Worcester Polytechnic Institute

Master's Thesis

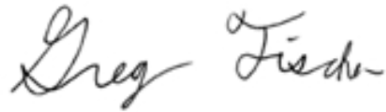
Implementation of a Modular Software Architecture on a Real-Time Operating System for Generic Control over MRI Compatible Surgical Robots

Author:
Katie Gandomi-Bernal

A thesis submitted in conformity with the requirements for the degree of Master of Science in Robotics Engineering

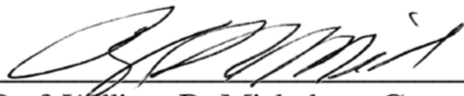
April 25, 2018

Approved By:



Prof. Gregory S. Fischer, Advisor
Worcester Polytechnic Institute

Prof. Loris Fichera, Committee Member
Worcester Polytechnic Institute



Prof. William R. Michalson, Committee Member
Worcester Polytechnic Institute

Abstract

Software used in medical settings operate in complex and variable environments. Programs need to integrate well not only with their electrical and mechanical components, but also within the socio-technological setting they participate in. In this Master's Thesis, a modular software architecture for controlling surgical robot systems within magnetic resonance scanners is designed and implemented. The C++ program runs on a sbRIO 9651 real-time operating system and an object oriented design is taken. Robot kinematics and controls are put into effect in software and validated. Communication with up to ten daughter cards occurs via SPI and external information is exchanged via OpenIGTLink. A web-based engineering console made with ReactJS is also constructed to provide a visual interface for actuating motor axes and executing robot functionality. Documentation of the code is provided and the program was validated quantitatively with software tests and qualitatively through experimentation in MRI suites.

Acknowledgements

This thesis becomes a reality with the kind support and help of many individuals. I would like to extend acknowledgment and thanks to all of them.

Foremost, I would like to thank Professor Gregory Fischer, my thesis advisor, for the opportunity to work on the exciting research happening at the WPI AIM Lab. Your guidance and mentorship throughout this process has influenced my work and inspired me to continue exploring the topic of medical robotics.

I would also like to express gratitude to my committee members Professor Loris Fichera and Professor William Michalson for their comprehensive counsel and insightful feedback that largely shaped the evolution of my thesis.

In addition, a tremendous thank you to my colleagues at the WPI AIM Lab who encouraged me to analyze my work critically, pushed me to work earnestly, and helped me think through challenging problems whole-heartedly.

Finally, I am highly indebted to my family, for without their love and support, I certainly would not be where I am today. Thank you !

Contents

1 Introduction & Literature Review **1**
1.1 Medical Robots & Software . . . 1
1.2 Regulations for Software on Medical Devices . . . 2
1.3 Prior Art . . . 3
1.4 MRI Compatible Surgical Robot System Architectures . . . 3
1.5 Medical Devices Software Design & Interoperability . . . 4

2 AIM Lab - MRI Surgical Robot Platform **5**
2.1 Platform Overview . . . 5
2.2 NI Module & Role of the C++ Program . . . 5
2.3 Daughter Cards & Robot Actuators . . . 6
2.4 Platform Specific Tools & Terminology . . . 7

3 Project Overview **9**
3.1 Thesis Contributions . . . 9

4 Software Architecture **10**
4.1 Design Overview . . . 10
4.2 C++ Low-Level Software . . . 13
4.2.1 Object Oriented Design . . . 13
4.2.2 Robot Kinematics . . . 15
4.2.3 Main Loop WorkFlow . . . 17
4.2.4 Robot Motors & Encoders . . . 18
4.2.5 SPI Communication . . . 22
4.2.6 OpenIGTLink Communication . . . 22
4.2.7 Implementation of Robot Controllers . . . 25
4.2.8 Other Robot Components & Utilities . . . 31
4.3 Developer's Web Application . . . 31
4.3.1 React Front End . . . 31
4.3.2 C++ HTTP Server API . . . 34
4.3.3 Robot Specific Functionality & State Machine . . . 38
4.4 A Generic Software Architecture . . . 39

5 Experimentation **40**
5.1 Validation: Real-Time Loop Rates . . . 40
5.2 Validation: Robot Targeting . . . 42
5.3 Validation: Robot Controllers . . . 45
5.4 Validation: NeuroRobot Surgical Workflow . . . 48
5.5 Validation: ProstateRobot Surgical Workflow . . . 50
5.6 Validation: Software Tests & Code Documentation . . . 52

6 Conclusion and Future Directions **54**

A Appendix **55**
A.1 Code Repository & Documentation . . . 55

References **56**

List of Figures

1 NeuroRobot . . . 1
2 Prostate Biopsy Robot . . . 1
3 Surgical Robot Platform System Architecture . . . 5
4 NI Module in the Robot Control Box . . . 6
5 ZFrame . . . 6
6 NeuroRobot Coordinate Frames . . . 7
7 Prostate Robot Coordinate Frames . . . 8
8 Diagram of Code Design . . . 10
9 UML Diagram of Neuro Robot . . . 11
10 UML Diagram of Prostate Robot . . . 12
11 Flow Chart of Main Loop . . . 17
12 Code Flow Chart . . . 26
13 Trapezoidal Velocity Generation Example . . . 28
14 Main Page of the Web Application . . . 32
15 Login Page for the Web Application . . . 32
16 Chrome Web Developer Console . . . 37
17 State Machine Drop Down Menu . . . 38
18 Plot of Profiled Main Robot Loop - Jitter . . . 41
19 Plot of Profiled SPI Loop - Jitter . . . 41
20 Plot of Profiled Main Loop - Jitter Improved . . . 41
21 Plot of Profiled Main Loop - Jitter Improved Scaled Y-axis . . . 41
22 Histogram of Profiled Code . . . 42
23 Histogram of Profiled Code - Zoomed In . . . 42
24 AAPM Phantom . . . 43
25 NeuroRobot MRI Targeting Example 1 . . . 43
26 NeuroRobot MRI Targeting Example 2 . . . 43
27 Targeting Tests with the NeuroRobot . . . 43
28 Targeting Tests with the ProstateRobot Angle 2 . . . 44
29 Plot of Profiled Velocity Control . . . 46
30 Plot of Profiled Position Control . . . 46
31 Plot of Profiled Control Output . . . 47
32 The Neuro Robot Experiment at UMass Memorial Hospital . . . 48
33 The Neuro Robot Surgical Work Flow . . . 49
34 The Prostate Robot Experiment at Brigham and Women's Hospital . 50
35 The Prostate Robot Surgical Work Flow . . . 51
36 An Example of Test Suite Output with all Tests Passing . . . 53
37 An Example of a Test Suite Output with one Test Failing . . . 53
38 Sample of Doxygen Code Documentation . . . 53

List of Tables

1 Targeting Results for NeuroAblation Robot . . . 42
2 Targeting Results for Prostate Robot . . . 44

Listings

1 Abstract Robot Class . . . 14
2 Abstract Kinematics Class . . . 15
3 Forward and Inverse Kinematics for the Biopsy Robot . . . 15
4 Motor Object Configuration . . . 19
5 MoveMotor() method . . . 19
6 SetMotorOutputSignal() Method . . . 20
7 Encoder Position Calculation . . . 21
8 OpenIGTLink Sync() Method . . . 22
9 OpenIGTLink ThreadIGT() Method . . . 23
10 Trajectory Following Method . . . 27
11 Trapezoidal Velocity Profile Implementation . . . 28
12 Joint Level Velocity Controller . . . 30
13 ControlCommandToMotorOutputSignal() Method . . . 30
14 GET Request . . . 35
15 PUT Request . . . 36
16 NeuroRobot Update() Method . . . 38
17 An Example of a Unit Test . . . 52

1 Introduction & Literature Review

In this section this thesis is introduced and prior art on the research topic discussed.

1.1 Medical Robots & Software

Software used in medical settings operate in complex and variable environments. Programs need to integrate well not only with their hardware, but also with the socio-technological setting they participate in. Doctors, surgeons, and nurses who work closely with these systems should consider a software a helpful tool rather than a complicated burden. Robots will begin to be more involved in medical operations as technology advances, and there will be a need for well written programs that can be used to singularly and generically control different systems.

Medical robots, in particular, need to have high quality, reliable software in order to ensure the safety and effectiveness of automated surgical procedures [20]. The program needs to be, among other facets, real-time, robust, and simple. Real-time operating systems (RTOS) can make a program more deterministic by guaranteeing the code will run at strictly defined times. Robustness implies that the software will not cause unexpected behaviors during the expected use-cases. Simple programs do not require an excessive amount of training before they can be used and are intuitive to both their developers and end-users.

The work presented in this thesis strives towards the criteria required of a software that controls surgical robots. The program aims to serve as a generic platform that can be extended to different systems while also maintaining an organized coding environment. The robotic systems shown in Figures 1 and 2 demonstrate two motivating examples for this type of architecture.

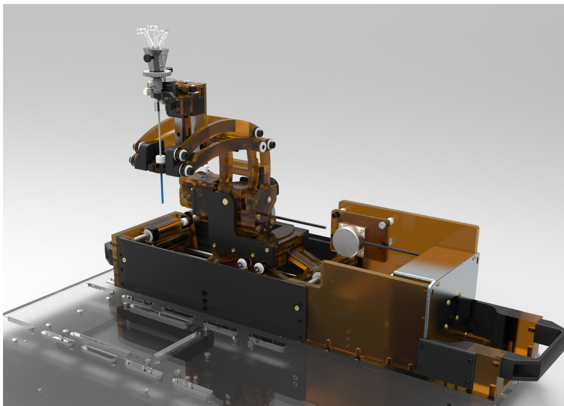


Figure 1: The figure shows the Neuro Robot [18] used for thermal ablation of brain tumors

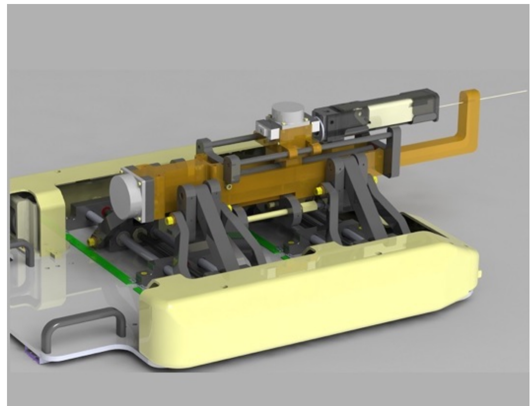


Figure 2: The figure shows the Prostate Robot [4] used for prostate cancer biopsies

1.2 Regulations for Software on Medical Devices

Existing government regulations for medical device software are largely focused on programs embedded in dedicated hardware and are focused around physical harm, invasiveness, proximity to sensitive organs, and transmission of diseases [20]. There is also extensive literature on software applications used for medical diagnostics, simulation, and servers [1, 5, 17]. The International Organization of Standards (ISO) and the Food and Drug Administration (FDA) provide regulations for software used in medical devices.

The ISO is an international regulatory organization that maintains standards for a variety of applications and fields. ISO 13485:2016 [10] is used to direct the quality management of products by maintaining well documented and controlled procedures. IEC 62304:2006 [11] specifically defines the life-cycle for software within medical devices.

The FDA of the United States approves medical devices for clinical use. 21 CFR §820.30 [7] lists a set of requirements that must be met during the design of medical devices and also applies to the development of software. The FDA classifies medical devices into three categories : Class I, lowest risk, Class II, moderate risk, and Class III, high risk. Surgical robots such as the DaVinci, DigiMatch Robodoc, the Armeo Robotic Arm exoskeleton, and RIO Surgical Robot Arm are classified as Class II medical devices and must meet Class II regulations.

In [9], of 3,140 medical devices recalled between 1992 and 1998, 242 instances where attributed to software failure, 192 of which were caused by software defects introduced when changes were made to the software after its initial production and distribution. This FDA guide further outlines software engineering practices to assist medical device manufacturers in avoiding such defects and resultant recalls. Namely, software verification and software validation are championed. The former includes composing comprehensive documentation, software testing, and code inspections, and is continuously performed throughout the development life cycle. The latter is performed on a finished device to ensure, through objective metrics such as experimentation and examination, that the software conforms to the use-cases it was constructed for.

[9] also notes that it can be difficult to gauge when software verification and validation are complete or when enough evidence has been collected. It recommends that a certain "level of confidence" be achieved and that the amount of confidence needed to move forward should depend on the hazard posed by the automated functions of the device. Software tests are not enough to fully verify that a program is complete and correct, and the guide suggests supplementing with experiments in simulated environments and a structured,documented development process to

provide comprehensive evidence that a medical device behaves as expected.

The standards and guidelines listed have shaped the work in this thesis. They have motivated the use of stringent verification and validations to obtain confidence in the software architecture. Life cycle maintenance and development have been also been considered in addition to design safety. While achieving full clinical approval is out side the scope of this work, the developed software has laid a strong foundation toward this goal.

1.3 Prior Art

The prior art section of this paper presents different software architecture taken by various medical and surgical robot platforms.

1.4 MRI Compatible Surgical Robot System Architectures

Magnetic Resonance Imaging (MRI) compatible surgical robot systems have become a topic of interest in research regarding automated medical procedures. In contrast to CT and Ultrasound scans, MRI can produce better soft tissue contrast for localizing tumors, tissue, and organ structures [14, 16]. Robots that can be placed within the bore of the MRI can use these scans for precise image guided intra-operative procedures [4, 18]. The design and implementation of software architectures on these systems impact the performance and safety of these robots.

Garnette et. al [8] developed a four computer architecture for MRI guided stereotaxy and micro-neurosurgery. A state-machine is a fundamental component of this system and allows for a higher level of confidence and security in the software since robot states can only be entered after predictable series of events. A heartbeat to the main computer is also monitored and if not detected all motion in the robot is stopped.

