will be limited by the hardware available (including the onboard computer, reaction wheels and the Inertial Measurement Unit). This convergence study assumes that there are no hardware limits. The convergence study was performed on the same attitude change maneuver seen above. The ELQR control scheme was used, the simulation time was twenty seconds, and the control moment required for the maneuver was recorded. The following plot of control moment required was then produced:

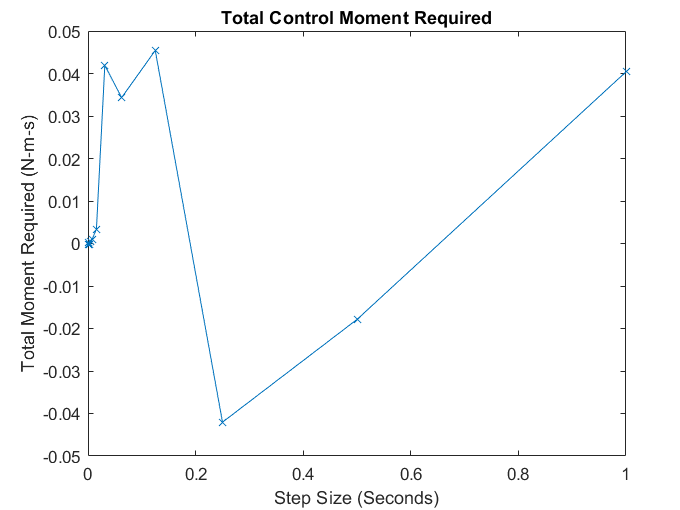


Figure , Total Control Moment required vs Step Size

From conservation of momentum, it is known that the total moment change applied by the reaction wheels must be zero, so long as the total change in angular velocity is zero (i.e., the craft begins and ends the simulation at rest). From zooming in on the smaller step sizes, we can see that a step size of approximately 0.01 seconds is required for the model to stabilize.

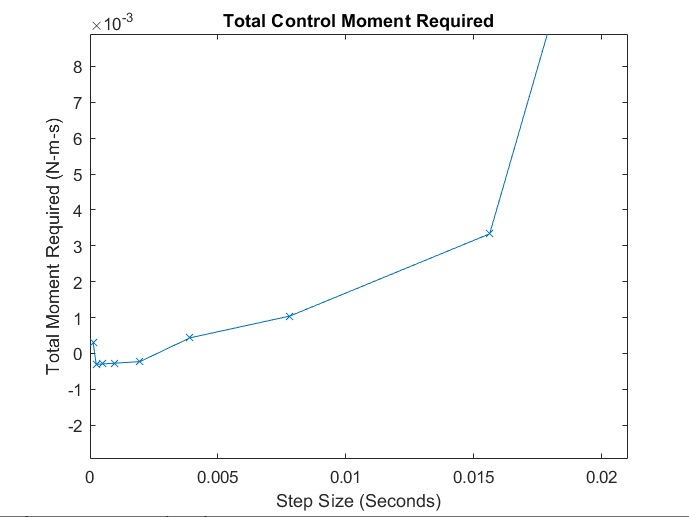
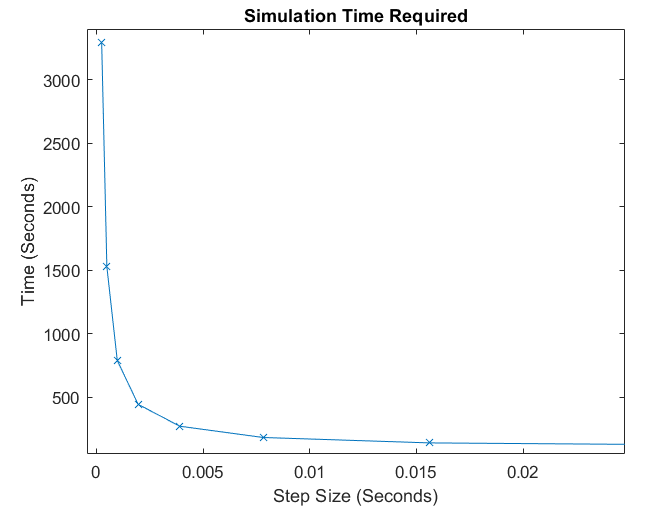


Figure , Total Control Moment required vs Step Size (Zoomed In)

Examination of the results of the results of the convergence study at each step size showed that at the larg step sizes, there was a complete failure to control the craft. At smaller step sizes (but not sufficiently small to complete control the system) the controller oscillated the control moment around zero, resulting in a jittering effect. This was usually biased on one side or another of the y-axis, resulting in a nonzero total moment. The nonzero moment observed even at very small step size is from a miniscule angular velocity at the end of the simulation.

As the step size decreases, the computational time increases exponentially, as seen below in Figure 16. It is therefore advantageous not to choose an incredibly small step size, as it will increase computational time without necessarily improving system behavior.



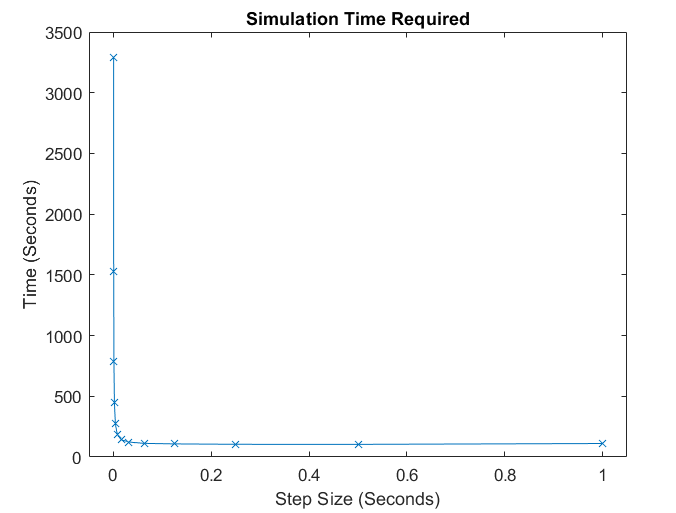


Figure , Computational Time Study Figure , Computational Time Study (Zoomed)

