Final Report on Self-Balancing Bot Project

J. Fred, F. Röben, B. Tubbs
April 24, 2018

Motivation

We chose a self-balancing mobile robot similar to a segway as our project to implement a MPC. This is a type of inverted pendulum which is a classic problem in dynamics and control theory used as a benchmark for testing control strategies. It is a fourth-order nonlinear unstable system and will fall over without an active controller. The main problem is driving it at both a specific location and balancing it upright when it arrives.[1]

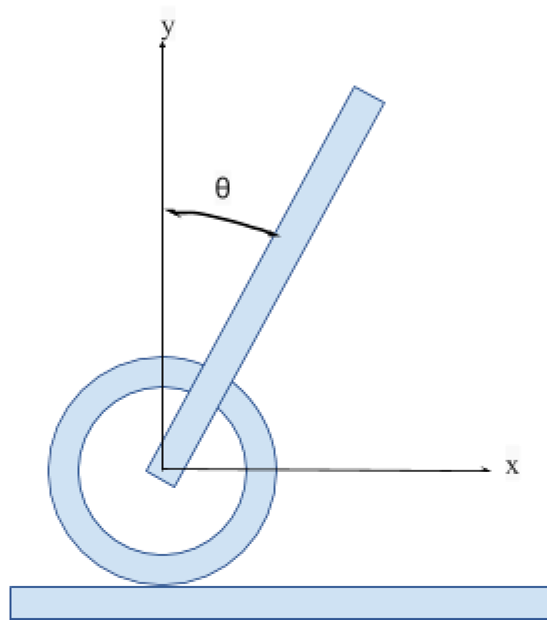
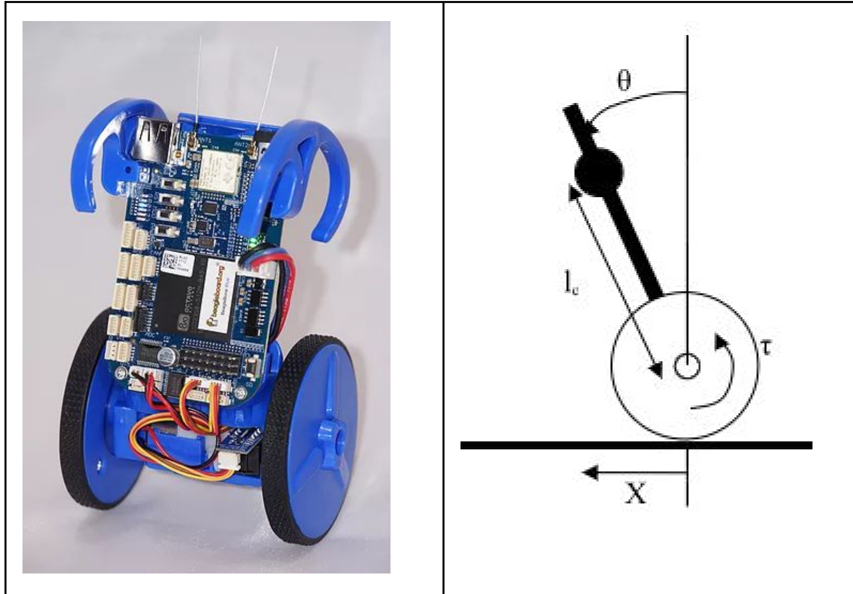


Figure 1.

There is only one variable to manipulate, which is the motor torque of the robot, but these two criteria to fulfill. We will give a short description of the objectives below:

Objective

Control a self-balancing 2-wheel mobile robot similar to a Segway human transporter vehicle.
The self-balancing 2-wheel bot is a classic inverted-pendulum control problem.



BeagleBone Mobile Robot

Controlling the inverted pendulum can be considered as two distinct control tasks:

1. Manipulate τ to minimize θ
2. Manipulate τ to control x to a setpoint

The two robots we have purchased are somewhat capable of the first task - maintaining the robot angle in the vertical with simple PID control algorithms already provided by the vendor. But the algorithms are still somewhat unstable and susceptible to disturbances. More importantly, they are not able to control the x-position of the robot, so they randomly 'wander' off to the left or right and have no way of maintaining a fixed position.

The ultimate objective is to control both robot angle and x-position simultaneously. This is not as simple as it might sound.

