ECE 4704 Project 2

Matthew Zhong

Introduction, Administration Issues, and Author’s Note

The chances are 50/50 that either you’re Dr. Williams or Sarah Maxseiner. If you’re Dr. Williams, congratulations – you already know what’s up. If you’re Sarah, chances are, you probably don’t have nearly as much information as to how I had made plans to do Project 2, so let me explain what’s going on.

Because of how I’m not the biggest fan of MatLAB or Mathematica, for Project 2, rather than designing a traditional robot and deriving its forward and inverse mechanical properties by hand or with one of the aforementioned tools, I wrote my own robot simulator from scratch in C++.

I began with the Simple Directmedia Layer library (SDL), which allows me to create a 2D window into which I can write text and draw lines. From this point, I built the rendering and coordinate transformation logic myself. I have to say, the results aren’t perfect, but I’m immensely proud of what I have produced, and I hope you enjoy reading and seeing these results as much as I enjoyed producing them.

This project is divided into several distinct parts. The first part is the flashy video – you can watch it either from the video I’ve uploaded on Canvas, or you can watch it in YouTube at the following link:

<https://www.youtube.com/watch?v=gBRRl-2vftM>

The second part is this report. You should probably read this report after watching the video – it’s good to have a brief understanding of what my program is capable of before I get into specifics of how I computed everything. I need to also note that the video was made with a slightly older build of the project, from which I have fixed a few computational issues, namely a major bug with computing the Jacobian matrix with angular velocity.