# **Conclusions**

After testing, it was shows that both PID and ELQR controllers are effective for a wide variety of states. It was also found that tuning the gains was very important to optimizing controller behavior. Thusly, no single set of gains is effective across all scenarios. For both controllers, if the angular velocities were sufficiently high, the inertial terms of Euler’s equations dominate the equations of motion and the controller is no longer able to control the system. The ELQR controller was slightly more effective at stabilizing tumbles, whereas the PID controller was more effective at attitude adjustment maneuvers.

## Future work

### Increasing Linearization accuracy

The linearized model involved numerically computing the angular accelerating via the forward difference method. Autolev is capable of doing these calculations via more accurate methods. Due to the significance of the linearized model, it would be pertinent to see if using these more accurate methods would improve the model.

### PID Controller Measuring Angular Velocity

Currently, the PID controller only measured the error of the Euler angles. This is disadvantageous when attempting to control a tumble, as angular velocity is more important than position. Future improvements to this control scheme could be a secondary PID system which measures and controls angular velocity. This controller would be switched to automatically if the angular velocities exceed a predetermined level. This would increase system robustness.

### Decreasing Computational Time via Variable Step Sizing

One of the limiting factors to testing this model is computational time available. The time that was required to run a simulation varied based on the velocity of the craft (as Autolev will dynamically adjust integration step size), but was generally 30-120 seconds of computational time for 60 seconds of simulation time[[1]](#footnote-1). As most simulations took approximately 20 seconds to complete, this was a significant roadblock, especially when executing convergence studies, which took many simulations to complete. If the angular accelerations and velocities are sufficiently low, a larger step size can be used without compromising simulation accuracy. This would decrease the computational time. Future work for this project could involve the creating of an algorithm to dynamically adjust step size based on the angular velocities and acceleration to decrease total computational time.

### Implementing Model-Based Predictive Control

Model based predictive control is a powerful control scheme which works via using an inverted model of the system to invert the equations of motion, and determining a control moment required to bring the system to the desired state. Generally, another control method (PID, Extended Kalman Filter, etc.) is applied to the error between the measured states and states expected by the model. This is a complex method of control, but generally one of the most effective control schemes. It has the advantage of being able to predict the effect of the inertial term of Euler’s equations. This makes it effective in situations where the inertial terms dominate, where other controllers would fail.

### Dynamically Optimize Gains

Currently, the gains are chosen by the user. This is not ideal, as in order for the system to reach the desired state most efficiently, the user must choose the correct gains, a process that may take several iterations. In the future, an algorithm should be developed such that the gains are chosen automatically from the difference between the initial states and the desired states.

It would also be useful to dynamically change these gains as the current states change. By changing the gains as the current states change, the system will be able to converge to the desired states more efficiently.

# **Citations**

1. (2010, December 12). Environmental protection of the geostationary-satellite orbit. Retrieved January 28, 2021, from Environmental protection of the geostationary-satellite orbit
2. FCC. (2020, April 2). Mitigation of Orbital Debris in the New Space Age. Retrieved from <https://docs.fcc.gov/public/attachments/DOC-363486A1.pdf>
3. Loff, S. (2015, July 22). CubeSats Overview. Retrieved January 28, 2021, from https://www.nasa.gov/mission\_pages/cubesats/overview
4. Michael et. Al (December 2017). OSCAR III Critical Design Report Design Team 3.
5. Wellsvar, T. (2019, April 23). Rensselaer Team Developing Tool to Battle Space Debris. Retrieved January 28, 2021, from https://news.rpi.edu/content/2019/04/23/rensselaer-team-developing-tool-battle-space-debris

# **Copy of AUTOLEV file**

## Agena-Gemini system

%% Autolev script for motion of Spacecraft (ideally 3U CubeSat OSCaR) using three Reaction Wheel

%% RW1, RW2, RW3 (associated with direction B1>, B2>, B3>, respectively) for attitude control.

%% In this initial treatment the control will be exercised as a reactive control moment applied

%% to each reaction wheel by the carrier B (for Body or Bus). perhaps in future versions we will

%% treat this as a controls problem where the angular velocity of each reaction wheel with

%% respect to the carrier B is controlled/prescribed by the control law.

%% For this analysis we will be using Reaction wheels aligned with orthogonal B directions B1>,B2>,B3>.

%% Each reaction wheel many be independently sized by adjusting its moment of inertial value about

%% is spin axis. Because of its symmetry and the fact that the RW centers of mass and spin axes

%% with respect to the carrier B, the masses, and moment of initial values are also considered as

%% part of the B. (Talk to me about this if you need clarification).

%

PAUSE 3

AUTOZ ON

CONSTANTS B1DIMENSION, B2DIMENSION, B3DIMENSION, RBO1, RBO2, RBO3

%% B1DIMENSION = The B1> dimension (width) of the three Unit (3U) OSCaR CubeSat.

%% Nominally this will be 0.10 [m].

%% B2DIMENSION = The B2> dimension (depth) of the three Unit (3U) OSCaR CubeSat.

%% Nominally this will be 0.10 [m].

%% B3DIMENSION = The B3> dimension (Length) of the three Unit (3U) OSCaR CubeSat.

%% Nominally this will be 0.30 [m].

%% RBO1 = The B1> dimension of position vector from Body B Geometric Center (BGC) to Body B CM (BO).

%% RBO2 = The B2> dimension of position vector from Body B Geometric Center (BGC) to Body B CM (BO).