The difficulty lies in the two control tasks to be conflicting in the near future (in view of a controller). For the system to be able to move from position A to B it first needs to get into an unstable forward leaning position and then start the motors to drive forward. This will move the robot and keeps it from tipping over. But even just to get this first initial leaning towards the direction of the new location it needs to drive a short distance backwards for the robot to tip in the right direction, which again is conflicting with a short-term mpc trying to balance or trying to drive directly to point B. While driving a control trying to balance the system would stop the

forwards movement and therefore stopping the robot. As you can see just the targets of balancing the robot and reaching, a desired set point poses conflicting criteria in the short term. So why not use a long control horizon, which can see the optimal trajectory from A to B in the full time horizon with a very detailed resolution (to be able to balance the unstable system)? Such an approach, as we will show works really well, if it can be calculated beforehand, but will fail in balancing the real robot, because the calculation time will be too high to react on the robot tipping over.

As can be seen this system is a highly unstable system, which needs both a fast control and a long foresight. We will try to find a way to combine ideas and MPCs to find a solution, which is in its origin based on an MPC adaption.

Useful links: Groups sheet, Project proposal homework page, hardware platforms considered.

Control System Design

Our current design for the controller on the robot is depicted in the diagram below.

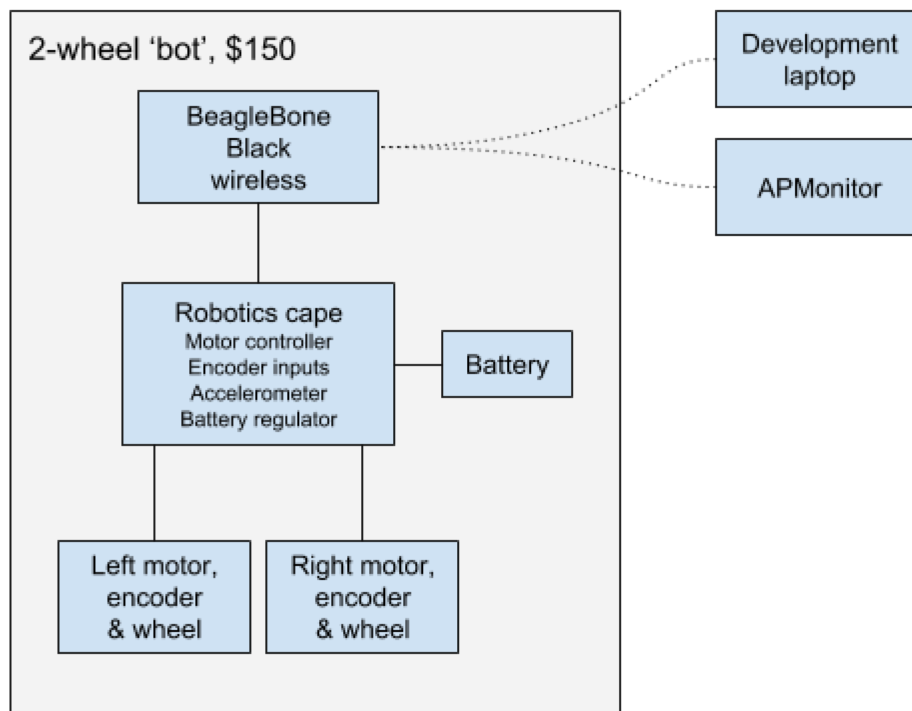


Diagram of Proposed Control System

Several robotics hardware platforms were considered - See Sheet of options.

Criteria for hardware platform selected:

- Fairly inexpensive so that all team members can purchase their own.

- Arduino is mediocre but prefer something better (newer, faster, runs Python) such as a BeagleBone or an Arduino replacement like a STM Nucleo board.
- Prefer hardware that handles encoders in hardware rather than ISRs.
- Off-the-shelf platform so that time and focus isn't diverted to mechanics, battery power, wireless communications etc.

Literature Review

A quick survey identified many research articles about control of mobile inverted pendulums.

[1] Wikipedia article with equations. Also mentions Kapitza's pendulum: https://en.wikipedia.org/wiki/Inverted_pendulum.

[2] Stabilization fuzzy control of inverted pendulum systems
J. Yi*, N. Yubazaki
http://neuron.tuke.sk/vascak/predmety/FSR/Eseje/Seliga%20-%20Stabilization%20fuzzy%20control%20of%20inverted%20pendulum%20systems.pdf

[3] The dynamics of a Mobile Inverted Pendulum (MIP)
Saam Ostovari, Nick Morozovsky, Thomas Bewley UCSD Coordinated Robotics Lab.
Has section 'Kinematics', 2.2, that lists equations and their derivation with Free Body Diagrams for the wheel and robot body. http://renaissance.ucsd.edu/courses/mae143c/MIPdynamics.pdf

[4] Approximate nonlinear model predictive control with in situ adaptive tabulation
John D. Hedengren, Thomas F. Edgar.
Computers and Chemical Engineering 32 (2008) 706–714