Yang et. al[26] proposes to improve breast cancer biopsies by using a teleoperated master-slave robotic system to perform these procedures under continuous MRI scanning. This work shows that haptics can be used as a different modality for controlling the robot through software. In this work communication between the master and slave systems is performed by a dedicated Ethernet connection between the control computers.

Pneumatically actuated MRI Compatible Robots have been explored in research as well due to their inherent ability to be made of non-ferromagnetic components. [27] investigated the controllability of one such robot with long transmission lines by developing three custom regulators for actuator position. Stoianovici et. al also developed a pneumatic robot that featured a watchdog system as a safety

mechanism. Their system also allows scan images to be transferred over the network for analysis.

Tokuda et. al presented a software system for intuitive navigation of transperineal prostate therapy for an MRI-guided robot. In this system six states called "workphases" provide synchronization with the clinical workflow. The solution also consists of a user-interface that guides the operator through these workphase states.

[21] investigated the use of real-time needle placement in the MRI with a Linux based workstation for robot control and navigation.

The systems discussed present a subset of representative software architectures used on MRI compatible surgical robots. They compare different control schemes, safety measures, and design structures that have shaped the work in this thesis.

1.5 Medical Devices Software Design & Interoperability

Software in medical devices and applications can also be used for a diverse set of situations and scenarios including but not limited to diagnostics, simulation, and servers [1, 5, 17].

Tahmasebi et. al [1] developed a medical examination simulator that integrates with a PhantomMTM haptic controller to provide users with real-time interactions with virtual data and patients in a simulated environment. The program serves as both a diagnostic tool for medical professionals and a vizualization method for different medical scans. The simulator follows a fully object oriented design structure and a graphic rendering back-end.

Schorr [19] explores different interoperability modes for control over surgical robot systems by developing a distributed network that can leverage information obtained by the various connected devices. This novel object distribution architecture enables networked integration of a robot control server and multiple clients to enhance performance. The system is safe-guarded against unauthorized entities by introducing a security control host that validates a clients access to the robot server.

Chakravorty et. al [3] proposes a novel service architecture named MobiCare for enabling a wide range of health related services through a mobile patient monitoring infrastructure that uses cellphones and clinical sensor systems. The client-server based system leverages HTTP requests to build a number of services and dynamically update the code as needed.

The software in the medical applications discussed have used different system architectures, communication structures, and modes of interoperability. The server as representative works in this topic and were their design decisions were taken into consideration when implementing the software architecture in this thesis.

2 AIM Lab - MRI Surgical Robot Platform

In this section the MRI Surgical Robot Platform at the WPI AIM Lab is detailed.

2.1 Platform Overview

The software developed in this Master's Thesis can be better understood by introducing the surgical system architecture it operates within. The current medical robot platform at the WPI AIM Lab can be separated by whether components are placed in the scanner console room or near the MRI machine. The surgical robot is placed in the bore of the scanner and is connected to the control box via a 151 pin, 22-ft long shielded robot cable. A Footpedal is connected to the control box to enable robot actuation when pressed. The control box also passes a fiber cable through the patch panel, a separator between the MRI room and the scanner console room. Safe from the magnetic fields of the MRI, the interface box receives the other end of the fiber cable. Network connections from the router within the interface box are broken out to the main computer, external connections, and the scanner console to allow for communication with the robot through the robot control box. A general overview of this platform is shown in Figure 3.

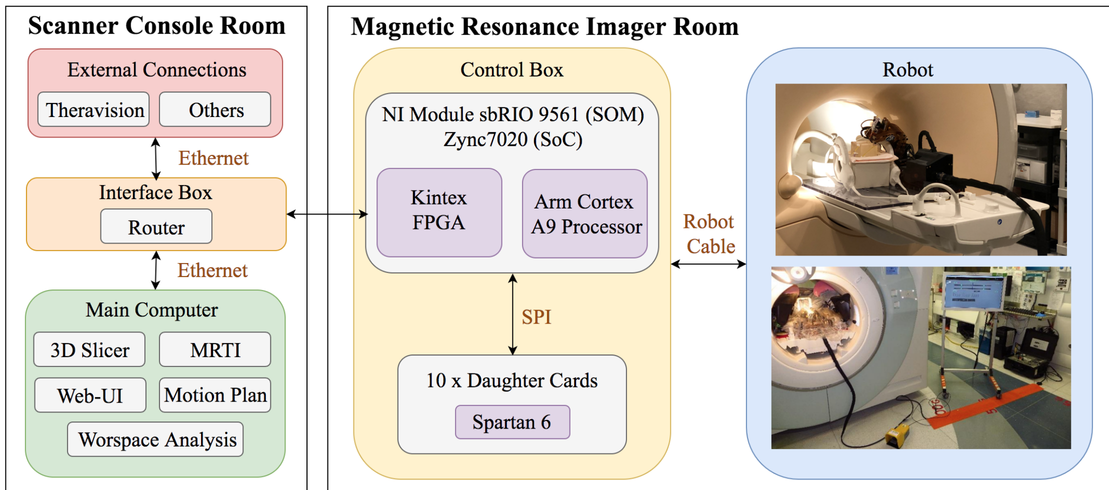


Figure 3: The robotic architecture used in the WPI AIM lab. Major components include the robot, control box, interface box, and the MRI scanner.

2.2 NI Module & Role of the C++ Program

The research in this paper is focused on the C++ program that exists on the sbRIO-9651 National Instrument (NI) System on a Module (SoM) that is located within the robot control box as shown in Figure 4. The NI Module contains a Kintex FPGA and an Arm Cortex A9 processor. The former communicates via SPI with up to 10 Spartan 6 Daughter Cards, the driver interface for a variety of actuators. The latter

is a dual-core processor that features a proprietary real-time Linux operating system for embedded systems. This is where the low-level C++ architecture developed in this thesis resides. This implemented software is responsible for actuating the robot within the MRI, exchanging data with external connections, and high-level functional planning. The program serves as a generic platform intended to be configured to work with different MRI-compatible surgical robotic systems. Two robots are currently set-up to use this system: a neurosurgery robot for thermal ablation of brain tumors and a prostate cancer biopsy robot. These systems are shown in Figures 1 and 2 respectively. Relevant coordinate frame transformations for these systems are presented in Figures 6 and 7 and are referenced throughout this thesis.

The C++ program also performs various functions by receiving commands from a web-interface running on the main computer. This computer may also relay information about the robot's workspace, motion planning, and real-time MRI thermal data from other applications.

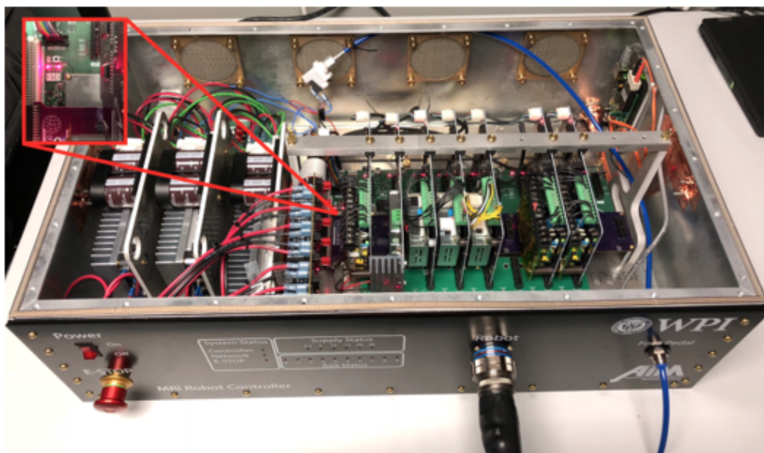


Figure 4: The sbRIO-9651 NI Module shown next to power supplies and Daughter Cards within the control box.

Figure 5: The figure shows the ZFrame used on the Neuro-Robot for registration.

2.3 Daughter Cards & Robot Actuators

The Daughter Cards used in the control box manage low level actuation of robot motors through Verilog code written for a Spartan 6 FPGA. These cards relay low level information to the C++ code via SPI. The Daughter Card type determines what mode of actuation and control is available for the motor. Three types of cards for motor actuation are currently considered : pwm, analog-input, and high-frequency. The application is extensible to a variety of custom card types. The pwm mode of actuation is controlled by sending a duty cycle to the Daughter Card from

the C++ code. Analog input mode can provide a control signal input that can be used to regulate motor movement. The high frequency mode has the most options for customization as the user can set a wave form frequency and amplitude to obtain a desired drive signal. The Daughter Cards can also be used to read and interpret analog signals from sensors such as load cells and encoders. In some cases, there are stand-alone Daughter Cards for only sensors.

2.4 Platform Specific Tools & Terminology

Tools and terminology specific to the project are introduced here. Registration is the process of calculating the transformation from the scanner coordinate frame to the robot coordinate frame. This is calculated by using the Acoustic MedSystems Theravision device used in [15] on MRI T2W Coronal Scans of a detected Z-Frame, shown in Figure 5. OpenIGTLink [24], is a standard communication protocol that is commonly used in operating rooms. It is used in the AIM Lab system to exchange data between external devices and the robot code. 3D Slicer, or simply Silcer [6], is an open source visualization software for medical image processing, and 3D rendering. Each MRI scan can be exported as a set of DICOM images which can be visualized and manipulated in Slicer. DICOM, Digital Imaging and Communications in Medicine, is a standard for handling and storing medical images. These tools and there role in the developed C++ software architecture will be detailed in the upcoming sections of this thesis.

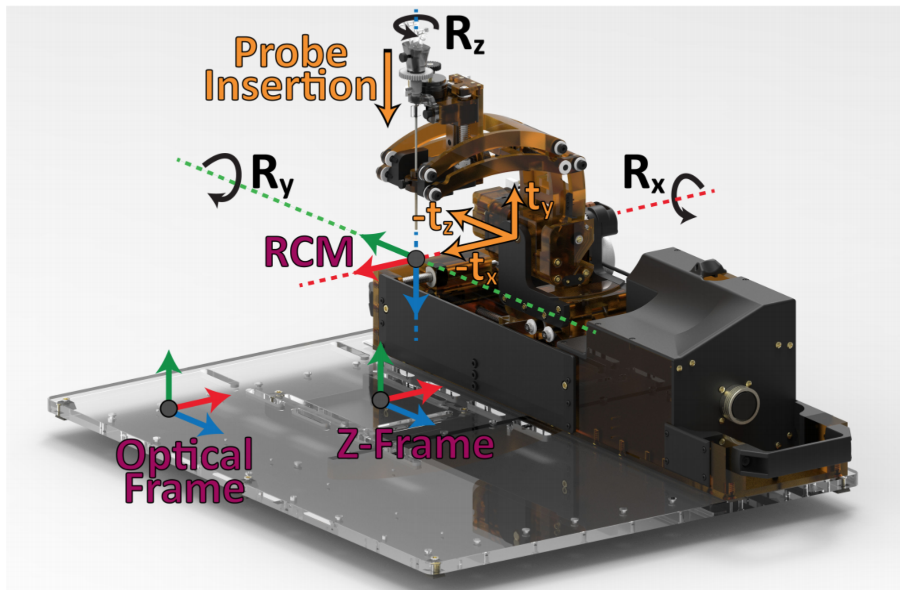


Figure 6: The figure, produced by Christopher Nycz, shows relevant NeuroRobot Coordinate Frames.

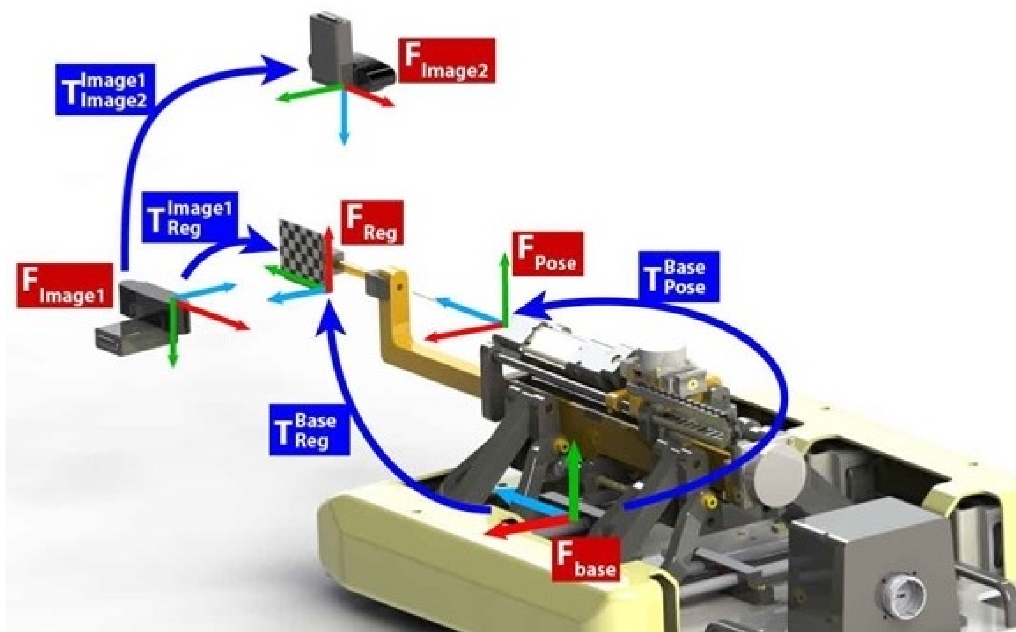


Figure 7: The figure produced by Marek Wartenberg, shows relevant Prostate Biopsy Coordinate Frames.

3 Project Overview

In this section the project goals and objectives are detailed.