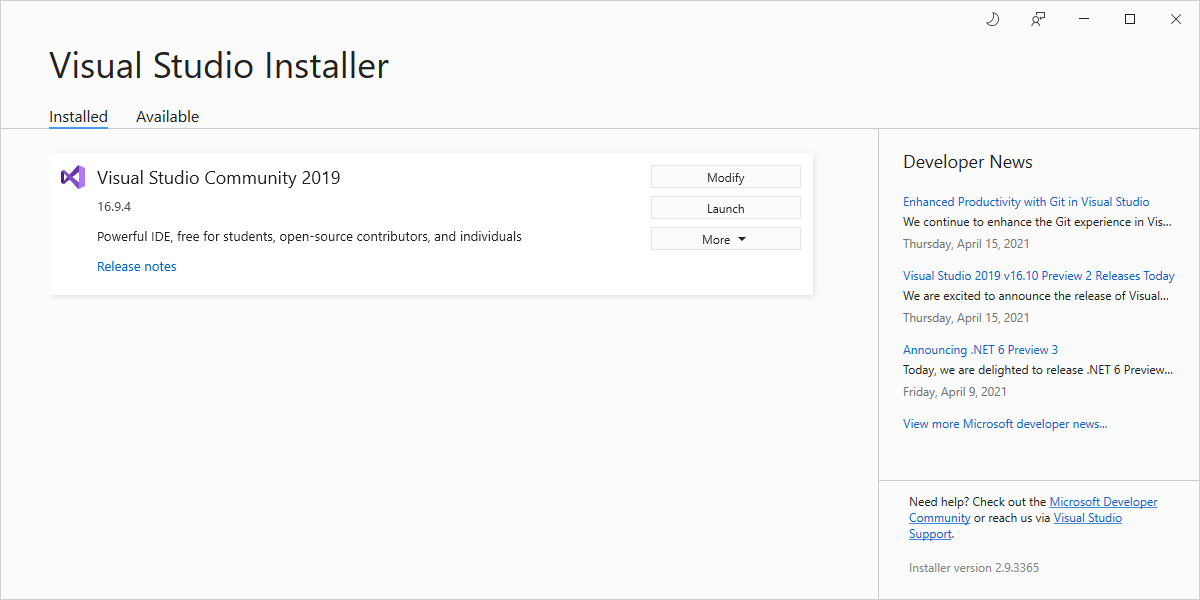
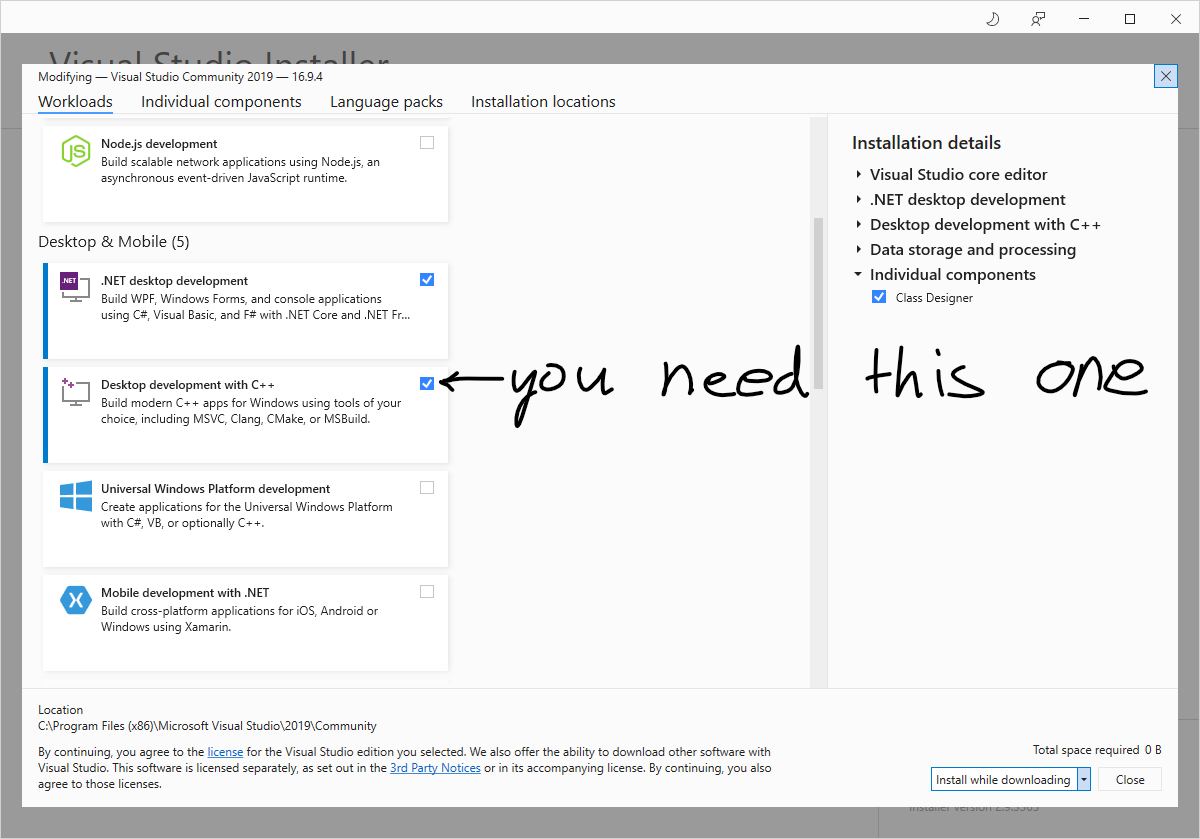
The final part of this project is, well, the code! I’ve uploaded a zip of the Visual Studio solution of this project, and I have also uploaded this solution to GitHub. Currently, the GitHub link I’ve created is private. I have sent invitations to both of you, under the emails [rywilli1@vt.edu] and [sarahbm9@vt.edu], meaning you’ll have to accept the invite link below:

<https://github.com/mzhong99/LaserRobotSimulator/invitations>

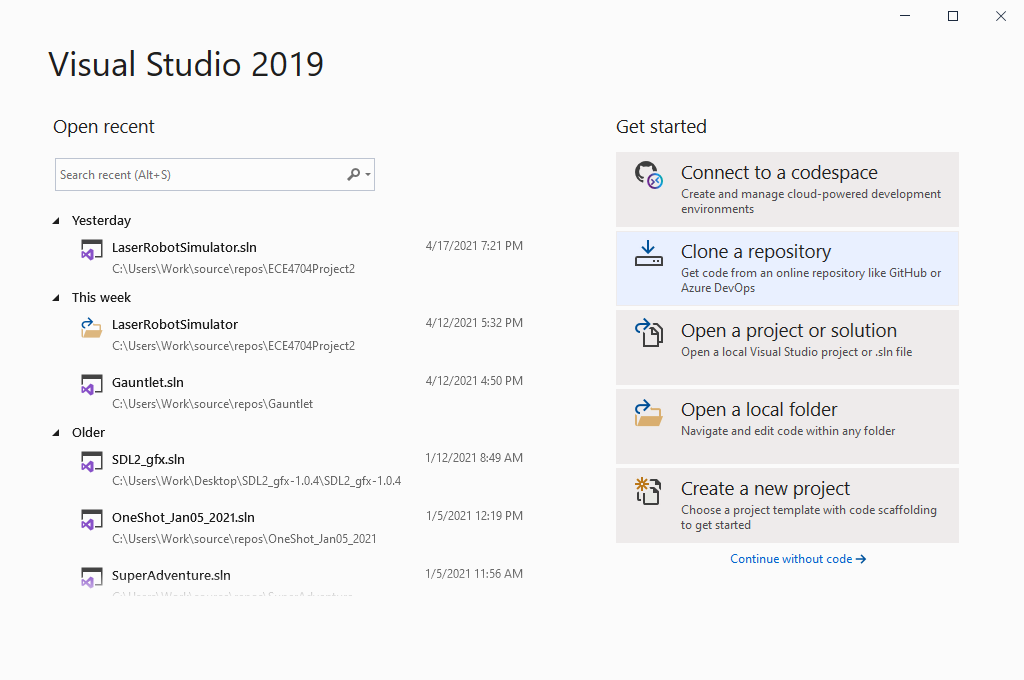
The code is meant for your inspection and compilation – if you want to compile it for yourself, you need to install **Visual Studio 2019**, at the following link.

<https://visualstudio.microsoft.com/downloads/>

Since I have used Visual Studio 2019 to implement my solution, older versions of Visual Studio will not support compilation of this project. If you want to compile and run my code yourself, then this is what you need to do:

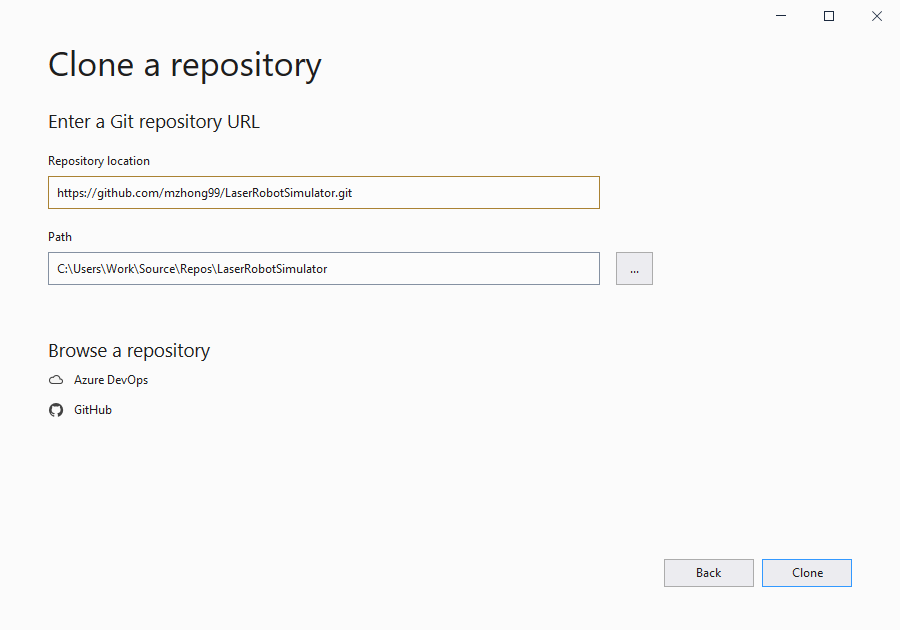
First, install Visual Studio 2019 on **Windows 10.** Make sure that you have “Desktop development with C++” installed as part of the Visual Studio package.  
  


Next, open Visual Studio 2019. When you get to the main bootup screen, you’ll see an option to either create new projects, clone from a repository, and more. Clone my repository, with the **Clone a repository** button.