Variables and Constants

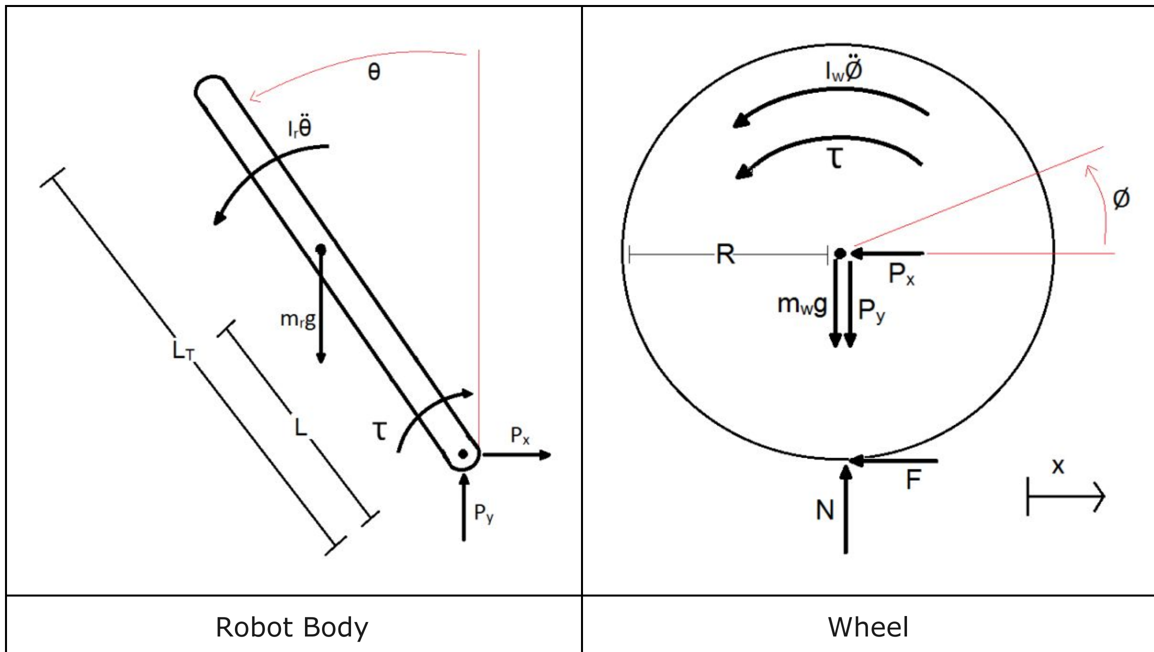
The table below lists the factors that influence the dynamic outcome. These are categorized into factors that cannot change (constants / parameters), factors that can change (degrees of freedom or manipulated variables)

	Description	**Type**
m_r	Mass of robot body	Constant
L	Length of robot body (center of gravity to wheel axis)	Configurable
I_r	Moment of inertia of robot body	Configurable

θ_r	Angle of inclination with respect to gravity	Measured variable
m_w	Mass of wheel	Constant
R	Radius of wheel	Constant
I_w	Moment of inertia of wheel	Constant
τ	Motor torque	Intermediate
F	Drive force of wheel	Intermediate
θ_w	Wheel angle	Control variable
x	Horizontal position, velocity, and acceleration	Intermediate
g	Acceleration of gravity	Constant

Equations

The following equations describe the dynamic response of the system. In this case they are equations of motion based on conservation of momentum. The final equations of motion and free-body diagrams were copied from reference [3].



Robot Body | Wheel

$$(I_w + (m_w + m_r)R^2)\ddot{\theta}_w + (m_r RL\cos\theta)\ddot{\theta}_r - m_r RL\sin\theta_r\,\dot{\theta}_r^{\,2} = \tau$$

$$(m_r RL\cos\theta_r)\ddot{\theta}_w + (I_r + m_r L^2)\ddot{\theta}_r - m_r gL\sin\theta_r = -\tau$$

Objective Function

To control the robot we will minimize a cost function over a moving horizon. The cost function will be dependent on the two main control variables, θ_r and x. Note that we assume no skidding of the wheel, so in our model the following relationship holds.

$$x = R\theta_w$$

We have the option of adding other state variables to the cost function if required. For example, the horizontal and angular velocities ($\dot{x}$, $\dot{\theta}_r$).