3.1 Thesis Contributions

The principal contribution of this thesis is an updated low-level C++ program that runs on the NI-module Arm Cortex within the robot control box. The need for a re-implementation of the software was due to the shortcomings of the previous code. Namely, the old program was difficult to extend/maintain, it was difficult for new members of the lab to get started with, it depended on legacy software, and the project requirements have evolved since the code was first conceived. Thus, the detailed contributions of this Master's Thesis are as follows:

- Develop a set of system requirements that address the shortcomings of the previous low-level C++ software
- Design and develop a modular and extensible low-level C++ architecture
  - The system should be designed as a modular architecture extensible to a variety of robots with the goal of implementing it on two robotic systems currently in the lab
  - The system should follow an object oriented approach
  - The system should include the Forward/Inverse Kinematics for Robots
  - The system should allow for position and velocity control
  - The system should allow for multi-threading for Communications (SPI, OpenIGT, HttpServer)
  - The system should have a new Web-based Developers UI
- Create a detailed user-manual of the code
- Validate the code through experimentation
  - Validate real-time consistency of communication lines
  - Validate robot targeting
  - Validate functionality of position and velocity controllers
  - Demonstrate the software's capability in the surgical operation workflow
  - Validate and safe-guard functionality with unit tests

4 Software Architecture

In this section the design and implementation of the developed C++ foundational code base and the ReactJS web application are elucidated.

4.1 Design Overview

The generic software architecture is a extensible system that is implemented and demonstrated on two MRI-compatible surgical robots at the WPI Aim Lab. This code base can be discussed in two parts: the low-level C++ implementation and the high-level developer-specific web console. The first discusses the design of the low-level program, presents why specific decisions where made, and details the various classes and their purpose in the overall software scheme. A general overview of this C++ code is presented in the diagram in Figures 8 and UML Diagrams of the code are shown in Figure 9 and 10. The developer's web console, in the second section, was created for engineers or operators to directly access and control the robot during the surgical workflow, more details in section 6.4. This part of the paper also details the development of the web-based user interface (UI).

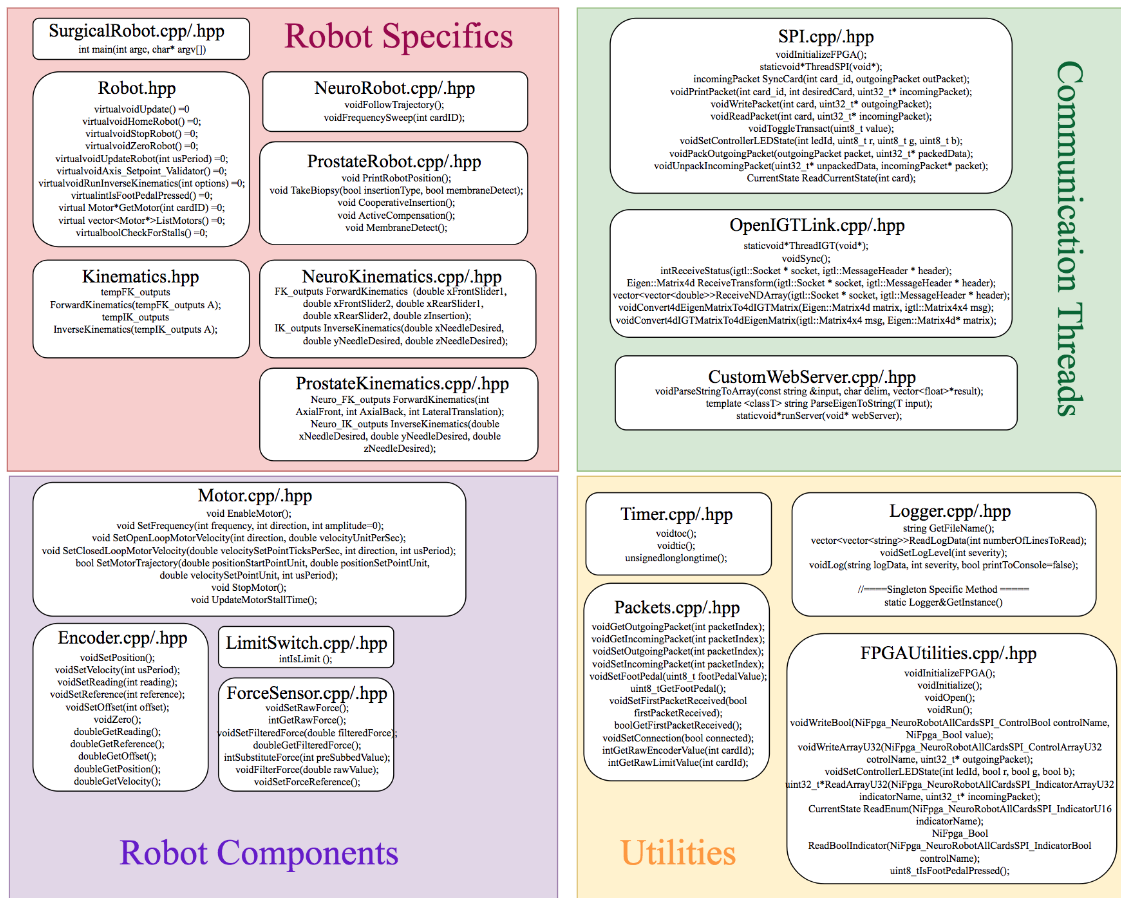


Figure 8: The figure shows a diagram that outlines the general design of the low level C++ software, class objects, and object functions.

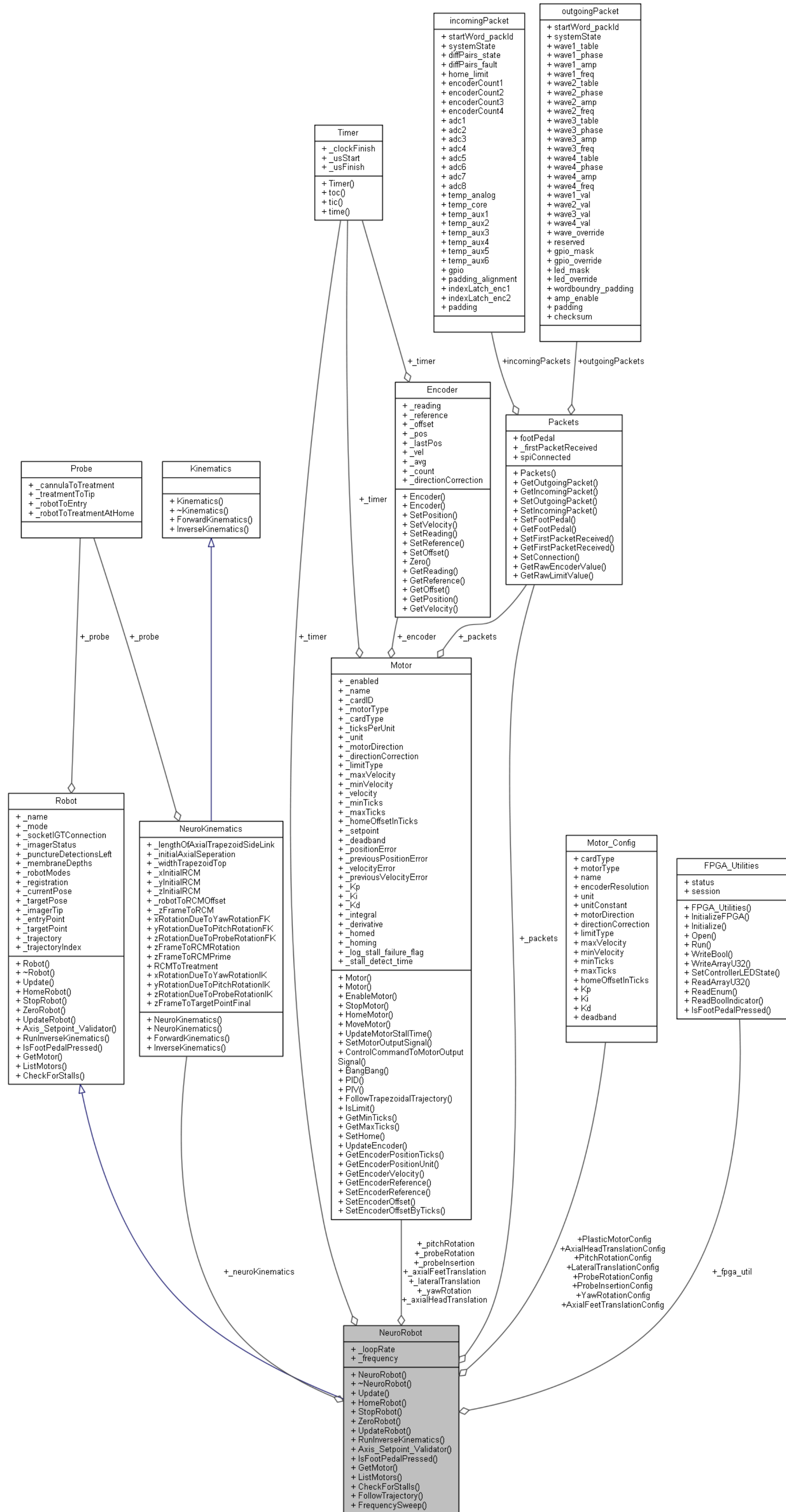


Figure 9: The figure shows a UML Diagram of the code structure for the Neuro Robot.

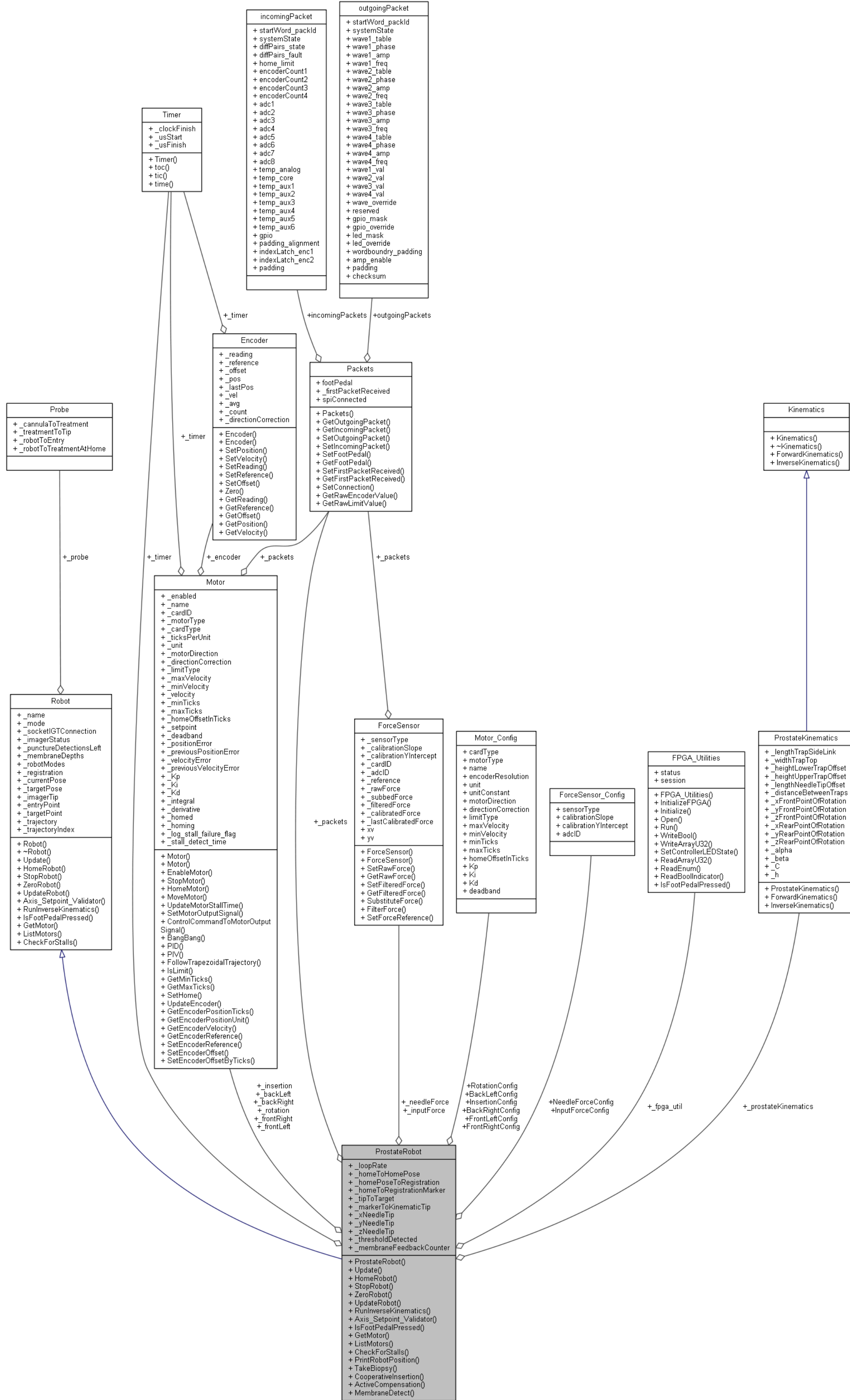


Figure 10: The figure shows a UML Diagram of the code structure for the Prostate Robot.

4.2 C++ Low-Level Software

4.2.1 Object Oriented Design

An object oriented design (OOD) approach was selected for use in the C++ code because it makes a program more modular, extensible, and maintainable. Objects, instances of a C++ class, encapsulate data and functionality that can be re-used in other places of the program. Abstract classes and interfaces can provide strict guidelines on how child classes must be written. The OOD approach also forces programmers, current and future, to plan out their code in advance before they start developing.