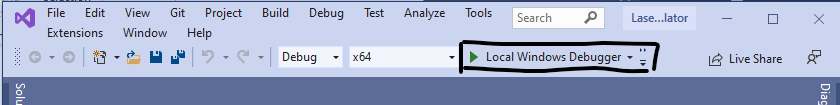


For the repository location, use the GitHub repository I have provided. Alternatively, you can use this direct link if you’re feeling lazy, as long as you have accepted my GitHub invite link from above.

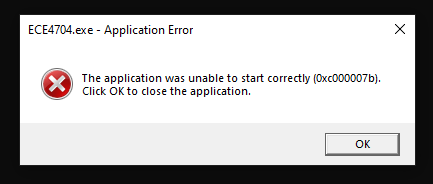
<https://github.com/mzhong99/LaserRobotSimulator.git>



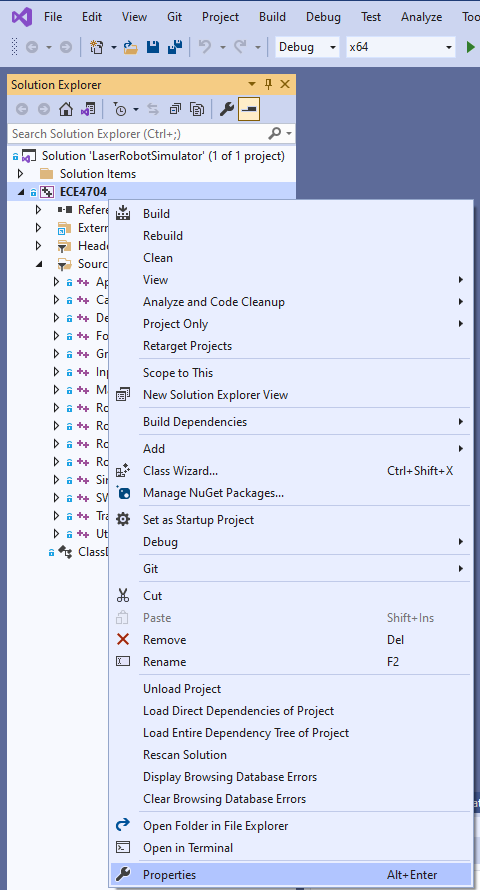
Once you’ve imported the repository, Visual Studio should open. At the top bar of the IDE, press **Local Windows Debugger.** Cross your fingers – if this builds the program and runs it successfully, then you’re done, and can proceed with the rest of this report!

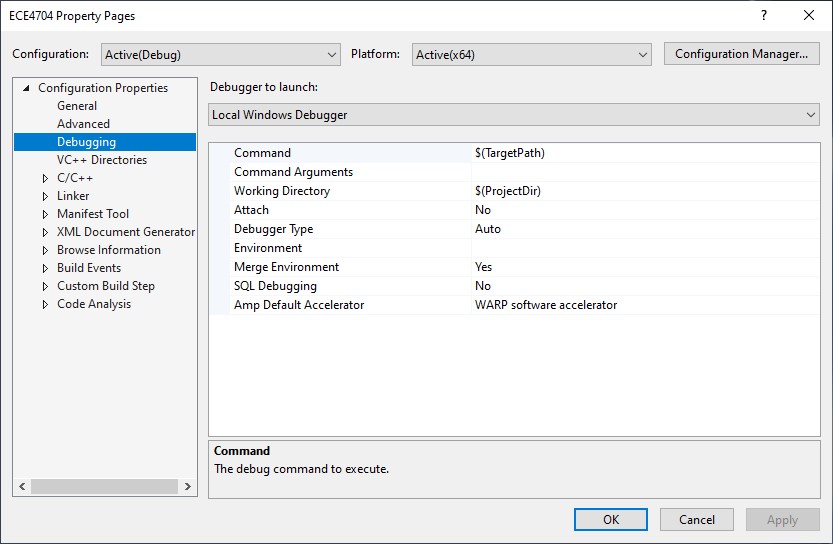


If this didn’t work, however, chances are it’s because the PATH variable I tried to update in Visual Studio didn’t work properly. For example, when I tried to follow these same steps on another PC, this is what happened.