Constraints and Challenges

- The range of possible torques that the motors can produce will be limited (lower and upper bounds)
- We expect scan-time or processing power to be a major challenge so limiting the complexity of the objective function will be a priority (e.g. number of timesteps)
- When it comes to implementation, the differences between the dynamics of the actual robots and the model will be critical. For example:
  - Friction (between robot and wheels, and between wheels and ground)
  - ‘Gyro-scopic’ effects of the motors spinning at high speed
  - Air resistance
- Considerable tuning or model estimation tasks may be needed.
- Since there is no actual torque measurement or control we will have to find a way to map motor power input signal to desired torque (this is likely to be a nonlinear relationship).

Uncertainties

- Calibration of ‘inclinometer’ - balance_config.h file has setting to zero-out inclination.
- Communications speed to APMonitor.

Approach

We adopted the following five-step approach.

1. Solve with Gekko one-shot

2. Simulate system with ODEs
3. Test PID controller
4. Test & tune MPC controller
5. Test MPC on robot

Steps 1 to 4 are complete but step 5 has not yet been started.

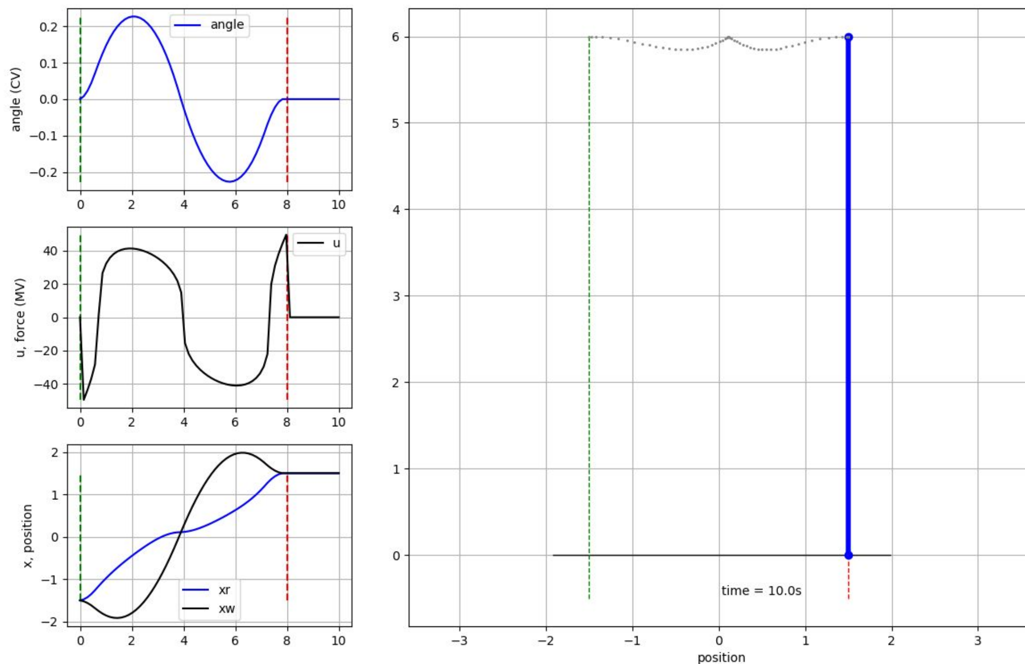
Timeline for the project and the anticipated final product for this project.

- Checkpoints
    - Compile and run rc_balance.c example from github.com/StrawsonDesign/Robotics_Cape_Installer (done)
    - Configure rc_balance.c example by editing balance_config.h (in progress)
    - Measure and model physical parameters
    - Implement model-based control
    - Implement return to x=0.
    - Implement auto-zero for inclinometer.

Results

1. Full horizon solution (long foresight) with MPC

First we built a GEKKO model from the equations described in the previous chapter. We defined the differentials and used arbitrary values for the robot parameters (at this stage we did not attempt to simulate real BeagleBone robot). The model optimizing the trajectory of the model had a fixed time at which the robot had to reach the target location with a velocity and acceleration of zero in a perfect upright position. As can be seen from the shown result, the GEKKO model was able to calculate such a trajectory.



In the upper left corner the angle of the robot is shown to be zero at the specified x target of 0 after 8 seconds, which was the given time to reach the final location. The torque of the motor and the position are at a value of zero right at the final point, which also translates into no further velocity or acceleration after it. However even with this calculation being fast in terms of large dynamic systems, it is to slow to balance a robot. Therefore, we moved on to write a MPC for balancing the robot, which was faster and focused just on balancing the robot (bringing the angle theta to zero).

2. Simulation with scipy.integrate.odeint

For subsequent simulations we used the odeint function from the `scipy.integrate` Python library. This was used to simulate the robot dynamics so that we could test different control algorithms applied to the robot. Two differential equations were implemented, and `numpy.linalg.solve` was used to solve for the two unknowns, the angular acceleration for the robot and the wheel. A motor torque, u, was the input to `odeint` from the control algorithms. The Python code for this section of the algorithm is shown below.