The corner stone of the developed low-level C++ software is the abstract Robot class. Children of a parent abstract class must provide an implementation of all pure virtual methods. The abstract Robot class, shown in Listing 1, contains only pure virtual methods, making it an "interface". Interfaces in C++ serve only as guides for methods every child must implement, but do not provide a default implementation. Since every robot is unique, this forces programmers to write these methods for their specific robot configuration, while at the same time, enabling code re-use.

Two classes currently extend the abstract Robot class: NeuroRobot and ProstateRobot. If a child class does not implement one of the abstract parent's pure virtual functions, the compiler throws an error and will not build. Another reason these methods are being forced on the programmer, is that they are used in other sections of the program. For example, the RunInverseKinematics pure virtual method is used generically in both the OpenIGTLink and HttpServer classes whenever new targetting information is available. Thus, this is being forced on the developer because if the function is not implemented the code will not run properly. Another example is the Update function which is called from the main file and handles the state-machine that directly runs robot functionality.

The abstract Robot class also contains a number of class member declarations. These are variables that should be initialized in all children of the abstract parent class. In this case, however, the compiler will not stop the programmer from continuing with out initializing these class members, but sections of the program may not function properly if these are not defined. The comments in the abstract robot class and in the documentation inform and recommend that the developer initialize these variables in the child class constructor. It is at their discretion to initialize these class members in new child classes.

Listing 1: This listing shows the abstract robot class in the C++ code.

```
class Robot {
public:
  //================ Constructor ================
  Robot() {};
  virtual ~Robot(){};

  //================ Parameters =================
  // An object of type robot must have the following parameters
  public :
  string _name; // Name of the given robot
  string _mode; // Defines the current robot mode
  string _socketIGTConnection; // Defines the status of the IGT Connection, the string is
      "Connected" when an IGT socket is connected
  string _imagerStatus; // Defines the current status the Imager (ie: MRI, Camera's, Ultrasound,
      etc), the status can report the stage of the procedure (ie "Valid Target", etc)

  Probe _probe; // Defines the specific probe structure on the robot

  int _punctureDetectionsLeft; // To preserve asynchronous behavior, global puncture index gives
      the current membrane the robot should expect during biopsy procedures
  vector<double> _membraneDepths; // Specifies estimates for membrane depths to detect
  vector<string> _robotModes; // Specifies robot specific methods

  Eigen::Matrix4d _registration; // Transformation between the imager and the robot's reference
      frame
  Eigen::Matrix4d _currentPose; // Transformation that represents current location of the
      treatment zone
  Eigen::Matrix4d _targetPose; // Transformation that represents desired location of the treatment
      zone
  Eigen::Matrix4d _imagerTip; // Transformation between the imager and the robot's treatment zone
      with respect to imager

  Eigen::Vector3d _entryPoint; // Entry point for the desired pass-through location of the
      treatment zone
  Eigen::Vector3d _targetPoint; // Target point for the final location of the treatment zone

  vector<vector<double>> _trajectory;// Defines a set of time stamped joint positions that the
      robot should follow
  unsigned int _trajectoryIndex = 0; // To preserve asynchronous behavior, global trajectory index
      gives the current setpoint the robot should follow

  //================ Public Methods ================
  // An object of type robot must have the following parameters
  // Main Method responsible for robot functionality
  virtual void Update() = 0;

  // Robot specific Methods for various robot specific functionalities
  virtual void HomeRobot() = 0;
  virtual void StopRobot() = 0;
  virtual void ZeroRobot() = 0;
  virtual void UpdateRobot(int usPeriod) = 0;

  // Robot specific helper methods needed by front end gui
  virtual void Axis_Setpoint_Validator() = 0;
  virtual void RunInverseKinematics(int options) = 0;
  virtual int IsFootPedalPressed() = 0;

  // Robot specific Motor related helper Methods
```

```
    virtual Motor* GetMotor(int cardID) = 0;
    virtual vector<Motor*> ListMotors() = 0;
    virtual bool CheckForStalls() = 0;

};

#endif /* ROBOT_HPP_ */
```

4.2.2 Robot Kinematics

To create a new robot, the programmer should create a new child class that extends the abstract Robot class. The programmer then initializes all the class member variables and writes all the pure virtual methods. It is then recommended that the programmer now implement a ChildRobotKinematics class, that extends the abstract Kinematics class, see Listing 2. The abstract Kinematics class has only two virtual methods : Forward Kinematics and Inverse Kinematics. While the programmer is not forced to make this class, it is recommended because kinematics are unique to each robot configuration and to the specific robot use case. Virtual functions were chosen to allow for re-implementation of kinematic parameters. With the abstract Robot class members/pure virtual functions defined and the new ChildRobotKinematic class complete, the last step to finish initializing a new robot class is to configure each motor axis. An example of one such implementation can be seen in Listing 3

Listing 2: This listing shows the abstract kinematics class in the C++ code.

```
#ifndef KINEMATICS_HPP_
#define KINEMATICS_HPP_

struct tempFK_outputs {};
struct tempIK_outputs {};
class Kinematics {
public:
    //================ Constructor ================
    Kinematics() {};
    virtual ~Kinematics(){};

    // Children classes must implement their own versions of Forward and Inverse Kinematics
    tempFK_outputs ForwardKinematics(tempFK_outputs A) { return A;};
    tempIK_outputs InverseKinematics(tempIK_outputs A) { return A;};

};

#endif /* KINEMATICS_HPP_ */
```

Listing 3: This listing shows an implementation of Forward and Inverse Kinematics for the Biopsy Robot

```
Prostate_FK_outputs ProstateKinematics::ForwardKinematics(double xFrontSlider1, double
    xFrontSlider2, double xRearSlider1, double xRearSlider2, double zInsertion) {
```

```
    struct Prostate_FK_outputs FK;

    //*** BASE FORWARD KINEMATICS ***//
    _xFrontPointOfRotation = (xFrontSlider1 + xFrontSlider2)/2;

    double yF_1 = _heightLowerTrapOffset + _heightUpperTrapOffset;
    double yF_2 = pow(_lengthTrapSideLink,2);
    double yF_3 = pow((xFrontSlider1-xFrontSlider2-_widthTrapTop)/2,2);
    _yFrontPointOfRotation = yF_1 + sqrt(yF_2 - yF_3);

    _zFrontPointOfRotation = -_C;

    _xRearPointOfRotation = (xRearSlider1 + xRearSlider2)/2;
    double yR_1 = _heightLowerTrapOffset + _heightUpperTrapOffset;
    double yR_2 = pow(_lengthTrapSideLink,2);
    double yR_3 = pow((xRearSlider1-xRearSlider2-_widthTrapTop)/2,2);
    _yRearPointOfRotation = yR_1 + sqrt(yR_2-yR_3);

    _zRearPointOfRotation = 0;

    _alpha = 0;//atan2(_xFrontPointOfRotation-_xRearPointOfRotation, _distanceBetweenTraps);
    _beta = 0;//atan2(_yFrontPointOfRotation - _yRearPointOfRotation, _distanceBetweenTraps);
    FK.xNeedleTip = (_lengthTrapSideLink + zInsertion)*cos(_beta)*sin(_alpha) +
         _h*sin(_beta)*sin(_alpha) + _xFrontPointOfRotation;
    FK.yNeedleTip = _h*cos(_beta) - (_lengthTrapSideLink + zInsertion)*sin(_beta) +
         _yFrontPointOfRotation;
    //*** NEEDLE DRIVER FORWARD KINEMATICS ***//
    FK.zNeedleTip = (_lengthNeedleTipOffset + zInsertion)*cos(_beta)*cos(_alpha) +
         _h*sin(_beta)*cos(_alpha) + _zFrontPointOfRotation;

    FK.BaseToTreatment << 1, 0,  0, FK.xNeedleTip,
                    0,  1,  0, FK.yNeedleTip,
                    0,  0,  1, FK.zNeedleTip,
                    0,  0,  0,  1;

    return FK;
}

Prostate_IK_outputs ProstateKinematics::InverseKinematics(double xNeedleDesired, double
     yNeedleDesired, double zNeedleDesired) {

    struct Prostate_IK_outputs IK;

    //*** BASE INVERSE KINEMATICS **//
    double xFrontPointOfRotationDesired = xNeedleDesired;
    double yFrontPointOfRotationDesired = yNeedleDesired;

    double xRearPointOfRotationDesired = xNeedleDesired;
    double yRearPointOfRotationDesired = yNeedleDesired;

    //Calculation for Front Slider 1 and 2
    double xF1_1 = 2*xFrontPointOfRotationDesired + _widthTrapTop;
    double xF1_2 = pow(_lengthTrapSideLink,2);
    double xF1_3 = pow(yFrontPointOfRotationDesired - _heightLowerTrapOffset -
         _heightUpperTrapOffset,2);
    IK.xFrontSlider1 = 0.5*(xF1_1 + 2*sqrt(xF1_2 - xF1_3));
    IK.xFrontSlider2 = 2*xFrontPointOfRotationDesired - IK.xFrontSlider1;
    //Calculation for Rear Slider 1 and 2
```

```
    double xR1_1 = 2*xRearPointOfRotationDesired + _widthTrapTop;
    double xR1_2 = pow(_lengthTrapSideLink,2);
    double xR1_3 = pow(yRearPointOfRotationDesired - _heightLowerTrapOffset -
         _heightUpperTrapOffset,2);
    IK.xRearSlider1 = 0.5*(xR1_1 + 2*sqrt(xR1_2 - xR1_3));
    IK.xRearSlider2 = 2*xRearPointOfRotationDesired - IK.xRearSlider1;
    //*** NEEDLE DRIVER INVERSE KINEMATICS ***//
    IK.zInsertion = zNeedleDesired -_lengthNeedleTipOffset;
    IK.zRotation = 0; //FROM OPENIGTLINKTRACKING
    return IK;
}
```

4.2.3 Main Loop WorkFlow

The main loop of the software architecture works as described by the flow chart in Figure 11. The components of the main loop will be presented in greater detail in the following section. The main loop begins with robot sensors and encoders receiving updated information via SPI. The code next checks if the Footpedal has been pressed. If so, it is safe for the robot to enter the robot mode switch case that handles functionality unique to each system. A Motor Move method is then entered to determine if the robot should actuate or not. If it has been determined that the robot should move, the SetMotorOutputSignalMethod should be called and the whole process is repeated. The process is asynchronous, allowing for updates to the robot on the fly, and main loop safety measures stop the robot when the Footpedal is not pressed or if a stall is detected

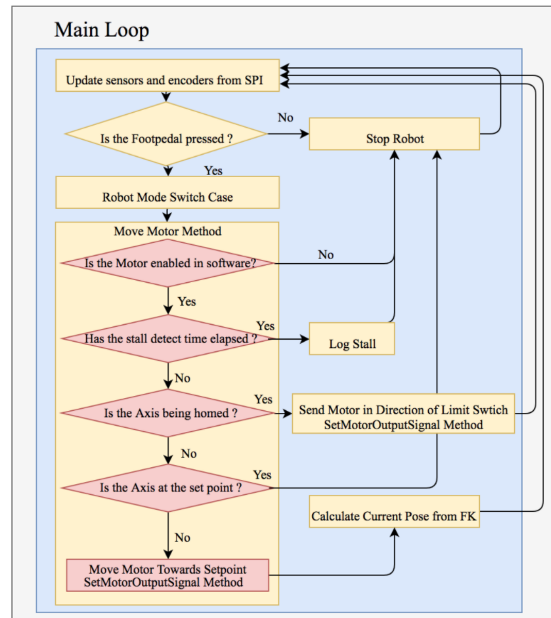


Figure 11: The figure shows a flow chart of the main loop in the developed software Architecture.

4.2.4 Robot Motors & Encoders

The Motor class does not extend from any parent class and is responsible for actuation of an individual motor axis of the robot. Different actuators can be represented in the code by configuring a motor type such as shinsei, dti, or others and a Daughter Card type such as pwm, analog-input, high-frequency or others.

The Motor class takes in a Motor Configuration structure that defines parameters a motor must know about itself, see Listing 5. The Robot class must implement a motor and motor configuration for each axis. The Robot class has a method called Update, shown in Listing 16, that is called in the main while loop. The Update function calls a MoveMotor method for each Robot Motor. If the given motor is enabled in software, the Footpedal is pressed, and the motor's setpoint is not the motor's current position, then the class moves the motor in the direction of the setpoint. The "direction" to move is determined as part of the robot configuration. The motorDirection variable spins the motor clockwise or counter clockwise for a given setpoint. The directionCorrection variable configures what should be considered the positive or negative direction for a motor axis. This allows the programmer to define the positive and negative directions of motion for the coordinate frames on the specific robot, and relate quadrature encoder readings to the robot's relative position.

The motor is actuated via the SetMotorOutputSignal method shown in Listing 6. SetMotorOutputSignal takes in a direction to rotate, a frequency, and optionally a wave amplitude. Depending on whether the motor is of type pwm, analog-input, or high-frequency, a different SetMotorOutputSignal routine is performed. For example: a motor with a pwm card type expects to be given a direction and duty cycle while a motor with a high-frequency card type expects a wave frequency and an amplitude to create a custome drive signal.

Each Motor object contains an Encoder object. The Encoder class receives position information in ticks from the Daughter Cards via SPI and calculates a current position using this received value, the encoder reference, and the encoder offset. The current encoder position is calculated differently depending on the motor directionCorrection variable and the two equations are shown in Listing 7. The encoder reference is set to define a given encoder reading as a zero position or to effectively cancel it out. The encoder offset variable is set to shift the current encoder position, and is used to restore current encoder positions over the web-UI.

Safety of the system was considered when writing the Motor and Encoder classes. If a motor should be moving (because the Footpedal is pressed, the axis is enabled in software, and the setpoint is not the same as the current position of the robot) then if the encoders have not started to count after a 0.5 second interval, the axis is

immediately disabled. This is done to prevent the motor from continuing to move when the encoders have stopped counting.