To solve this problem, click the **Solution Explorer** dropdown and select Properties. Then, under **Configuration Properties**, select the **Debugging** option from the list on the left of the window.

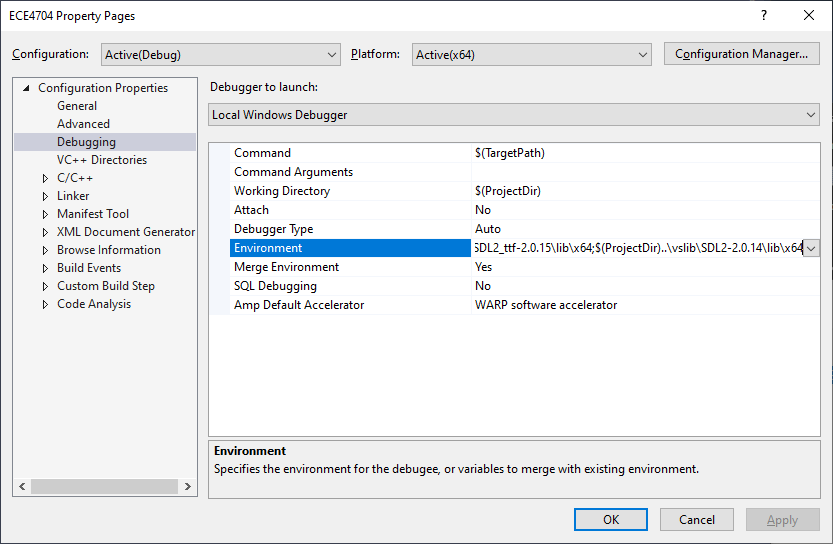




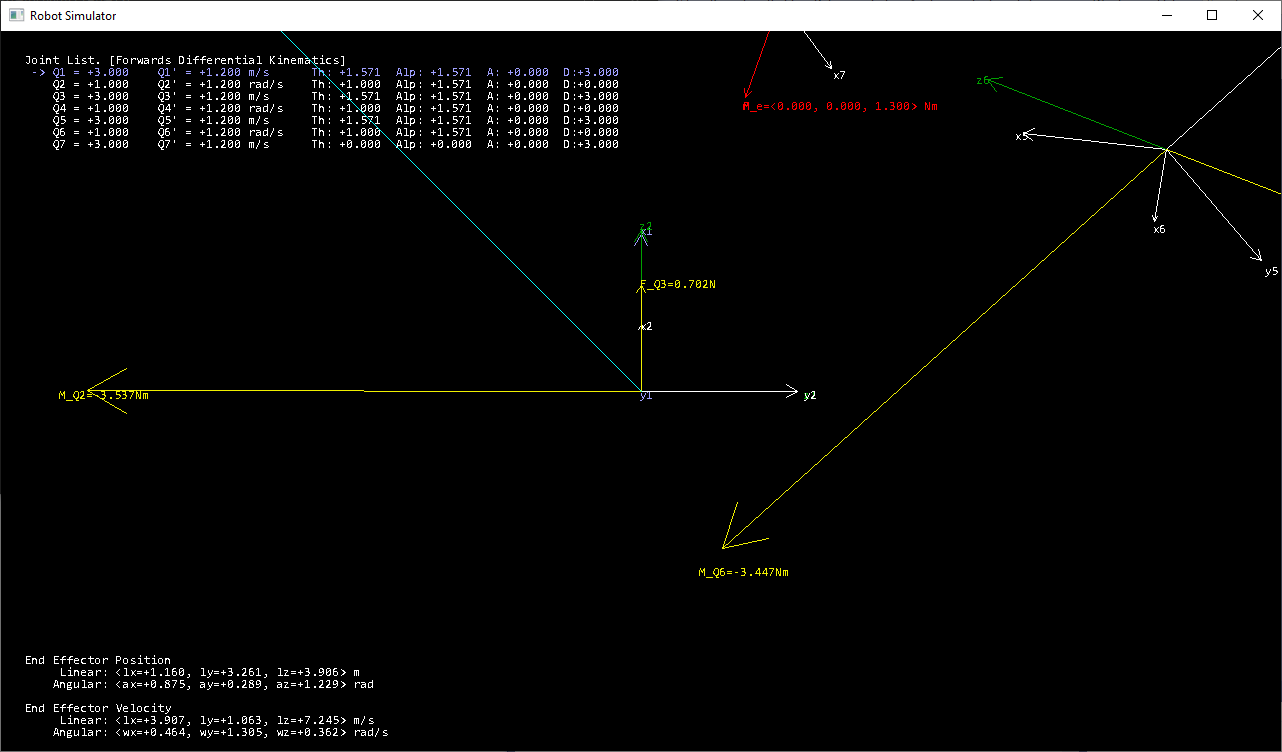
In the Environment box, paste in the following line:

PATH=%PATH%;$(ProjectDir)..\vslib\SDL2\_image-2.0.5\lib\x64;$(ProjectDir)..\vslib\SDL2\_ttf-2.0.15\lib\x64;$(ProjectDir)..\vslib\SDL2-2.0.14\lib\x64

(Note: it’s all one line. Just copy and paste the code snippet above.



Press **OK**, then exit this window. Try to run it again. Hopefully, this fixes the problem I had when I pulled this on another PC. If everything works, you should see my robot simulator running!



If this **still** doesn’t work for you, please don’t hesitate to send me an email at [mzhong99@vt.edu]. I’m more than happy to perform technical support to get this running on your PC, or as a last-ditch alternative, a live Zoom demo at a time of your choice.

**Updates from after the video was made:**

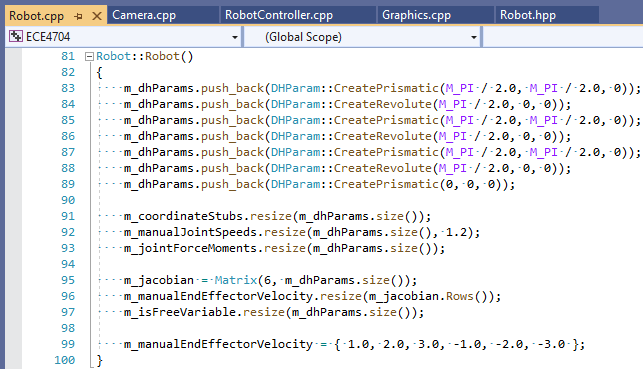
* I fixed my angular Jacobian matrix – running the program computes the correct forward differential kinematics and statics now.
* Control for differential kinematics has changed and is now significantly easier to understand.

Using the Robot Simulator

My Robot Simulator has a rather involved set of controls, which you can use to manipulate and analyze a robot in 3D. At a glance, the controls can be found in the table below:

|  |  |
| --- | --- |
| Keys | Function |
| WSAD, Space, C | Move view location. Treat this like a first-person videogame – to move forwards and backwards, press W or S. To move left or right, press A or D. To gain a higher view, press Space, and to gain a lower view, press C. |
| Middle Mouse | Click and drag the middle mouse button to pan the camera in 3D. |
| Left and Right Arrow OR 1-9 | Use the left and right arrow keys OR press the 1-9 keys to select a joint. |
| Control+1-9 | Hold control, then press 1-9 to toggle showing the joint based on the key selected. |
| Shift+1-9 | Hold shift, then press 1-9 to hide all other joints except for the key selected, the prior joint, and the base coordinate frame. |
| Backtick (left of the 1 key) | Press Backtick to show all joints, including those which you may have hidden from Control + 1-9 or Shift + 1-9. |
| P | Press P to toggle whether isometric or perspective rendering is used. Isometric rendering is better for analyzing the directions of certain vectors, while perspective rendering is easier to tangibly parse. |
| J | Press the J key to toggle whether Jacobian |
| T | Press the T key to toggle whether statics vectors are shown. |
|  |  |
|  |  |
|  |  |
|  |  |

The only thing which is currently runtime-unsupported is changing the DH joint parameters themselves. If you want to change the DH parameters, you must manually modify them in Robot.cpp, in the constructor.



There are two currently-supported joint types – prismatic and revolute. Observe the function headers for the DHParam struct to determine how to create new joints.