%% RBO3 = The B3> dimension of position vector from Body B Geometric Center (BGC) to Body B CM (BO)

VARIABLES Q{3}',EP{4}', U{9}'

% Q1 = X (n1>) Location of Body B geometric Center ( ideal coincident with the Body B Center of Mass)

% Q2 = Y (N2>) Location of Body B geometric Center ( ideal coincident with the Body B Center of Mass

% Q3 = Z (N3>) Location of Body B geometric Center ( ideal coincident with the Body B Center of Mass

% ep{4} EULER PARAMETERs

BODIES B, RW1, RW2, RW3

%% B = CubeSat Body/Chassis/Carrier

%% RWj = j-th Reaction Wheel.

Newtonian N % Define Newtonian Frame to be "N"

POINTS O % Absolute Original

POINTS BGC % Body B Geometric Center (not generally co-incident with BO)

% Define names to be used for the respective masses of each body

% Mass Properties Calculations (Remember the mass of the RWs are already

% considered in the overall mass of OSCaR B)

MASS B=MASSB, RW1=0, RW2=0, RW3=0

INERTIA B, IB11, IB22, IB33, IB12, IB23, IB31

INERTIA RW1, IRW1, 0, 0, 0, 0, 0

INERTIA RW2, 0, IRW2, 0, 0, 0, 0

INERTIA RW3, 0, 0, IRW3, 0, 0, 0

% Important Position Vectors

P\_O\_BO> = Q1\*N1> + Q2\*N2> + Q3\*N3> % Position of Body/Bus/Carrier CM for analysis and Animake

P\_BGC\_BO> = RBO1\*B1> + RBO2\*B2> + RBO3\*B3> % position vector of Body B CM (BO) relative to

% Body B Geometric Center BGC

% Orientation relations (use Euler Parameters for detemining and tracking orientation)

DIRCOS(N,B,EULER,EP1,EP2,EP3,EP4)

SIMPROT(B,RW1,1,0) % RW1 rotates relative to B about direction B1>. Angle is set to 0 because

% RW1 symmetry about B1> axis, and the rotation angle does not need to be

% tracked in this applicaion.(this also simplifies and accelerates program)

SIMPROT(B,RW2,2,0) % RW2 rotates relative to B about direction B2>. Angle is set to 0 because

% RW2 symmetry about B2> axis, and the rotation angle does not need to be

% tracked in this applicaion.(this also simplifies and accelerates program)

SIMPROT(B,RW3,3,0) % RW3 rotates relative to B about direction B1>. Angle is set to 0 because

% RW3 symmetry about B3> axis, and the rotation angle does not need to be

% tracked in this applicaion.(this also simplifies and accelerates program)

ANIMATE(N,O,B) % generate data for animake animation

% Angular velocities

W\_B\_N> = U4\*B1> + U5\*B2> + U6\*B3>

W\_RW1\_B> = U7\*B1>

W\_RW2\_B> = U8\*B2>

W\_RW3\_B> = U9\*B3>

% Velocities

V\_O\_N> = 0> % Point O is inertially fixed

V\_BO\_N> = U1\*N1> + U2\*N2> + U3\*N3> % Velocity of CM of the body/Chassis/carrier B

V\_RW1O\_N> = V\_BO\_N> % these commands are provided to make AutoLev happy but will

% not ultimately contribute because MRW1 = 0

V\_RW2O\_N> = V\_BO\_N>

V\_RW3O\_N> = V\_BO\_N>

% Kinematical Differential equations

Q1'=U1

Q2'=U2

Q3'=U3

% Tell AutoLev that the three Control Torques will be specified quantities

CONSTANTS CTORQUE1, CTORQUE2, CTORQUE3

%Apply Control moments to Reaction Wheels by Carrier

TORQUE(B/RW1,CTORQUE1\*B1>) % Control Torque applied to RW1

TORQUE(B/RW2,CTORQUE2\*B2>) % Control Torque applied to RW2

TORQUE(B/RW3,CTORQUE3\*B3>) % Control Torque applied to RW3

KINDIFFS(N,B,EULER,EP1,EP2,EP3,EP4)

ZERO = FR() + FRSTAR()

KANE()

%UNITSYSTEM KG,METER,SECOND

UNITS [B1DIMENSION, B2DIMENSION, B3DIMENSION, RBO1, RBO2, RBO3]= METRES, [MASSB]= KG, [U1, U2, U3] = m/s, [U4,U5,U6,U7,U8,U9]=Rad/s

OUTPUT T, Q1, Q2, Q3

OUTPUT T, U1, U2, U3

OUTPUT T, U4, U5, U6

OUTPUT T, U7, U8, U9

CODE DYNAMICS() OSCAR001.C,SUBS

% Important Position Vectors

P\_O\_BO> = Q1\*N1> + Q2\*N2> + Q3\*N3> % Position of Body/Bus/Carrier CM for analysis and Animake

P\_BGC\_BO> = RBO1\*B1> + RBO2\*B2> + RBO3\*B3> % position vector of Body B CM (BO) relative to

% Body B Geometric Center BGC

% Orientation relations (use Euler Parameters for detemining and tracking orientation)

DIRCOS(N,B,EULER,EP1,EP2,EP3,EP4)

SIMPROT(B,RW1,1,0) % RW1 rotates relative to B about direction B1>. Angle is set to 0 because

% RW1 symmetry about B1> axis, and the rotation angle does not need to be

% tracked in this applicaion.(this also simplifies and accelerates program)

SIMPROT(B,RW2,2,0) % RW2 rotates relative to B about direction B2>. Angle is set to 0 because

% RW2 symmetry about B2> axis, and the rotation angle does not need to be

% tracked in this applicaion.(this also simplifies and accelerates program)