Listing 4: This listing shows an example for how a Motor object is created inside a child Robot class.

```
ProbeInsertionConfig = {
                      card_type::externaldriver_non_pwm, // Card Type - Ex: pwm, non_pwm,
                          analog_input, high_frequency, unspecified
                      motor_type::shinsei, // Motor Type - Ex: shinsei, piezo, dti, unspecified,
                      "Probe Insertion", // Motor Name
                      5000, // Encoder Resolution
                      "mm", // Axis Units
                      47.24, // ticksPerUnit
                      1, // motorDirection
                      1,  // directionCorrection
                      limit_type::lower, // limitType - Ex: upper, lower, unspecified
                      0x0001ffff, // maxVelocity
                      0x0, // minVelocity
                      0, // Software Limit - Minimum
                      1650, // Software Limit - Maximum
                      0, // _homeOffsetInTicks
                      1, // Control Gain Kp
                      1, // Control Gain Ki
                      0, // Control Gain Kd
                      1 // Deadband
                   };


   _probeInsertion = Motor(packets, 3, ProbeInsertionConfig);
```

Listing 5: This listing shows the Move Motor Method used in the Motor Class.

```
bool Motor::MoveMotor(){
   _timer.tic();

   // If the motor axis has been enabled in software
   if(_enabled){

      // If the encoder values are changing
      if(_encoder._pos != _encoder._lastPos){
         _stall_detect_time = _timer._usStart;
      }

      // Verify that the axis has not stalled
      if(_timer._usStart - _stall_detect_time < 500000){
         _log_stall_failure_flag = 0;
         // Check if the motor axis is currently homing
         if(_homing){
            // If homing, send the motor to its home limit switch
            HomeMotor();
         }

         // Otherwise if the motor axis is not currently homing
         else {
            // Get the current position of the motor axis
            int current = GetEncoderPositionTicks();
```

```
            // Send the axis towards its setpoint
            if(_setpoint < (current-_deadband)){
               EnableMotor();
               SetMotorOutputSignal(_velocity, 1);
            } else if (_setpoint > (current + _deadband)) {
               EnableMotor();
               SetMotorOutputSignal(_velocity, 0);
            } else {
               // If we've reached the setpoint stop the motor axis
               _stall_detect_time = _timer._usStart;
               StopMotor();
               return true;
            }
         }
      }

      // If the motor axis has stalled
      else {
         // Stop the motor and log that a stall was detected
         StopMotor();

         // Disable the motor, and await the user to re-enable on the UI
         _enabled = false;

         // We only want to show the stall log failure once
         if(_log_stall_failure_flag <= 2){ _log_stall_failure_flag += 1;}

         // So if the log stall failure flag is one, show the error, else don't
         // prevents the console from being flooded
         if( _log_stall_failure_flag == 1 ){
            Logger& log = Logger::GetInstance();
            log.Log("The " + _name + " Motor has STALLED !!!!", LOG_LEVEL_CRITICAL, true);
         }
      }
   }

   // If the motor axis has been disabled software
   else {
      _stall_detect_time = _timer._usStart;
      StopMotor();
   }

   return false;
}
```

Listing 6: This listing shows the SetMotorOutputSignal() method in the C++ code. Note the switch case which takes in the motor type.

```
// Note: The amplitude parameter is optional
void Motor::SetMotorOutputSignal(int motorOutputSignal, int setDirection, int amplitude)
{
   int signal;
   int direction;
   int direction1;
   int direction2;

   int actualDirection = (_motorDirection ? setDirection : !setDirection);
   if((IsLimit() && ((!setDirection && (_limitType==limit_type::upper)) || (setDirection &&
```

```
        (_limitType==limit_type::lower))))){
      StopMotor();
      return;
   }

   switch(_cardType){
      case card_type::externaldriver_pwm:
         // Decide the direction and frequency
         direction = (actualDirection) ? 0x00000000 : 0x80000000;
         signal = (direction | (_maxVelocity & motorOutputSignal));

         // Send desired motor response to daughter card
         _packets->outgoingPackets[_cardID].wave1_freq = signal;
         break;

      case card_type::externaldriver_non_pwm:
         // Decide the direction and frequency
         direction = (actualDirection) ? 0x40000000 : 0xC0000000;
         signal = (direction | (_maxVelocity & motorOutputSignal));

         // Send desired motor response to daughter card
         _packets->outgoingPackets[_cardID].wave1_freq = signal;
         break;

      case card_type::highfrequency:
         // Decide the direction and frequency
         direction1 = (setDirection) ? 0x00 : 0x01;
         direction2 = (setDirection) ? 0x01 : 0x00;

         // Send desired motor response to daughter card
         _packets->outgoingPackets[_cardID].wave1_table = direction1;
         _packets->outgoingPackets[_cardID].wave2_table = direction2;

         _packets->outgoingPackets[_cardID].wave1_freq = motorOutputSignal;
         _packets->outgoingPackets[_cardID].wave2_freq = motorOutputSignal;

         _packets->outgoingPackets[_cardID].wave1_amp = amplitude;
         _packets->outgoingPackets[_cardID].wave2_amp = amplitude;
         break;

      default: // Unspecified or unlisted
         break;
   }
}
```

Listing 7: This listing shows the two equations that can be used to set the encoder position depending on the direction that should be consider positive for the robot's coordinate frame.

```
   if (_directionCorrection){
      _pos = (_reference + _offset - _reading); // Calculate the current position
   } else {
      _pos = (_reading - _reference + _offset); // Calculate the current position
   }
```

4.2.5 SPI Communication

SPI is used to communicate between the Arm Cortex on the NI-Module and the Verilog code written on the Spartan 6 on the Daughter Cards. In the C++ software, a dedicated thread was launched for this transaction of information and data was distributed to other classes by a shared pointer to a Packets object; however, due to issues with Jitter, see Section 6.1 the thread was consolidated into the main loop. To safe-guard the data in the Packet object, a mutex is used to lock and unlock critical region access when shared data is manipulated. The SPI class first packs outgoing messages to each respective Daughter Card converting a struct to a uint32*. Then the code waits until the Daughter Cards are ready to send data. Once ready, the C++ program toggle transacts all the Daughter Cards at once, effectively initializing the transactions. The SPI class then unpacks incoming packets from uint32* to a class member struct within the Packet Object. Each incoming and outgoing packet contains 256 bytes of data.

4.2.6 OpenIGTLink Communication

OpenIGTLink is a standardized mechanism for communication among computers and devices in operating rooms for a variety of image-guided therapy (IGT) applications [24]. OpenIGTLink communication is used in the AIM Lab MRI surgical robot platform to exchange data with Slicer and other external connections.

The OpenIGTLink object takes in three parameters: a Robot pointer to the main robot object, a Robot pointer to a cached robot object, and a port to listen on. There are two central functions of this class: the Sync() method and the ThreadIGT() method shown in Listings 8 and 9 respectively.

The Sync() method checks if there are any differences between the main robot object and the cached robot instance. Any detected differences are then sent over to the connected client sockets, if one exists. This method is called in the main while loop.

The ThreadIGT() method is launched in a dedicated thread. It creates the C++ program's OpenIGT server and allows clients to connect / disconnect from it repeatedly. Clients can directly change the value of different variables in the main Robot* object by sending messages over the OpenIGTLink connection. For example, if a client wants to change the Registration transform of the robot, they simply need to send a message with a header name of "Registration" and a data type of "Transform", and the C++ OpenIGTLink Server will interpret that as a request to change the main robot object's registration transform to the value of the transmitted transform.

Listing 8: The listing shows the Sync() method used in the OpenIGT class for external communication. This method is responsible for sending data out from the C++ OpenIGTLink Server whenever a difference is detected between the main robot object and the cached robot object.

```
void OpenIGTLink::Sync() {
  // Given that the socket is not null and we're connected to a client socket
  if (socket.IsNotNull() && _clientSocketConnected != 0) {
    // SEND ROBOT REGISTRATION
    if (!_robot->_registration.isApprox(_cachedRobot->_registration)){
      // Convert Eigen Matrix to IGT Matrix and update the cached robot
      igtl::Matrix4x4 msg;

      for(int i = 0; i < 4; i++){
        for(int j = 0; j< 4; j++){
          msg[i][j] = _robot->_registration(i, j);
        }
      }
      //Convert4dEigenMatrixTo4dIGTMatrix(_robot->_registration, msg);
      _cachedRobot->_registration = _robot->_registration;

      // Create Transformation Matrix to transmit
      igtl::TransformMessage::Pointer registrationMsg = igtl::TransformMessage::New();
      registrationMsg->SetDeviceName("scanner_to_robot_reg");
      registrationMsg->SetMatrix(msg);
      registrationMsg->Pack();
      socket->Send(registrationMsg->GetPackPointer(), registrationMsg->GetPackSize());
    }
    .....
```

Listing 9: The listing shows a subsection of the ThreadIGT() method. This method is launched in a separate thread and is responsible for receiving data from a connected client socket.

```
void* OpenIGTLink::ThreadIGT(void* igt) {
  // For Profiling the code
  Timer _timer;
  //Logger& log = Logger::GetInstance();

  // Get an IGT Object
  OpenIGTLink *igtModule = (OpenIGTLink*) igt;

  // Create New Sockets on the provided port
  igtl::ServerSocket::Pointer serverSocket;
  serverSocket = igtl::ServerSocket::New();
  int r = serverSocket->CreateServer(igtModule->_port);

  // Check if we can create a server socket
  if (r < 0) {
    // If we cannot error back
    std::cerr << "Cannot create a server socket." << std::endl;
    exit(0);
  }

  // While we are listening on this port
```

```
while (1) {

  // Check if we can connect on the client socket
  igtModule->socket = serverSocket->WaitForConnection(1000);

  // Connection Specific Variables State -- Not Connected
  igtModule->_robot->_socketIGTConnection = "Listening";
  igtModule->_clientSocketConnected = 0;

  // If we were able to connect to the client socket
  if (igtModule->socket.IsNotNull()) {
    // Connection Specific Variables State -- Connected
    igtModule->_robot->_socketIGTConnection = "Connected";
    igtModule->_clientSocketConnected = -1;

    // Create a message buffer to receive header
    igtl::MessageHeader::Pointer headerMsg;
    headerMsg = igtl::MessageHeader::New();

    // Allocate a time stamp
    igtl::TimeStamp::Pointer ts;
    ts = igtl::TimeStamp::New();

    // While the socket is not null and we're connected to the the client socket
    while (igtModule->socket.IsNotNull() && igtModule->_clientSocketConnected != 0) {
      // Initialize receive buffer
      // Receive generic header from the socket
      headerMsg->InitPack();

      // To preserve asynchonicity set a time out for how long to wait to receive data
      igtModule->socket->SetReceiveTimeout(1000); // In milliseconds

      // -- The _clientSocketConnected variable becomes zero when the Receive method is no
           longer connected to the client
      igtModule->_clientSocketConnected =
           igtModule->socket->Receive(headerMsg->GetPackPointer(),headerMsg->GetPackSize());

      // Check that the received data is valid, else just listen again
      if (igtModule->_clientSocketConnected != 0) {

        // De-serialize the header
        headerMsg->Unpack();
        cout << "Unpack Header" << endl;

        // For profiling OpenIGTLink Communication
        _timer.tic();

        // Get time stamp
        igtlUint32 sec;
        igtlUint32 nanosec;
        headerMsg->GetTimeStamp(ts);
        ts->GetTimeStamp(&sec, &nanosec);

        // ====== Check the data type and unpack the message! ======
        //  REQUEST: NAME -- CURRENT_POSE & TYPE -- TRANSFORM
        if ((strcmp(headerMsg->GetDeviceName(), "CURRENT_POSE") == 0) &&
             (strcmp(headerMsg->GetDeviceType(), "TRANSFORM") == 0)) {
          igtModule->_robot->_currentPose = igtModule->ReceiveTransform(igtModule->socket,
               headerMsg);
```

```
            }

            // REQUEST: NAME -- REGISTRATION & TYPE -- TRANSFORM
            else if ((strcmp(headerMsg->GetDeviceName(), "REGISTRATION") == 0) &&
                 (strcmp(headerMsg->GetDeviceType(), "TRANSFORM") == 0)) {
               igtModule->_robot->_registration = igtModule->ReceiveTransform(igtModule->socket,
                   headerMsg);
            }
          .....
        }
      }
    }
    // On thread end, close the socket
    igtModule->socket->CloseSocket();
    return NULL;
}
```

4.2.7 Implementation of Robot Controllers

Implementing robot position and velocity controllers require specific infrastructure and methods in the code in order to obtain the desired functionality and behavior. A high level plan for how the code would maintain asynchronous behavior and follow trajectories was first developed. Then methods for profile generation and joint level velocity regulation were programmed. Position control was controlled via a 1 tick dead-band while velocity control was performed by maintaining a trapezoidal profile. The following sub-sections will delve into the planning and code used to implement these robot controllers.

Motion Planning Design and Pipeline

The motion planning and trajectory following pipeline is illustrated in the flow chart shown in Figure 12.

The procedure begins when the user changes the robot mode from the default "Manual" to "Trajectory" in the Web UI [1]. The trajectory list is a class member of the abstract Robot class and is populated in the ThreadIGT() Method in the OpenIGTLink Class. The trajectory list is a 2D C++ array in which each row represents a new trajectory and each column represents the time, desired position, and desired velocity for each axis in joint space. If the trajectory list is empty, that means no paths were sent by OpenIGTLink, thus the result is reported to the user and the robot mode is reset to "Manual". If the trajectory list is populated, then the trajectory index, the position in the list used for maintaining asynchrony in the code, is checked. If the trajectory index is greater than the length of the list, then the trajectory is complete, thus the result is reported to the user and the robot mode is reset to "Manual". At this point the trajectory list is also emptied in order to

[1]See Section 5.3.3 for more information of changing robot modes

prepare for new paths.