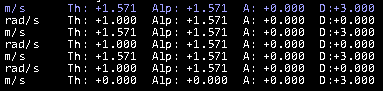


Note that currently, this system expects over-constrained kinematic chains, so at least six joints. This simplifies calculation in several interesting ways, which I’ll discuss in the Theory section.

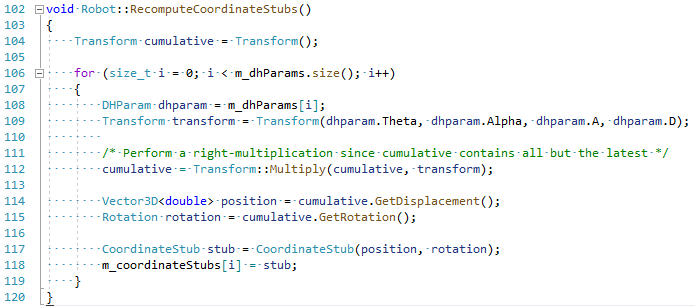
Theory and Implementation

In this project, we were supposed to either extend the robot which we created in the previous project or build a new robot with the same or more joints. My solution is a step beyond that – I will discuss and analyze how this program analyzes a seven-joint robot’s forward kinematics, forward differential kinematics, inverse differential kinematics, manipulator statics, and joint reaction statics.

Let’s first talk about my algorithm to derive the forward kinematics of a robot. For reference, the DH parameters I will refer to are below. Prismatic joints are denoted with a m/s mark, and revolute joints are marked with a rad/s mark. The default D parameter for a prismatic joint is 3.0, and the default theta parameter for a revolute joint is 1.0 radians.



We know that the forward kinematics of each joint can be computed iteratively using a series of homogeneous transformations created by DH parameters. In essence, this means that my program would need to perform the same algorithm to achieve the same results.



Above is the solution that I use to derive my forward kinematics. Rather than just having relative homogeneous transformations saved, I also directly compute the homogeneous transform needed to relate each joint’s coordinates in terms of the **base frame**. Each coordinate stub produced essentially stores the rotation and orientation needed to represent the coordinate stub in terms of the base frame. It’s also important to note that my code is zero-indexed – i.e., the coordinate stub at index **zero (0)** corresponds to joint **one (1)**, and so on. Matrices and vectors are similarly indexed.

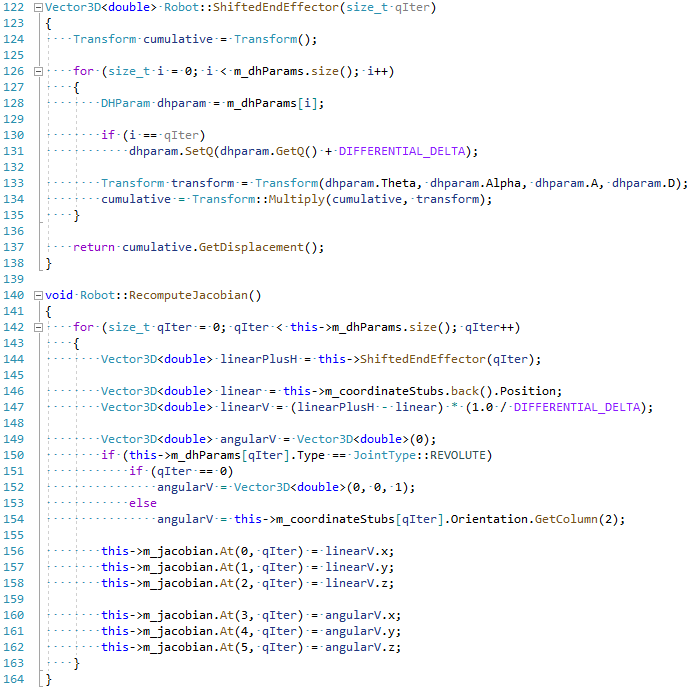
Next, to derive the forward differential kinematics of my system, I needed to perform a Jacobian matrix transform. Since I wasn’t about to implement my own symbolic differentiator, I needed to find something more practical for use. The angular Jacobian was actually more straightforward to implement since I just applied Theorem 3.3 from the textbook. Obviously, prismatic joints don’t rotate, so I zero out the prismatic joint columns, but the theory is the same.