SIMPROT(B,RW3,3,0) % RW3 rotates relative to B about direction B1>. Angle is set to 0 because

% RW3 symmetry about B3> axis, and the rotation angle does not need to be

% tracked in this applicaion.(this also simplifies and accelerates program)

ANIMATE(N,O,B) % generate data for animake animation

% Angular velocities

W\_B\_N> = U4\*B1> + U5\*B2> + U6\*B3>

W\_RW1\_B> = U7\*B1>

W\_RW2\_B> = U8\*B2>

W\_RW3\_B> = U9\*B3>

% Velocities

V\_O\_N> = 0> % Point O is inertially fixed

V\_BO\_N> = U1\*N1> + U2\*N2> + U3\*N3> % Velocity of CM of the body/Chassis/carrier B

V\_RW1O\_N> = V\_BO\_N> % these commands are provided to make AutoLev happy but will

% not ultimately contribute because MRW1 = 0

V\_RW2O\_N> = V\_BO\_N>

V\_RW3O\_N> = V\_BO\_N>

% Kinematical Differential equations

Q1'=U1

Q2'=U2

Q3'=U3

% Tell AutoLev that the three Control Torques will be specified quantities

CONSTANTS CTORQUE1, CTORQUE2, CTORQUE3

%Apply Control moments to Reaction Wheels by Carrier

TORQUE(B/RW1,CTORQUE1\*B1>) % Control Torque applied to RW1

TORQUE(B/RW2,CTORQUE2\*B2>) % Control Torque applied to RW2

TORQUE(B/RW3,CTORQUE3\*B3>) % Control Torque applied to RW3

KINDIFFS(N,B,EULER,EP1,EP2,EP3,EP4)

ZERO = FR() + FRSTAR()

KANE()

%UNITSYSTEM KG,METER,SECOND

UNITS [B1DIMENSION, B2DIMENSION, B3DIMENSION, RBO1, RBO2, RBO3]= METRES, [MASSB]= KG, [U1, U2, U3] = m/s, [U4,U5,U6,U7,U8,U9]=Rad/s

OUTPUT T, Q1, Q2, Q3

OUTPUT T, U1, U2, U3

OUTPUT T, U4, U5, U6

OUTPUT T, U7, U8, U9

CODE DYNAMICS() OSCAR001.C,SUBS

## 

# **Autolev, C and MATLAB tutorial**

# Overview

Autolev is a powerful tool for the computation of multibody dynamics for systems that are otherwise too complex to be solved for simply by hand, or by writing code for from the ground up. Autolev, though powerful, lacks some of the built in features that MATLAB offers. As these can be very useful to modeling systems, it would be advantageous for users to be able to merge these two software packages. Currently, Autolev does not offer MATLAB support as a built in capability. This guide is meant to help a user connect Autolev code to MATLAB via the use of MEX functions. This will work for most C codes, but the examples used in this guide will be specifically for Autolev auto generated code.

# Requirements

The following requirements are needed by the user in to successfully connect Autolev Code to MATLAB.

* Autolev, and working Autolev code
* MATLAB (This guide was developed for 2020a, 2020b and 2017b. Other versions may have additional undiscovered issues)
  + A MATLAB C compiler. MinGW64 works well (install via MATLAB’s add on explorer)
* Knowledge of how to code in MATLAB. Knowledge of SIMULINK can also be helpful
* Basic Knowledge of C or programming languages similar to C (Object oriented languages like Java, or even Python will be enough)

# Setup

## Run your Autolev code

Once the user has a complete Autolev code, it is advisable to test as much as possible now. Mistakes and errors in the base code will be far easier to debug when the code is in its most basic form, before the other layers of cross-platform complexity are applied. If you are running the auto-generated Autolev code with a language that is not C (Such as Fortran), apply the following command at the end of your code:



To code the dynamics in C, use a C compiler (such as Codeblocks IDE) to ensure that this still works as expected. Save a copy of your unmodified C code.

## Modify Autolev Code for MEX conversion

All quantities that will be controlled/changed by MATLAB should be declared as a CONSTANT in Autolev. This includes control torques, forces applied, etc. This will allow MATLAB to feed the changed value into Autolev. Re-test all of your C code. Ensure that the code functions as expected with constant inputs.

# Creating a MEX Function from Auto Generated C Code

MATLAB support pages for additional help with MEX:

<https://www.mathworks.com/help/matlab/call-mex-file-functions.html>

<https://www.mathworks.com/help/matlab/ref/mex.html>

## Overview:

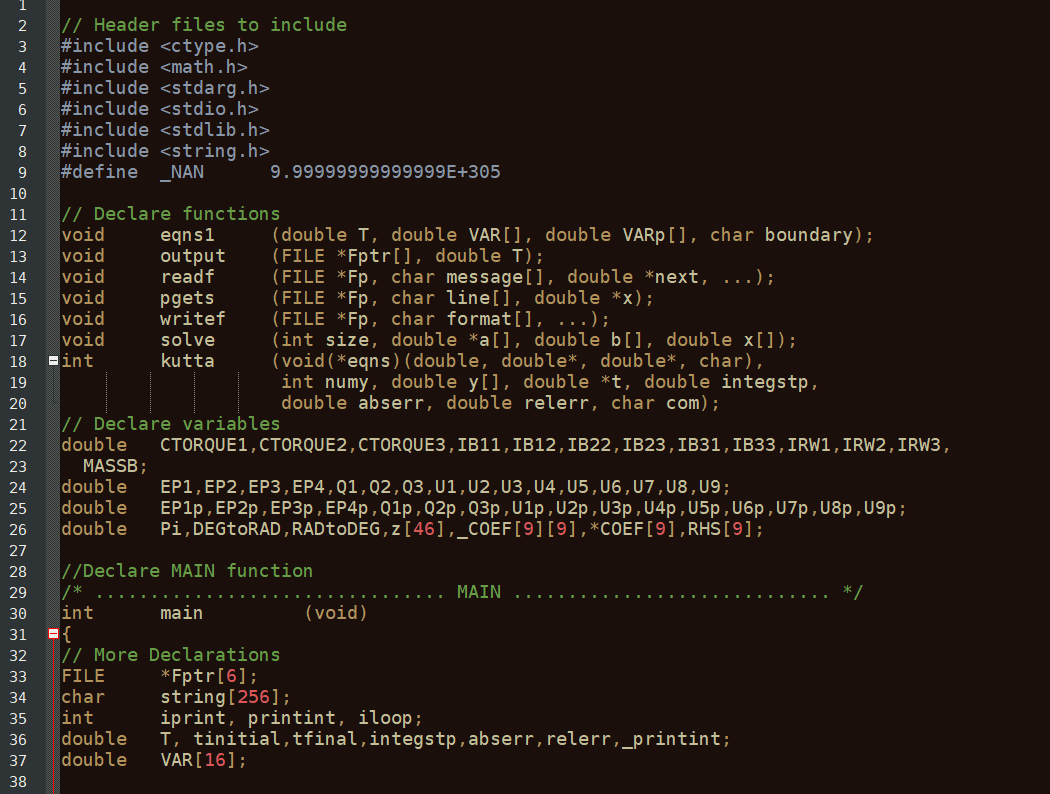
MEX stands for MATLAB Executable Function. This is a function written in slightly modified C that can be run in MATLAB just like a regular MATLAB function. It can take inputs, deliver outputs, write to the console, etc. The auto generated C code that Autolev returns can be simply modified into a MATLAB executable function, which will allow the code to be run in MATLAB, integrate into SIMULINK, etc.

A C code cannot be run as a MEX function without some modifications. Many of the commands generated by Autolev will cause errors or erroneous things occur. Due to the cross-platform nature of these two programs, there are often no error messages, or the messages provided are not clear. This guide is meant to help overcome some of these hurdles.

Before you begin to modify your own code, it can be advantageous to complete a simple example. A good place to start is to complete this tutorial by Shawn Lankton: <http://www.shawnlankton.com/2008/03/getting-started-with-mex-a-short-tutorial/>

## Modification:

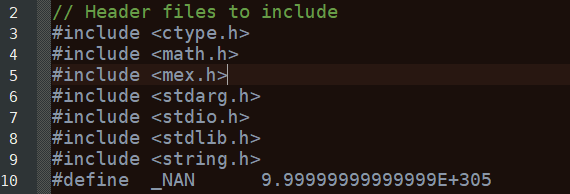
When Autolev generates code, the initial C code will look something like this:



Most of the modifications that are necessary will occur in the first 100 or so lines of code.

### Headers

The first thing to do is to include the MEX header function in the header declarations. This is what allows MATLAB to run all of the MEX specific commands (such as printing to the console, reading in variables, etc.).



### Main Function Declaration

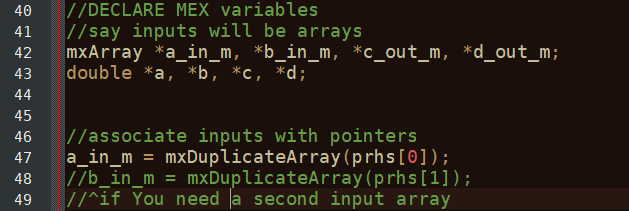
Next, the MAIN function declaration is modified. Instead of being a MAIN function, this is modified to be a mexFunction. The signature of the MEX function is always the same.



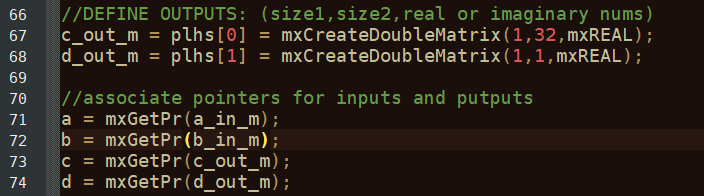
The mxArray plhr and prhs represent **P**ointers to the **L**eft **H**and **S**ide and **P**ointers to the **R**ight **H**and **S**ide. The Left hand side are outputs, and the right hand size are inputs

### Declaring Inputs and Outputs

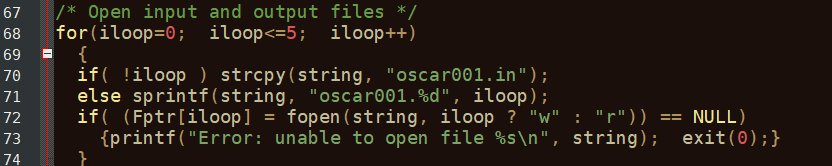
Now declare pointers for inputs



Then declare the outputs:

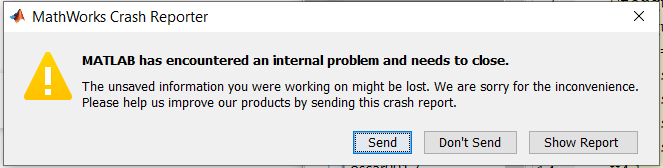


Now, you have declared the inputs and outputs. Locate where the C code opens the input and output files. Since the inputs and outputs are no longer files, this can be removed/commented out. In this example, this looks like this:



### Notes on printf and exit(0);

This is a good time to note some of the differences between MEX and C. In C, the command exit(0) will close the running C code, and the printf error message will be left in the open window, allowing the user to read it. In MEX, the command exit(0) will attempt to close the program it is running in (which is MATLAB). MATLAB is not designed to be closed in such a way. In earlier versions of MATLAB this will cause MATLAB to crash completely, while in later versions (at the time of writing, this is 2020b) it will give an error that looks like below, but will not crash.