Code sample 1.

```
import numpy as np

def modelDerivFunc(xz, tz, u):

    [θr, θr_dot, θw, θw_dot] = xz   # unpack inputs.

    A = np.array([
                  [Mr*R*L*np.cos(θr), (Iw + (Mw + Mr)*R**2)],
                  [(Ir + Mr*L**2)   ,  Mr*R*L*np.cos(θr)]
                  ])

    B = np.array([
                  [Mr*R*L*θr_dot**2*np.sin(θr) + u - θr_dot*Fr],
                  [Mr*G*L*np.sin(θr)      - u - θw_dot*Fw]
                  ])

    # Solve for X where A.X=B
    X = np.linalg.solve(A, B)
    θr_ddot, θw_ddot = X   # un-pack X

    return [θr_dot, θr_ddot, θw_dot, θw_ddot]
```

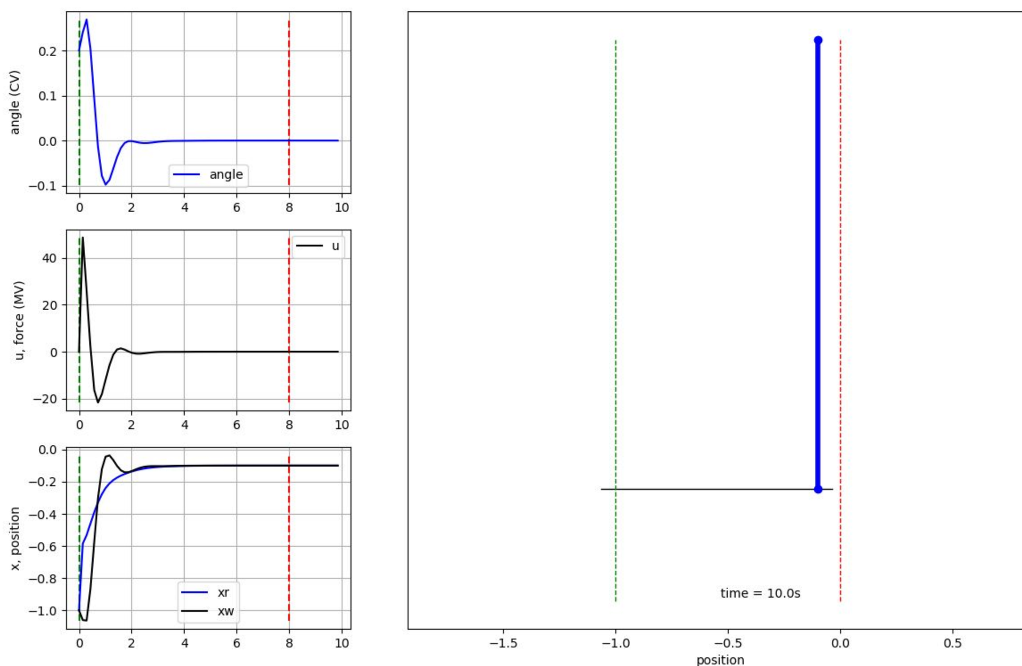
Code sample 2.

```
from scipy.integrate import odeint

for i in range(len(t) - 1):

    Y = odeint(modelDerivFunc,
            [θrz[i], θr_dotz[i], θwz[i], θw_dotz[i]],
            [t[i], t[i+1]], args=(uz[i], ))
    θrz[i+1], θr_dotz[i+1], θwz[i+1], θw_dotz[i+1] = Y[1].T

    # Read in measurements from the system (odeint)
    m.theta.MEAS = θrz[i+1]
    m.theta_d.MEAS = θr_dotz[i+1]
    m.phi.MEAS = θwz[i+1]
    m.phi_d.MEAS = θw_dotz[i+1]
```

```
# solve MPC model
m.solve(remote=True)

# Readout new manipulated variable values
uz[i+1] = m.tau.NEWVAL
```