For the linear segment of the Jacobian, I elected to perform finite-difference analysis. Rather than learn a new formula which I did not understand nearly as well, I simply analyzed the changes in the end effector position when manipulating each joint by a small delta. Essentially, for each column of the linear Jacobian, I varied the joint parameters using the formula below.



The algorithm this translates into from my code looks is shown below. The differential delta used is 1e-10.

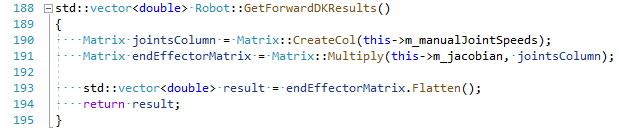


While I was tweaking this robot, I discovered some interesting ways to avoid singularities in my Jacobian. Specifically, I noticed that the reason why singularities existed was because the joint structures we’ve been presented with do not cover all ranges of motion. Notice that my robot, in contrast, has three prismatic joints (one for each of the x0, y0, and z0 axes) and three revolute joints, each of which are situated in such a way that the joints rotate about different axes. By covering the full range of motion required, I can completely avoid a robot which could potentially have singular motion.

It follows that my forward and inverse differential kinematics calculations naturally resolve themselves. Following the formula as was discussed in class, I simply multiply the Jacobian by corresponding joint velocities to compute the end effector’s linear and angular velocity.



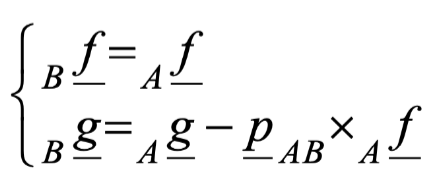
The corresponding algorithm used in my program is the easy to understand:



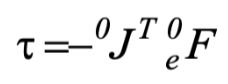
For inverse differential kinematics, the process taught in class involves inverting a matrix. Since inverting a matrix is not always feasible for my solution, I can instead perform Gauss-Jordan elimination on an augmented matrix which contains the Jacobian on the left and the end effector linear and angular velocity on the last column. This also allows me to detect whether free variables exist in my solution, meaning in my simulator’s inverse differential kinematics computation, you can actually adjust the free variables yourself.



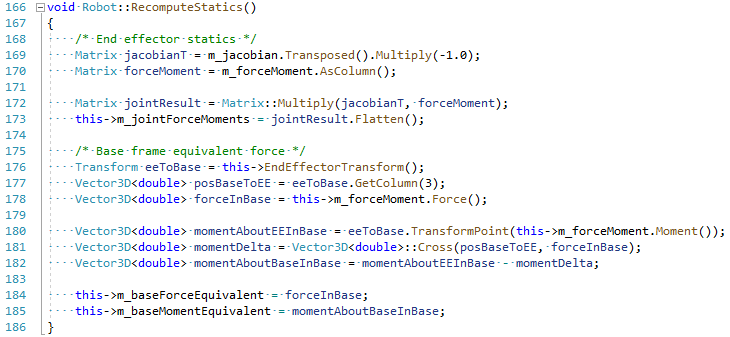
The final portion of this software involves statics computations. Given an arbitrary force and moment couple applied on the end effector, I elected to translate this couple to the **base frame**. After some debugging, the method that worked best for me was using the first-principles formula, wherein I compute the additional moment applied manually through the cross-product formula.



Additionally, to compute the joint statics of the robot presented, I simply use the transposed Jacobian formula, where the transposed Jacobian is multiplied by -1.0 and then by the applied force-moment couple to obtain the joint forces and moments for each joint. Interestingly enough, this formula also works equally as well for forces as it does for moments since the formula relies on the **principle of virtual work.** Rather than multiplying a torque by a small change in angle Q, the prismatic work equivalent multiplies a force by a small change in displacement… which is still Q. Because of this, the formula supplied in the slides can be directly used with no harm.



You can see the equivalent processes represented by algorithms in my code below.



**Refer to the YouTube video I linked for diagrams and examples showing each of these, and play around with the program yourself if you want more diagrams and examples!**