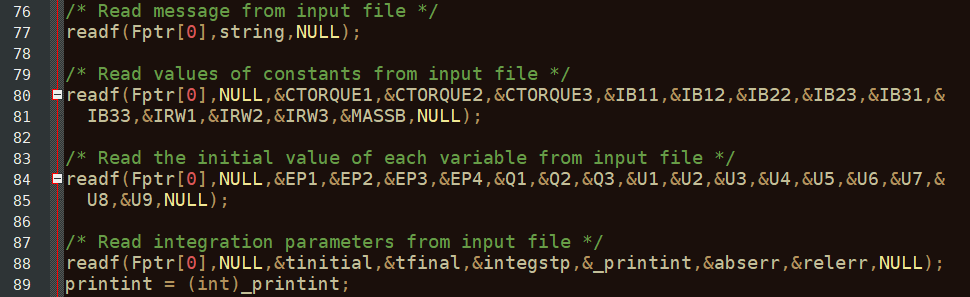


As such, all instances of exit(0) **MUST** be removed. When doing this, replace the print command with: mexPrintf(“error message”);

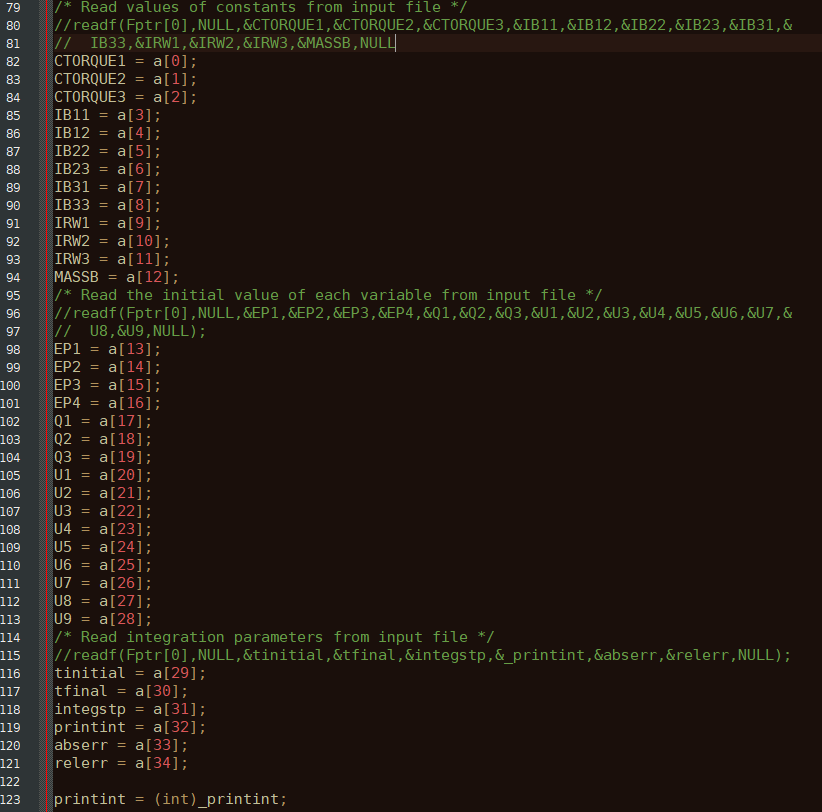
This prints to the MATLAB console, where it can easily be read by the user. Do this for all instances where the message should be printed to the console.

### Replacing inputs

Now, locate where the C code reads the input file. In this example, this looks like:

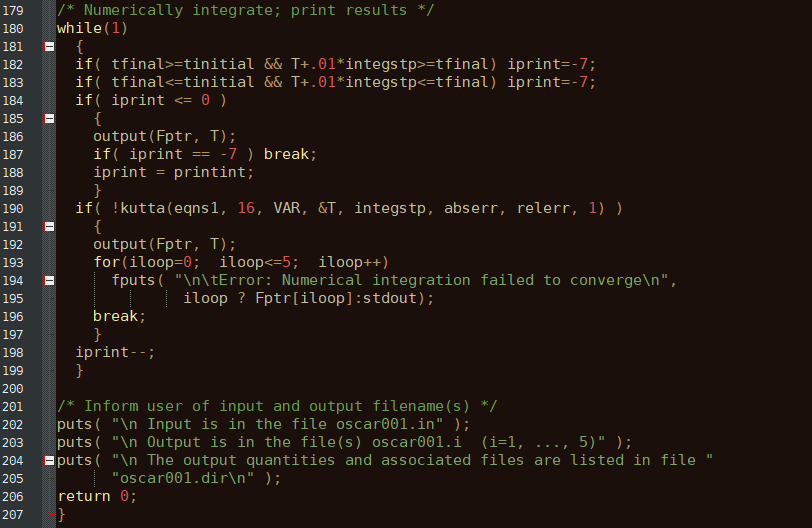


These are then replaced with the inputs that we have declared earlier. In this example, the readf functions are removed/commented out, as their inclusion will only slow down the function.