3. Robot angle control with PID

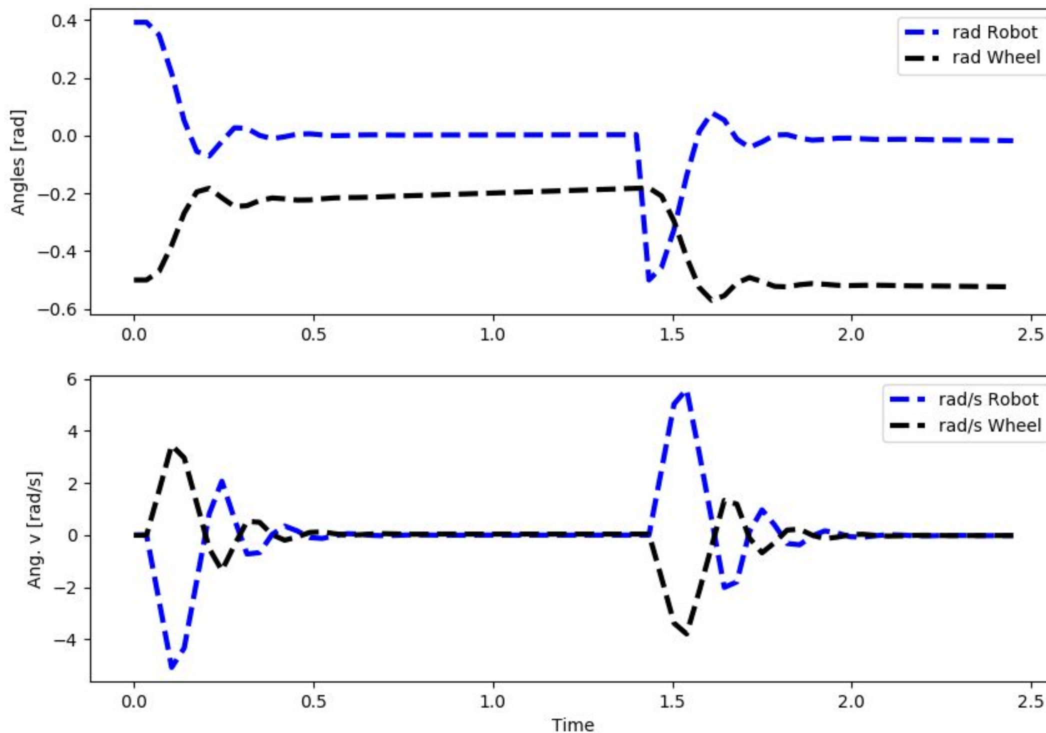
A simple proportional-integral-derivative (PID) controller (actually without the integral) was implemented to test the odeint simulation. The PID was manually tuned and de-tuned to see the effects. As expected, if the PID was de-tuned or turned off, the pendulum would fall down.



4. Robot angle control (iterative) with MPC

This next step included designing a MPC with a much shorter horizon (less foresight) so it could react faster on deviations in the robot angle. Again, we took the general dynamic equations given in the earlier sections of this report. A short horizon of 3-4 time steps each 0.035 seconds long was used for the regarding MPC. Since a fixing on a specific time step in the future is not a feasible solution in this case, we started using setpoints of the controlled variables (CV). Therefore, the CV of the robot angle theta was set to 0, which means it is in an upright position. The code can be found in the Appendix.

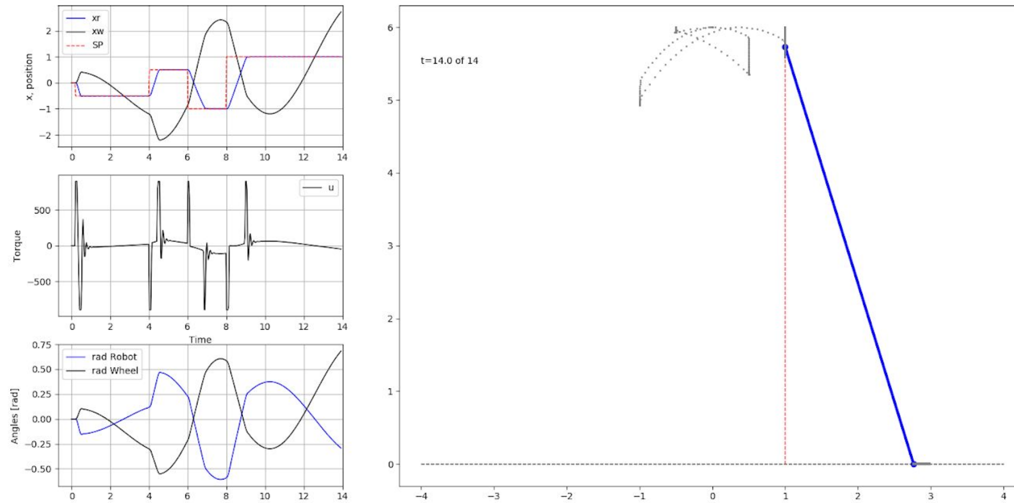
To test this new MPC just for balancing the robot we started the robot in a tipped starting position and added some “pushes” to the angle of the robot in the later time-frame. The system was able to balance the ODE depicting the real system measurements. The initial tipped position is balanced after around 0.5 seconds (meaning nearly no movement of the robot) and the recovery time after a big at 1.4 seconds needs around the same time. This all seems like good results, but there seemed to be a small offset between the ODE results and the MPC after a bigger number of pushes in different directions, which resulted in a system with some small deviations from the set points. Since the pushes were only done at the ODE, so that the MPC could just measure the changes, we assumed it was because of the weights on the measurement use of the variables for the MPC. Although we defined to use 100% of the measurement for the MPC, it did not seem to follow the measurement but rather take also a part from its calculated values. However the overall the balance MPC did a good job at holding the ODE in an upright position at a reasonable calculation time.