If there are trajectories remaining to follow, then the code obtains the 1D array at trajectory index from the trajectory list. This 1D array holds position and velocity information for each Motor object in the Robot class. The array is looped through and the position setpoints are set. The code checks if all Motors are within 1 tick of their setpoint. If they are, the trajectory index is incremented so the next 1D trajectory in the trajectory list is used.

If a Motor object has not reached its setpoint, then the setpoint and the Current Position are passed into another function, FollowTrapezoidalVelocity(), in the Motor class that generates a trapezoidal velocity profile. This function outputs a desired velocity and the current velocity of the motor. These serve as inputs to a joint level velocity controller that aims to regulate the motor speed. The velocity controller returns a motor signal which is then custom tuned for different motor types. Once safeties have been checked the value is finally used to set the Motor output in order to control velocity of the axis in the SetMotorOutputSignal() function.

The functionality described takes place in the FollowTrajectory() method shown in Listing 10. At this point the whole process is repeated taking approximately 100us. The trajectory following code was designed specifically to work asynchronously and not block Main Loop functionality which may include other software commands such as enabling the motors or homing an axis.

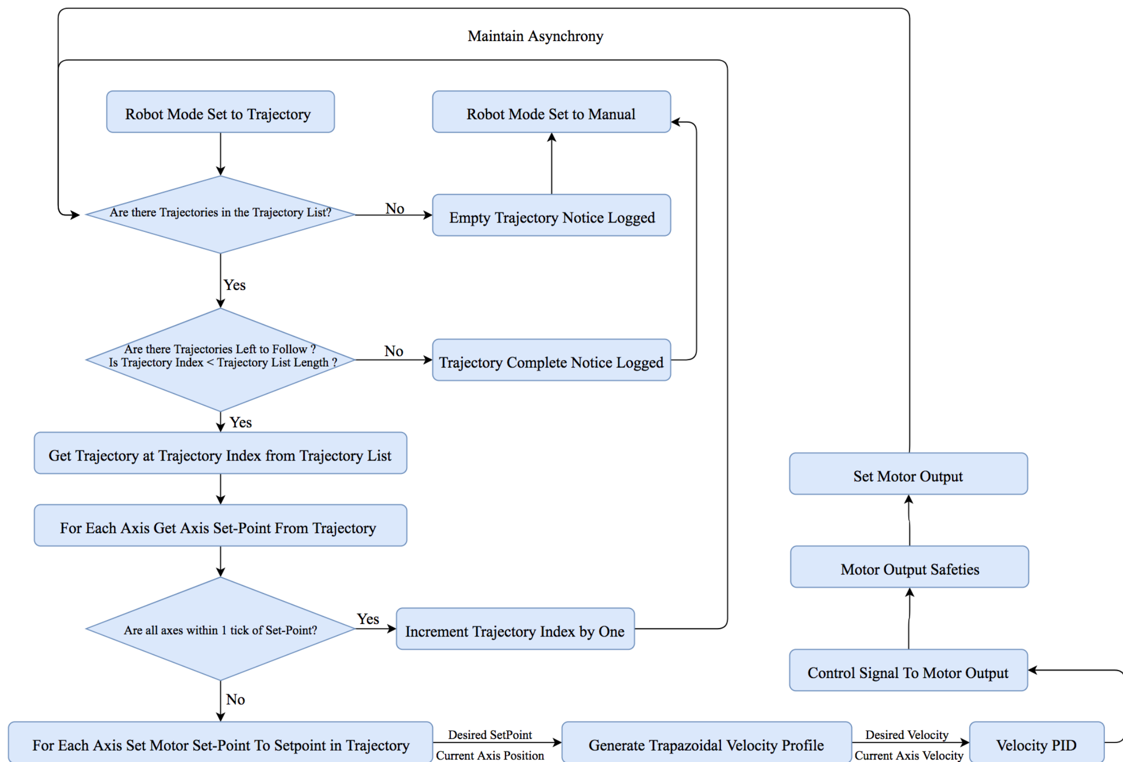


Figure 12: The figure shows a flow chart of the trajectory motion planning pipeline.

Listing 10: This listing shows the high-level robot trajectory following method that is responsible for commanding motor axes.

```
// NeuroRobot specific method for following a given trajectory
void NeuroRobot::FollowTrajectory(){

   // TODO: Reach set points in desired amount of time
   // TODO: Perform transformations and unit conversions to incoming trajectory as needed

   // If there are setpoints to follow in the trajectory
   Logger& log = Logger::GetInstance();

   // If we haven't reached the end of the trajectory
   if(_trajectoryIndex < _trajectory.size()){
      // If the current position is NOT the desired position, then travel to the desired positions
      if(
!(_yawRotation.GetEncoderPositionTicks() >= _trajectory[_trajectoryIndex][1] -
     _yawRotation._deadband && _yawRotation.GetEncoderPositionTicks() <=
     _trajectory[_trajectoryIndex][1] + _yawRotation._deadband) ||
!(_probeRotation.GetEncoderPositionTicks() >= _trajectory[_trajectoryIndex][2] -
     _probeRotation._deadband ....)
      ) {
         // Motor Set points are set to their current trajectory values
         _yawRotation._setpoint = _trajectory[_trajectoryIndex][1];
         _probeRotation._setpoint = _trajectory[_trajectoryIndex][2];
         ...

         // Motor Velocity set points
         double desiredPositionTicks = _trajectory[_trajectoryIndex][1];
         double desiredVelTicksPerSecond = _trajectory[_trajectoryIndex][8];
//       _yawRotation.BangBang(desiredVelTicksPerSecond);


         if(_trajectoryIndex == 0){
            _yawRotation._velocity = _yawRotation.FollowTrapezoidalTrajectory(0,
                 desiredPositionTicks, desiredVelTicksPerSecond);
         } else {
            _yawRotation._velocity =
                 _yawRotation.FollowTrapezoidalTrajectory(_trajectory[_trajectoryIndex-1][1],
                 desiredPositionTicks, desiredVelTicksPerSecond);
         }

      }

      // If the current position is the desired position, then increment the trajectory index
      else {
         _trajectoryIndex += 1;
      }
   } else {
      log.Log("Trajectory Complete !", LOG_LEVEL_INFO, true);
      _yawRotation._velocity =
           _yawRotation.FollowTrapezoidalTrajectory(_trajectory[_trajectoryIndex-1][1],
           _trajectory[_trajectoryIndex-1][1], 0);
//    _mode = "Manual"; // Lets go back to manual mode on completion
   }
}
```

Trapezoidal Velocity Profile Generation

The Motor Class method FollowTrapezoidalTrajectory shown in Listing 11 is used to generate a trapezoidal velocity profile.

The method receives a desired velocity, Vdesired, the current position, Pcurrent, of a Motor Object as reported by the encoder calculation, and the start, Po, and end, Pfinal, position of the trajectory. This information is used to generate a trapezoidal profile as shown in Figure 13.

There are three stages to a trapezoidal profile: ramp up to reach the desired velocity, maintaining a constant desired velocity, and ramp down to slow down as the axis approaches the desired position. The current position of the motor determines what stage the motor is in and what velocity setpoint should be returned. The steepness of the trapezoid is the acceleration and deceleration of the ramp up and down stages in the profile, respectively. The determined velocity setpoint is used as the desired motor velocity in a joint level velocity control loop.

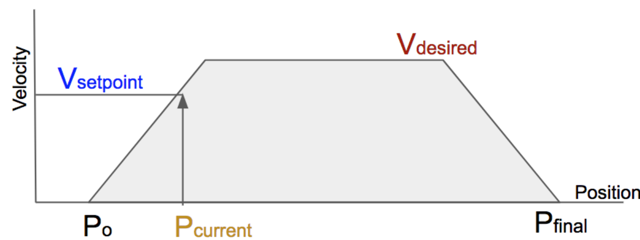


Figure 13: The figure shows an example of how the trapezoidal velocity method generates a velocity setpoint.

Listing 11: The listing shows the method that controls trapezoidal velocity profile.

```
int Motor::FollowTrapezoidalTrajectory(double positionStartPointTicks, double
     positionSetPointTicks, double velocitySetPointTicksPerSecond) {
   // Set Motor Setpoint
   _setpoint = positionSetPointTicks;

   // Get the current position of the motor in ticks
   int currentPositionTicks = GetEncoderPositionTicks();

   //variables for creating trap profile
   int totalTrajectoryLength = fabs(positionSetPointTicks - positionStartPointTicks);
   int traveledTrajectoryLength = fabs(currentPositionTicks - positionStartPointTicks);

   // created trapezoidal trajectory profile
   double rampUp = 0.3;
   double rampDown = 1 - rampUp;
   if (traveledTrajectoryLength < rampUp*totalTrajectoryLength) {
      double acceleratingVelocityRatio = (traveledTrajectoryLength)/(rampUp*totalTrajectoryLength);
      if(acceleratingVelocityRatio < 0.05){
         velocitySetPointTicksPerSecond = (velocitySetPointTicksPerSecond*0.05);
```

```
        }else {
          velocitySetPointTicksPerSecond = acceleratingVelocityRatio*velocitySetPointTicksPerSecond;
        }
    } else if (traveledTrajectoryLength >= rampUp*totalTrajectoryLength && traveledTrajectoryLength
          <= rampDown*totalTrajectoryLength) {
        double constantVelocityRatio = 1;
        velocitySetPointTicksPerSecond = velocitySetPointTicksPerSecond*constantVelocityRatio;

    } else if (traveledTrajectoryLength > rampDown*totalTrajectoryLength && traveledTrajectoryLength
          <= totalTrajectoryLength) {
        double deceleratingVelocityRatio = abs(totalTrajectoryLength -
             traveledTrajectoryLength)/(rampUp*totalTrajectoryLength);
        velocitySetPointTicksPerSecond = velocitySetPointTicksPerSecond*deceleratingVelocityRatio;

    } else {
        StopMotor();
        return true;
    }

    double desiredPositionUnits = positionSetPointTicks / _ticksPerUnit;
    double desiredVelocityUnits = velocitySetPointTicksPerSecond / _ticksPerUnit;

    // use controller to determine pwm value
    double motor_control_cmd = PID(desiredPositionUnits, desiredVelocityUnits);
    double motor_signal_velocity = ControlCommandToMotorOutputSignal(motor_control_cmd);

    return motor_signal_velocity;
}
```

Joint Level Velocity Controller

The joint level velocity controller method is used to maintain a desired velocity for the time step through the control loop for a given motor axis and is shown in Listing 12. It receives a desired velocity in units and returns a motor command. The gains used in the controller are custom to each axis allowing for different types of motors to be tuned individually. The integral and derivative errors are stored as class members to preserve their values as the trajectory occurs and to maintain asynchronous behavior.

The VelocityController() method returns a motor control command. This motor control command is used as an input to the ControlCommandToMotorOutputSignal() method in the Motor class shown in Figure 13. This method is used to scale and offset the output from the velocity controller to the range used for a given motor type. This allows the controller to be used generically with different types of motors and different motor outputs for maintaining a velocity profile. The developer then simply needs to add another case to the switch statement when a new motor type is used.

Once a motor signal has been calculated from the motor control command from the controller, the value is safety checked against the maximum allowed velocity for

that axis. If it is greater than this value, the velocity is capped.

Listing 12: The listing shows the Joint Level Velocity Controller used to regulate robot motor output. Note that the gains are custom per axis.

```
int Motor::VelocityController(double desiredPositionUnits, double desiredVelocityUnits){

   // TODO: Replace these weights with the robot specific weights
   double KP = 1;
   double KI = 1;
   double KD = -5;

   // TODO: Figure out why current velocity units is zero
   Logger& log = Logger::GetInstance();
   long double currentVelocityUnits = (_encoder._vel/_ticksPerUnit);
   double currentPositionUnits = GetEncoderPositionUnit();

   double positionErrorUnits = fabs(desiredPositionUnits - currentPositionUnits);
   double velocityErrorUnits = KP*-170*positionErrorUnits - currentVelocityUnits;

   _integral = _integral+(KI*-10)*velocityErrorUnits;

   long double outputPIV = fabs(_integral - KD*currentVelocityUnits);

   log.Log("POS: " + to_string(GetEncoderPositionTicks()), LOG_LEVEL_INFO, false);
   log.Log("VELOCITY: " + to_string(desiredVelocityUnits) + " " + to_string(outputPIV) + " " +
       to_string(currentVelocityUnits), LOG_LEVEL_INFO, false);

   return outputPIV;
}
```

Listing 13: The listing shows the method that takes a control command that is custom generated from a velocity controller output signal into a motor output that is motor type specific.

```
double Motor::ControlCommandToMotorOutputSignal(int controlCommand){
   double motor_signal_velocity = _minVelocity;
   switch(_cardType){
      case card_type::externaldriver_pwm:
         motor_signal_velocity = controlCommand + _minVelocity;
         break;

      case card_type::externaldriver_non_pwm:
         break;

      case card_type::highfrequency:
         break;

      default: // Unspecified or unlisted card type
         break;
   }

   // Check Motor Safety's before returning values
   if(motor_signal_velocity > _maxVelocity){
      motor_signal_velocity = _maxVelocity;
   }
```

```
  if(motor_signal_velocity < _minVelocity){
    motor_signal_velocity = _minVelocity;
  }

  return motor_signal_velocity;
}
```

4.2.8 Other Robot Components & Utilities

Other Robot Components and Utilities include :

- **Force Sensor** - interprets raw and calibrated force sensor ADC readings, this is used in the Cooperative Control Case Study for the Prostate Robot [25]
- **Limit Switch** - checks if a limit switch has been triggered
- **Timer** - has two main methods tic() and toc() that are used for profiling sections of code
- **Packets** - handles shared data distributed to robot components via incoming SPI information packets
- **FPGA Utilities** - contains FPGA helper methods such as initialization and LED toggling
- **Logger** - singleton used for warning/error handling, creates/writes to a log file on the NI Module and sends log information to be displayed on the web-gui. Code exceptions are also recorded by this singleton.