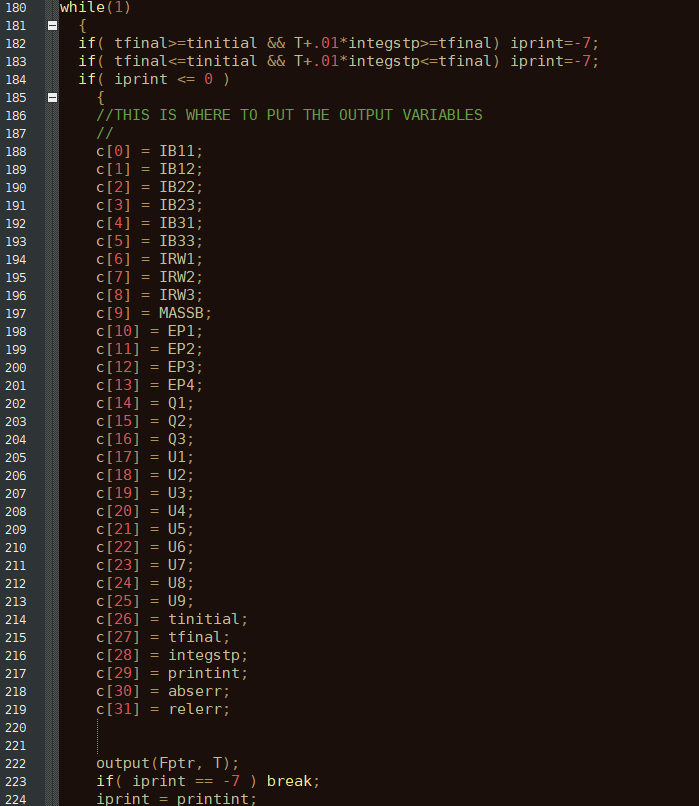


### Replacing Outputs

Next, the outputs will be modified so they report to MATLAB rather than printing to an output file. There will be a section that has the comment “print result”. In this example, this looks as follows:



Add the code below before the output command. This assigns values to the output variables. Note: you will have to select the variable names based on what you want your specific code to output.



Finally, remove the “Return 0;” command from the last line of the function, as the mexFunction does not return anything.



### Final Words on Converting from C to MEX

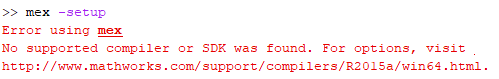
MATLAB will not run some C commands properly. As mentioned earlier, one of these is exit(0). Another is the print function. The standard printf function must be replaced with the MEX command mexPrintf, which will print to the MATLAB console everywhere that the program wishes to print something for the user. These two are the more common and troublesome in auto generated C code from Autolev, but it is not an exhaustive list. Check the MATLAB MEX documentation for a complete list.

# Using MATLAB to Run MEX Code

## MATLAB Setup

To get started, MATLAB must be setup to run MEX code. This is done first by running MATLAB’s automatic setup via the command:

mex –setup

If you do not have a C compiler, you will get the error: 

If this occurs, install one. The recommended one is MinGW64 Compiler (C), which is available via the MATLAB add-on installer.



Next, with the modified C code in your current folder, use the command:

mex Filename.c

This creates the MEX function from the code you have modified. This will give errors if you have made any mistakes. Fix these according to the error messages.

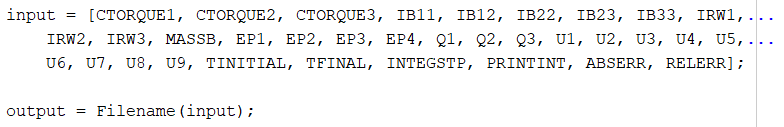
If the code is successful, you will see



You should see a new file appear in your current folder with the file extension “.mexw64”. This is the actual MEX file MATLAB will run.

## Simple Example

You can now run a simple instance of your code.

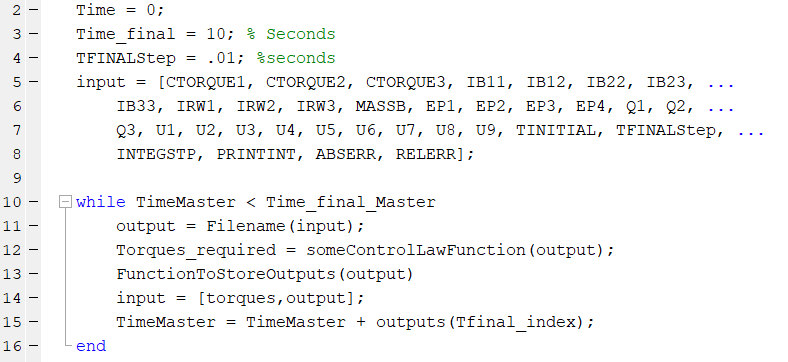


If all goes well, the variable output will contain all of the outputs in the form of an array, and should be the same as if you ran your original C code. Check the two against each other. Do they match? If they do not, check the MEX modification that you have written and see if you have made an error.

## Basic Implementation Notes

Once fully configured the MEX function can now be run just like any other MATLAB function. One example of how this could be advantageous is that one can now use built-in MATLAB functions that would otherwise be difficult to write in C, such as linear quadratic regulators, numerical integrators and differentiators, etc.

Below is an example of *pseudocode* running the code used above in a simple example of a control loop, where the control law is defined by the function someControlLawFunction.



After setting up the initial inputs, as well as the initial and final times, the code is run as normal, except the input to the MEX function has a Tfinal as something quite small, on the magnitude of hundredths of a second. The control law is calculated, the outputs are stored, and the outputs of the MEX function and the calculated control torques are set as the inputs for the next iteration of the MEX function, continuing until the desired end time is reached. The time step must be small, as the inputs will remain constant over it.

This is reasonably representative of most systems, as the input thruster, momentum wheel, etc. can only updated so fast. It also allows for convenient execution of the script. As the MEX function will be run many times, it can be advantageous to strip it down to its bare components, and remove unnecessary function calls (such as opening and closing files, writing to outputs as well as to the MEX outputs, etc.) in order to optimize the amount of computational time this will require.