5. Position control (iterative)

This next step included designing MPC control, manipulating drive torque to control the robot angle *and* position. The simulation below shows several positional setpoint changes and the

robot position (the upper end of the robot) following the setpoint changes.



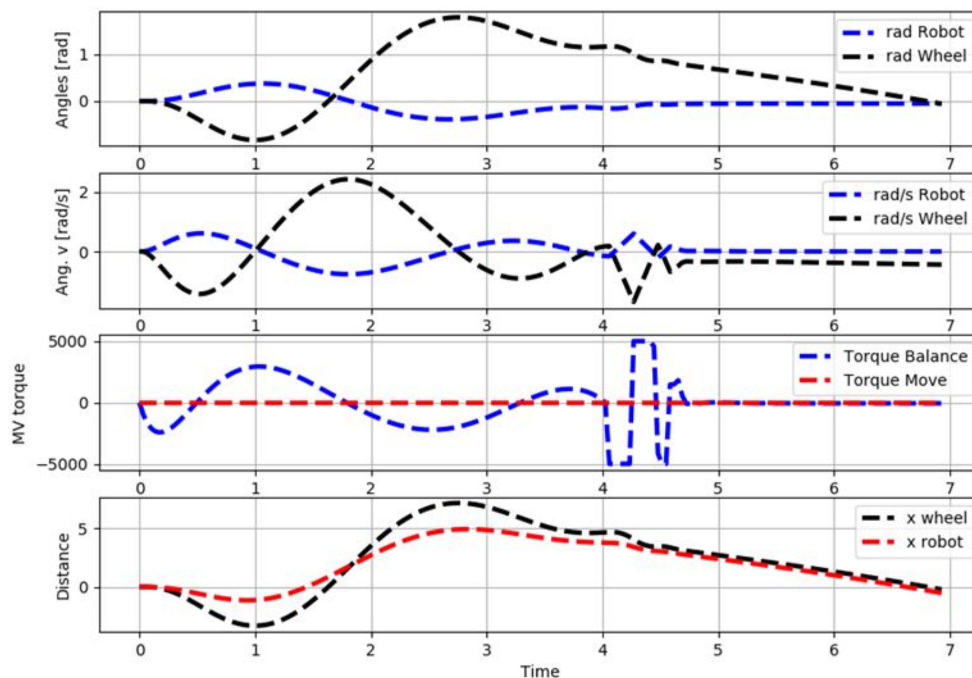
Unlike the first Full horizon MPC simulation, this simulation did not stabilize the wheel position at a final time - the wheel would continue to move while keeping the robot position at its setpoint. Attempts were made using gekko.fix() and gekko.Obj with a final objective, but results weren't as good as the first Full horizon MPC simulation.

Detail of first Full horizon MPC simulation showing robot position and wheel position converging on setpoint	Detail of MPC simulation showing robot position converging but wheel still in motion

6. Combined approach (first batch then iterative)

One approach included an initial calculation of the trajectory to reach a specific target in a given amount of time, while not tipping over the robot. To be able to include both criteria we fell back on the first model described in this section. Before starting the ODE, which could resemble moving the robot away from a steady position, a GEKKO model with long foresight preemptively calculated a trajectory of tau values. This trajectory was able to drive the robot with the given criteria from its starting location to the new set point. At this set point the previously described balance control would take over to control the robot in an upright unmoving position. The

balancing worked quite well, but keeping the ODE in one spot was not working as good as we wanted. The robot would either move, while being nearly upright or just tip over while staying in one location. The problem was to set the two criteria at the right values (which was hard since it always leads to one being slightly more important). There was also some mismatch between the ODE and the MPC even between the measured values. So even after reducing mismatch due to some bad parameter, defining the measured and calculated theta would deviate from each other. The overall idea worked well in our opinion and could lead to a useful application if it is fast enough for the real beaglebone. The robot could be balanced by the balancing MPC, while another MPC is calculated a new motor activation trajectory to drive it to a new set point. At this new point, it could balance while getting new data for another set point and so on.