4.3 Developer's Web Application

4.3.1 React Front End

ReactJS is an open source Javascript Library that allows developers to create declarative and flexible web UI's. The library is largely maintained by Facebook and provides speed and scalability by using a virtual DOM[2]. This allows React to only re-render sections of the DOM that have been modified, rather than re-rendering the whole page. It also uses JSX to allow for Javascript and HTML to coexist in a single file.

The Web UI can be accessed with a valid username and password on the login screen. The main page of the application is comprised of five sectors: The Targeting Panel, The Motor Panel, The Status Panel, The Communication Panel, and The

[2]Document Obect Model. This means the Web Page has objects that can be manipulated.

Logger Panel. Each panel has a little dot in the top right corner that becomes red when the UI is not connected to the C++ back-end server. All the ReactJS reducers, javascript code, HTML, CSS, and API for this front-end were developed in this thesis. The following paragraphs give a detailed high-level description of the function of each sector of the web application.

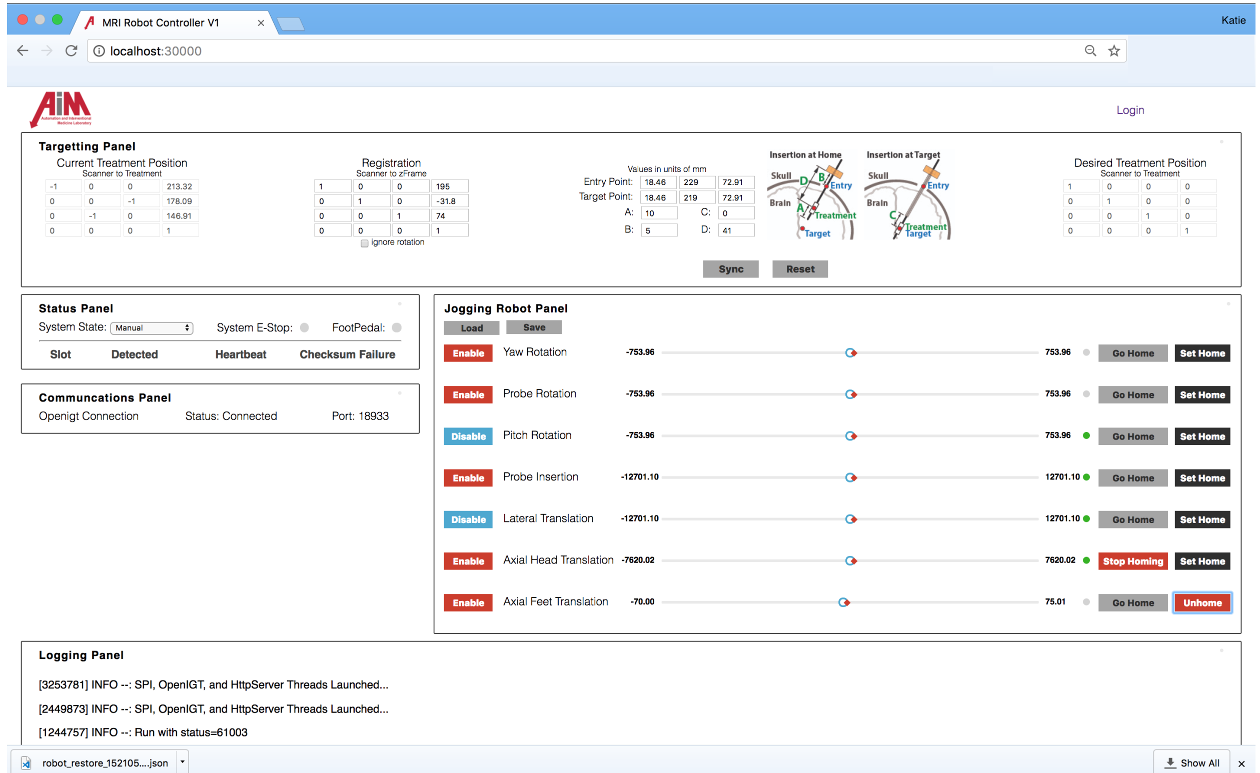


Figure 14: The figure shows the main page of the engineers console.

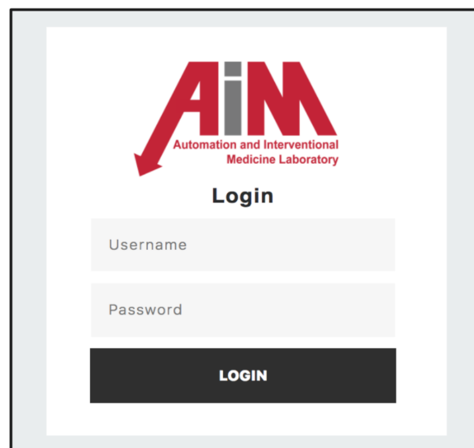


Figure 15: The figure shows the Login Page used to authenticate users for access to the Main Page of the UI.

The Targeting Panel

The targeting panel is located at the top of the UI and displays all data with respect to the scanner's coordinate frame. The left most 4x4 homogeneous transformation shows the current treatment location. To the right of this non-editable field is the 4x4 registration transform. This field is editable and can be populated either by hand or via OpenIGTLink. The targeting panel has a robot-generic input fields for the x,y,z entry and target points that are located above the A,B,C,D robot-probe specific input fields. These probe offsets are defined by the image shown in the targeting panel and represent the tip to treatment zone distance, the entry point to robot distance, the treatment to cannula distance, and the treatment to robot at the home configuration distance, respectively. These are unique to the robot and to the probe used. The final 4x4 transformation is the desired treatment position. When an entry and target point have been input, the inverse kinematics of the robot are called, and this transform is calculated. When the robot moves and reaches the desired target location, the current treatment transform and the desired treatment transform should match. To send values to the C++ backend, the sync button[3] should be clicked.

The Motor Panel

The motor panel[4] is located below the targetting panel and on the right side of the UI. It shows control options for each robot axis.

At the top of the panel are two buttons: load and save. The save button generates a file that records the current control configuration of all robot motors. When this generated file is dragged on top of the load button, all robot motors are updated to the state recorded in that saved file. This functionality was implemented to compensate for incorrect encoder counting during MRI scans, system reboots, or other such scenarios during a procedure. In order to not lose the robot position, the code allows the user to save and restore.

The motor panel is largely comprised of individual motor jog control options. Each row represents an individual motor axis on the robot. The enable button is a software trigger that prevents the motor from moving when disabled. The name of a robot axis is displayed in order to indicate which motor the options will manipulate. The slider in the middle of the row allows the user to jog the motor. This is performed by dragging the motor setpoint, represented by a blue circle, around the slider. The motor will then move its current positon, represented by a red diamond, towards the setpoint when both the axis is enabled and the footpedal is pressed. The

[3]This button only appears after an editable input field has been changed

[4]Also called the Jogging Robot Panel

numbers on the left and right side of the slider are the software limits. They are set to an arbitrarily large value when the motor has not been homed, and they change to the motor's configured software limits when the axis has been homed. The gray circle to the right of the slider represents the motor's limit switch. It turns green when the limit switch has been triggered and is gray otherwise. The "Go Home" button when clicked turns red and changes its text to "Homing..". Then, regardless of the blue setpoint in the slider, the axis will move towards its limit switch. The Set-Home Button is clicked to tell the program that an axis is already in the home configuration without having to re-run the homing procedure.

The Status Panel

The status panel shows information on the system and the Daughter Cards. The first row tells the user the System State, whether the E-Stop has been pulled, and whether the footpedal is pressed. The System E-Stop circle becomes green when the E-Stop has pulled and is gray other wise. The Footpedal circle becomes green when the Footpedal is pressed and is gray other wise. A list of Daughter Cards statuses are also shown in this panel. Each row shows the slot id of the card, if it is detected in that slot, if its heartbeat is still live, and the number of check sum failures experienced on the card.

The Communication Panel

The communication panel shows network connections. Each row shows the type of connection, whether a client has connected, and on what port. This is typically used to monitor OpenIGTLink clients.

The Logger Panel

The Logger Panel is located at the bottom of the UI and shows messages about the system. These messages can range from INFO, or nominal data, to SEVERE, or critical errors. The logger panel can tell the user if the calculated inverse kinematics are impossible for the system to reach, if a motor has stalled, or other system wide messages or exceptions.

4.3.2 C++ HTTP Server API

A C++ HTTP Server back-end was created to serve API requests from the ReactJS frontend. Three kinds of requests are used "GET", "POST", and "PUT". All data sent to and from the server are sent as JSON, the JavaScript Object Notation. C++ does not natively support creating HTTP Servers or JSON objects. The former was

made by using the SimpleServer C++ library [5] and the latter was accomplished by using the picojson [6] library.

The CustomWebServer class in the C++ software, is launched in a dedicated thread and it creates the HTTP Server. This class receives a pointer to an abstract Robot, a list of pointers to IGT objects, and a port to listen on. The Robot* parameter allows the Web UI to access and change fields within the main robot object used throughout the C++ program. The Web frontend can send a request to the HTTP Server to modify a specific robot field. For example, when the sync button on the UI is pressed in the targeting panel, a "PUT" request is made on the path "\registration" to the HTTP Server, see Listing 15. The C++ server interprets that as a request to change the registration transform in the Robot Object, and fulfills the request by unpacking the transform data sent as a JSON object. The same idea is applied when the front end asks to receive information about the robot, such as the configurations of each motor. An example of a GET request is shown in Listing 14. The Logger singleton in the C++ code records all JSON object transactions timestamped between the frontend and the backend.

Listing 14: This listing shows an example of a GET request in the C++ HTTP Server. GET requests are used when the ReactJS front end wants to receive information.

```
 // ****************************************************
// **** GET /api/targetting
server.resource["/api/targeting"]["GET"] = [&](shared_ptr<HttpServer::Response> response,
     shared_ptr<HttpServer::Request> request) {
Logger& log = Logger::GetInstance();
try {
      // Record requests to the log
      log.Log("Started GET '/api/targeting' for " + request->remote_endpoint_address(),
          LOG_LEVEL_DEBUG, false);

       // Put together motor data as JSON for processing
      string json = "{";
        json += "\"id\": 1,";
        json += "\"registration\":" +
             ParseEigenToString<Eigen::Matrix4d>(_robot->_registration) + " ,";
        json += "\"ignore_registration_rotation\": false,"; // TODO: Add card type here
        json += "\"entry_point\":" + ParseEigenToString<Eigen::Vector3d>(_robot->_entryPoint)+
             ",";
        json += "\"target_point\":" +
             ParseEigenToString<Eigen::Vector3d>(_robot->_targetPoint)+ ",";
        json += "\"treatment_to_tip_offset\":" + to_string(_robot->_probe._treatmentToTip) +
             ",";
        json += "\"canula_to_treatment_offset\":" +
             to_string(_robot->_probe._cannulaToTreatment) + ",";
        json += "\"robot_to_entry_offset\":" + to_string(_robot->_probe._robotToEntry) + ",";
        json += "\"treatment_to_robot_at_home\":" +
             to_string(_robot->_probe._robotToTreatmentAtHome)+ ",";
```

[5] Simple-Web-Server from https://github.com/eidheim/Simple-Web-Server
[6] PicoJSON from https://github.com/kazuho/picojson/

```
      json += "\"current_position\":" +
          ParseEigenToString<Eigen::Matrix4d>(_robot->_currentPose) + ",";
      json += "\"desired_position\":" +
          ParseEigenToString<Eigen::Matrix4d>(_robot->_targetPose);
    json += "}";


    SimpleWeb::CaseInsensitiveMultimap header;
    header.emplace("Access-Control-Allow-Origin", "*");

    response->write(SimpleWeb::StatusCode::success_ok, json, header);
  }
  catch(const exception &e) {
    response->write(SimpleWeb::StatusCode::server_error_internal_server_error, e.what());
    log.Log("Failed GET '/api/targeting' for " + request->remote_endpoint_address() + "
        error: " + e.what(), LOG_LEVEL_ERROR, false);
  }
};
```