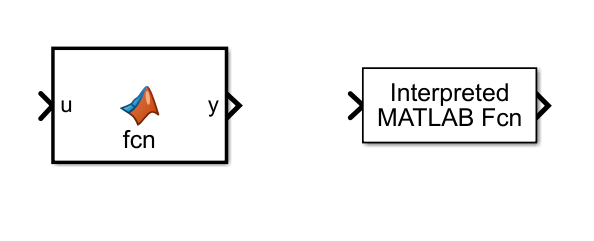
## Simulink

### Introduction

One of the more powerful MATLAB add-ons is Simulink, which allows users to create diagrams of code, and is taught in many RPI classes as the tool to use for controlling systems. SIMULINK is a diagram-like program, where variables traverse from one block to another.

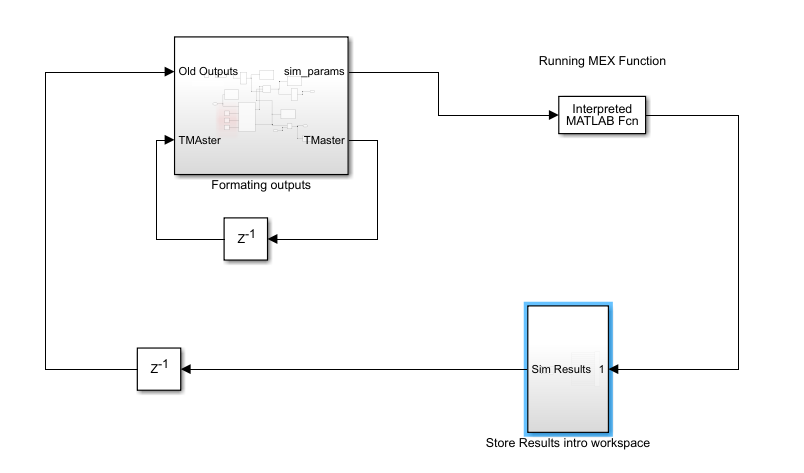
### MEX and MATLAB Interpreted Functions

In order to implement MEX files in SIMULINK, a minor modification must be made. The MEX function cannot be run in a standard MATLAB function block, as it is not supported in current versions of SIMULINK (although this may change in later versions). Instead am interpreted MATLAB function block must be used. They function quite similarly, except the Interpreted MATLAB function block runs code that is saved in a .m file specified, rather than a file saved in the Simulink simulation. Additionally, only one output/input can be made (although these can be arrays/matrices).

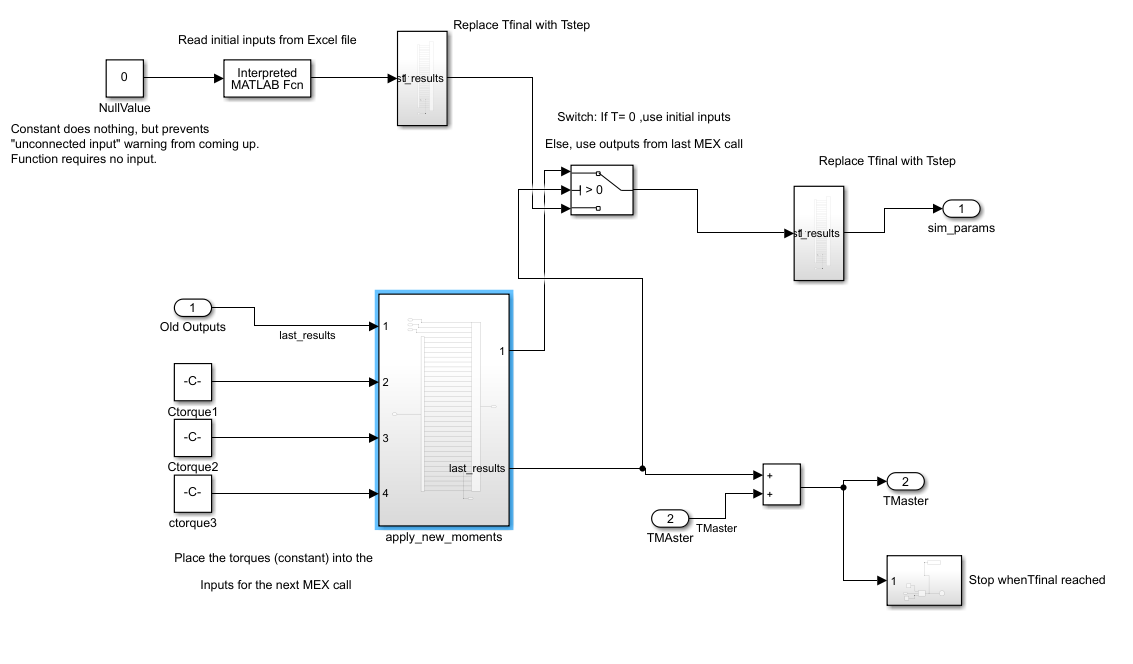


### Basic Example

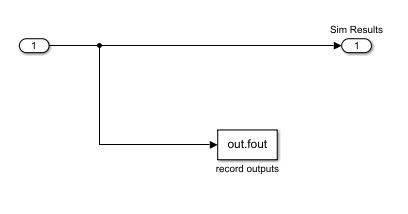
A basic example of running a MEX function in Simulink without a control law can be seen below:



Inside the Formatting outputs subsystem is the initial setup and formatting the outputs from the MEX function into the inputs for the next iteration.



The Store Results subsystem simply stores the results to the workspace for plotting later:



This SIMULINK diagram will work the same as the sample MATLAB function above. A control law could be implemented to vary the control torques.

# Conclusion

Both Autolev and MATLAB are powerful tools for computation. By mastering the ability to combine the two, they can both be used for their respective strengths, and create more complex system with less effort by the user. Once one has done several conversions, they are quick to implement, and can provide a simple way of running complex multibody code in MATLAB.

1. This was on a Dell XPS 15, with 16 Gb of ram and a 2.5-GHz Intel Core i7. More powerful computers will reduce computational time. [↑](#footnote-ref-1)