Implementation

One of the robots we purchased (the Zumo 32U4) comes with an example program that balances it on it's end. We tested this and it is only partially effective. It uses the built-in gyroscope, accelerometer, and PID controller to attempt to maintain a vertical position. When we tested it on a hard floor, we found that it regularly lost control, accelerating to the left or right before falling over. The fact that this robot is running an Arduino-compatible microcontroller at a very fast processing rate illustrates how challenging the implementation task is.

Next steps

Currently, the biggest challenge ahead is implementing an iterative MPC strategy that controls the robot position and wheel position (or robot angle) as satisfactorily as the full horizon MPC simulation.

Once an MPC strategy is designed, the next step would be to run it on the robot using a remote solver (GEKKO) via a WiFi connection to a local machine with more processing power than the onboard microcontroller.

Suffice to day, we do not expect the model to work first time on the robot. One of the main reasons this is unlikely is probably going to be differences between the simulation model used to design the controller and the actual environment (the robot in the real world).

If this proves ineffective (too long turnaround time for next control signal) we will need to think of an alternative way to implement MPC. One option is to develop an 'explicit MPC' solution, for example by pre-calculating control actions and populating a lookup table that could be implemented on the robot and that the code could quickly access in real-time. [4]

Conclusion and Learning

The project was a great chance to test the knowledge and skills that we learned from the course on a 'real' problem. We found that attempting a challenging problem like this (segway bot control) involved a lot of problem-solving and increased our understanding and competence with the tools and techniques.

In particular, we encountered the following challenges which provided significant learnings:

- Short vs long time horizons - the mobile inverted pendulum has a noteworthy characteristic; the control variables often need to be moved in the opposite direction to the set-point in the short term in order to reach the target in the long term. This forced us to think about the time horizon and the separate (but connected) control objectives of (i) keeping the robot in the vertical position and (ii) driving it to the desired x-position.

- Model mis-matches – simulation versus MPC. Deciding to simulate the model dynamics separately from the control system (in odeint and GEKKO respectively), forced us to tackle the more realistic issue of matching the controller's model to the 'real' robot (represented by our odeint code).

- Visualizing the system – because the segway is a mechanical system it is easy to represent visually. Writing an animation script that provided a live visualization of the system during the simulation and also a video sequence at the end was very useful for

our intuition and problem-solving.

- Tuning the MPC – because the segway required a well-designed and well-calibrated controller we were forced to spend considerable time researching and experimenting with the many control parameters and settings of the APMonitor solver in GEKKO.

- Speed of solver – It quickly became apparent that the speed of the MPC model was way too slow to operate in real time. This raised the important realisation that implementation of the eventual controller on the real robot would require a significant speed-up for it to work in real-time.

- Unfortunately, the time we spent fixing and calibrating the MPC controller and investigating different types of controller meant that we did not get chance to test our solutions on the robots yet.

Possible Future Research

One solution to the problem of implementing the full MPC solver on the robot is the possibility of generating a continuous function approximation (mapping function) of the optimal control actions from the state variables.

One way to do this might be to use a neural network to ‘learn’ the function-mapping from repeated examples of MPC control signals generated by GEKKO. There may be other methods.

A second potential application of machine learning could be to train a neural network to model the system dynamics purely from actual sensor data. If this is possible with a reasonable amount of data, it could reduce the amount of effort needed in building models from first principles and the process of parameter estimation or model fitting.

The second potential application could be in implementing the MPC control policy on the robot. If a neural network can be built that is capable of approximating the role of the MPC solver in producing optimal control signals for given state variable values, this could be a way to speed up and simplify the implementation of the controller. However, the ability of a neural network to comprehensively replicate the control policy with sufficient accuracy is the question.

Finally, reinforcement learning is an area of artificial intelligence that has been around for a long time but is only recently starting to result in major achievements, particularly in robot motion planning and control.[5] Could a reinforcement learning algorithm achieve the ‘end-to-end’ learning task of developing an optimal control policy for the robot simply with the inputs from the sensors and the control objective. If possible, this would essentially replace the whole MPC development process. Another benefit could be to allow some kind of ‘online learning’ to occur

that would ensure that the controller adapts to any changes in the robot or environment that may occur over time.

This avenue of research will require study of research from other fields such as robotic control and artificial intelligence.

[5] Reinforcement Learning: An Introduction. Second edition, in progress. November 5, 2017.
Richard S. Sutton and Andrew G. Barto, 2014, 2015, 2016, 2017
The MIT Press

Acknowledgments

We would like to thank Professor Hedengren for his time and dedication to this course and the other staff at Brigham Young University for helping us complete this rewarding project.