Listing 15: This listing shows an example of a PUT request in the C++ HTTP Server. PUT requests are used when the ReactJS front end wants to change information in the robot code.

```
// ****************************************************
// **** PUT /api/targeting/registration
server.resource["/api/targeting/registration"]["PUT"] = [&](shared_ptr<HttpServer::Response>
    response, shared_ptr<HttpServer::Request> request) {
Logger& log = Logger::GetInstance();
try {
  // Get the request body
  picojson::value request_body;
  std::string err = picojson::parse(request_body, request->content);
  if (! err.empty()) {
    cerr << err << endl;
  }

  // Record requests to the log
  Logger& log = Logger::GetInstance();
  log.Log("Started PUT '/api/targeting/registration' for " +
      request->remote_endpoint_address(), LOG_LEVEL_DEBUG, false);

  picojson::value registrationVector = request_body.get("registration");
  _robot->_registration = (Eigen::Matrix4d() <<
      stof(registrationVector.get(0).to_str()),stof(registrationVector.get(1).to_str()),
      stof(registrationVector.get(2).to_str()), stof(registrationVector.get(3).to_str()),
                                  stof(registrationVector.get(4).to_str()),
                                      stof(registrationVector.get(5).to_str()),
                                      stof(registrationVector.get(6).to_str()),
                                      stof(registrationVector.get(7).to_str()),
                                  stof(registrationVector.get(8).to_str()),
                                      stof(registrationVector.get(9).to_str()),
                                      stof(registrationVector.get(10).to_str()),
                                      stof(registrationVector.get(11).to_str()),
                                  stof(registrationVector.get(12).to_str()),
                                      stof(registrationVector.get(13).to_str()),
                                      stof(registrationVector.get(14).to_str()),
                                      stof(registrationVector.get(15).to_str())).finished();
```

```
    string json = "{";
                json += "\"id\": 1,";
                json += "\"registration\":" +
                    ParseEigenToString<Eigen::Matrix4d>(_robot->_registration) + " ,";
                json += "\"ignore_registration_rotation\": false,"; // TODO: Add card type here
                json += "\"entry_point\":" +
                    ParseEigenToString<Eigen::Vector3d>(_robot->_entryPoint)+ ",";
                json += "\"target_point\":" +
                    ParseEigenToString<Eigen::Vector3d>(_robot->_targetPoint)+ ",";
                json += "\"treatment_to_tip_offset\":" +
                    to_string(_robot->_probe._treatmentToTip) + ",";
                json += "\"canula_to_treatment_offset\":" +
                    to_string(_robot->_probe._cannulaToTreatment) + ",";
                json += "\"robot_to_entry_offset\":" + to_string(_robot->_probe._robotToEntry)
                    + ",";
                json += "\"treatment_to_robot_at_home\":" +
                    to_string(_robot->_probe._robotToTreatmentAtHome)+ ",";
                json += "\"current_position\":" +
                    ParseEigenToString<Eigen::Matrix4d>(_robot->_currentPose) + ",";
                json += "\"desired_position\":" +
                    ParseEigenToString<Eigen::Matrix4d>(_robot->_targetPose);
            json += "}";

   SimpleWeb::CaseInsensitiveMultimap header;
   header.emplace("Access-Control-Allow-Origin", "*");

   response->write(SimpleWeb::StatusCode::success_ok, json, header);

 }
 catch(const exception &e) {
   response->write(SimpleWeb::StatusCode::server_error_internal_server_error, e.what());
   log.Log("Failed PUT '/api/targeting/registration' for " + request->remote_endpoint_address()
        + " error: " + e.what(), LOG_LEVEL_ERROR, false);
 }
 };
```

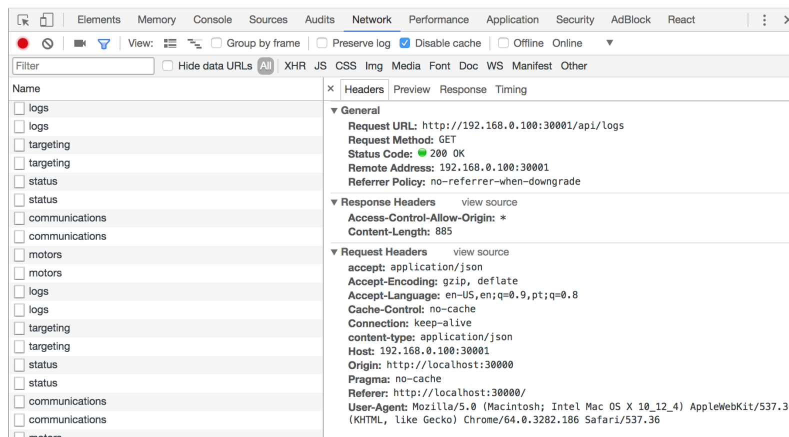


Figure 16: The figure shows API requests being made in Chrome's Web Development Console.

4.3.3 Robot Specific Functionality & State Machine

The two surgical robots at the WPI Aim Lab have different procedures they must be able to perform. The user can select between different robot specific function on the web application via a dropdown menu in the Status Panel.

The dropdown menu is populated via a "GET" request to the C++ HTTP Server. The server responds by returning a JSON object of the Robot classes list of states. The states are unique to each robot, for example, the Prostate Biopsy Robot can allow for cooperative control over the insertion access. When a selection is made on the web UI a "PUT" request is made to the C++ HTTP Server. This changes the current robot state.

The Update() method in the Robot class has a state machine that executes different procedures depending on the current robot state. The default state is "Manual" and allows the user to jog the robot via the sliders in the Motor Panel. Other states, however, can control the setpoints of different robot axes, thereby moving the "blue" circles in the sliders, and this may prevent the user from changing the setpoints by hand on the web-interface.

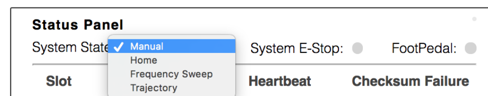


Figure 17: The figure shows the Drop Down Menu for Robot Specific Functionalities as used in the Update() methods state machine.

Listing 16: This listing shows the Update() method for the NeuroRobot. Note the switch case which takes in the robot state and the asynchronous nature of the function.

```
// This is the central NeuroRobot method -- everything happens in this method
// This method should never be blocking, it must be asynchronous
void NeuroRobot::Update(){
  // Update sensor and encoder readings via the SPI packets
  UpdateRobot(_loopRate);
  // If the Footpedal is pressed
  if(_fpga_util->IsFootPedalPressed()){
    if(_mode == "Manual"){
      //_timer.tic(); // SetPoints are given by the User via the Sliders
    } else if (_mode == "Home"){
      HomeRobot(); // SetPoints are given by the Home Method
    } else if (_mode == "Trajectory"){
      FollowTrajectory(); // SetPoints are given by the Trajectory
    } else if (_mode == "Frequency Sweep"){
      FrequencySweep(1); // Sweep Frequencies for a specific motor
    }
    // Move motors to their setpoints
    _yawRotation.MoveMotor();
    _probeRotation.MoveMotor();
    _pitchRotation.MoveMotor();
```

```
        _probeInsertion.MoveMotor();
        _lateralTranslation.MoveMotor();
        _axialHeadTranslation.MoveMotor();
        _axialFeetTranslation.MoveMotor();
    }
    // If the footpedal is not pressed
    else {
        // Update Motor Stall Detect
        _yawRotation.UpdateMotorStallTime();
        _probeRotation.UpdateMotorStallTime();
        _pitchRotation.UpdateMotorStallTime();
        _probeInsertion.UpdateMotorStallTime();
        _lateralTranslation.UpdateMotorStallTime();
        _axialHeadTranslation.UpdateMotorStallTime();
        _axialFeetTranslation.UpdateMotorStallTime();

        _probeInsertion._encoder._timer.tic();
        _yawRotation._encoder._timer.tic();
        // Stop all the motors of the robot
        StopRobot();
    }
    CheckForStalls();
    // As the current position of the robot changes, update _currentPose transform
    Eigen::Matrix4d fk = _neuroKinematics.ForwardKinematics(
_axialHeadTranslation.GetEncoderPositionUnit(), _axialFeetTranslation.GetEncoderPositionUnit(),
            _lateralTranslation.GetEncoderPositionUnit(), _probeInsertion.GetEncoderPositionUnit(),
            _probeRotation.GetEncoderPositionUnit(), _pitchRotation.GetEncoderPositionUnit(),
                _yawRotation.GetEncoderPositionUnit()).zFrameToTreatment;

    _currentPose = _registration * fk ;
}
```

4.4 A Generic Software Architecture

The code developed in this thesis research was constructed with the intent of being able to generically control different MRI-Compatible Robotic systems. The software follows an object oriented design structure that allows new robot objects to be created based off of a parent Abstract Robot Class. Robot objects can be constructed with the ability to configure the code to the desired use case and class members such as Motor objects can be instantiated and customized as needed. The generic motor class methods such as SetMotorOutputSignal() and ControlCommandToMotorOutputSignal() can vary functionality by motor type. The motor configurations also allow for customization of axis names, position limits, velocity boundaries, control gains, and others. This allows the Web UI to display a unique setup for the robot being used by receiving generic API requests. Robot specific functionality can be executed on the Web Application by setting robot "modes" in a dropdown menu that triggers a case in the state machine in the robot main loop. While the code will continue to grow, the steps taken to establish a modular, extensible, and generic software architecture should facilitate development of the code for any new robotic system.

5 Experimentation

In this section validations for the developed software architecture are presented.

5.1 Validation: Real-Time Loop Rates

Loop times were validated by profiling the code using the Timer classes tic() and toc() functions. First inspection showed a substantial amount of jitter in the system, see Figures 18 and 19, with spikes tripling the desired loop rate of 700us for the SPI thread and 1000us for the main thread.

The cause of the jitter was context switching between different processes in the operating system as determined by the scheduler. The operating system is able to switch between multiple processes very quickly making it seem as though they are running simultaneously. This is called multitasking. Parallelism is when a process runs on different cores in the system. The sbRIO-9651 Arm Cortex has a two core processor and Linux POSIX threads allow their core to be explicitly set.

There are two main types of schedulers that the NI Module operating system can use: first in first out (FIFO) and round-robin (RR). The former only runs threads in descending order by priority and will not give processor time to threads of equal or lower priority until the first has finished. The latter guarantees that threads of the same priority will each run for the same amount of time, in a cyclic manner. The time devoted to each thread in a round robin scheme can be set in software. The switching between threads is called a context switch. Experimentation with thread allocation, core distribution, and thread priority revealed that this round-robin cycling was causing consistent jitter in the system.

The jitter was fixed by consolidating the SPI Thread into the Main Robot Thread. This initially lead to an increase in the main loop rate to 1700us. However, this was largely fixed by optimizing the compilation of the matrix multiplication library, Eigen, by applying the flag "-Ofast". These two changes lead to the most tremendous impacts toward jitter and loop rate performance.

Additional optimizations included increasing the priority of the main thread and dedicating one core of the CPU exclusively to this loop. The other core would be used to run the OpenIGTLink and HTTP Server threads. With these modifications the loop can run as fast as 800us and improved performance with significantly less jitter at 1000us is shown in Figure 20 and 21.

Another improvement made to the loop times was to toggle transact all Daughter Cards at once. The SPI functionality, at its fastest, originally ran at 1500us. By transacting all daughter cards at once made possible by a dedicated bus, the loop time improved to 700us. This was due to the fact that enabling all cards at once is faster, than doing each card on an individual basis. The transact time independently

takes 400us and currently the code is blocked when waiting for it to complete.

The code was tested under a number of different conditions to validate the robustness of the applied optimizations. The robot motors were moved, data was sent through OpenIGTLink, requests were made to and from the Web Application and other functionality were executed with little to no impact on the desired loop rate. The code was also left to run for over an hour and the number of instances of each recorded loop rate was plotted on a histogram. The desired loop rate of 1000us occurred significantly more frequently than any of the deviated loop rates. This result is illustrated in Figure 22 where Figure 23 shows a zoomed in version.

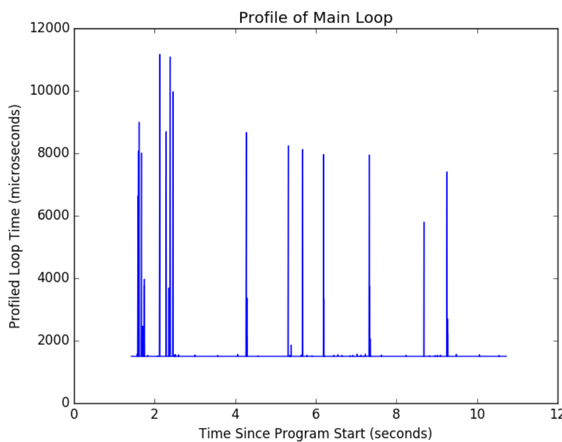


Figure 18: The figure shows the loop profile of the SPI thread before the fix to the jitter was applied.

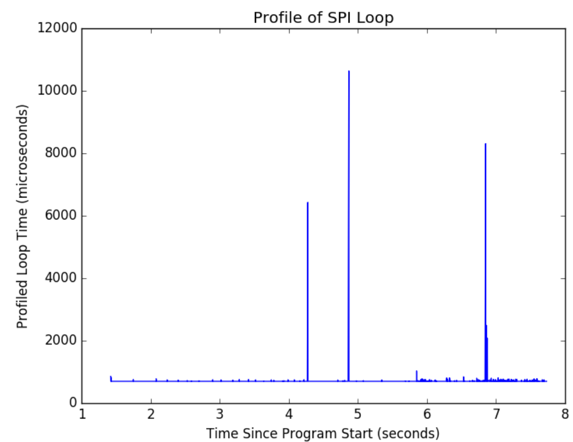


Figure 19: The figure shows the loop profile of the Main thread before the fix to the jitter was applied.

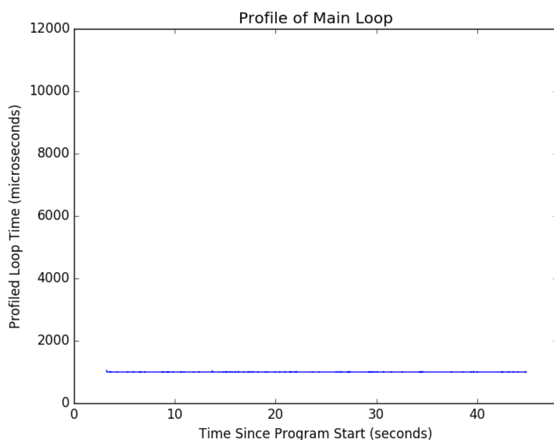


Figure 20: The figure shows the loop profile of the consolidated Main Thread after the fix to the jitter was applied for a desired loop rate of 1000us

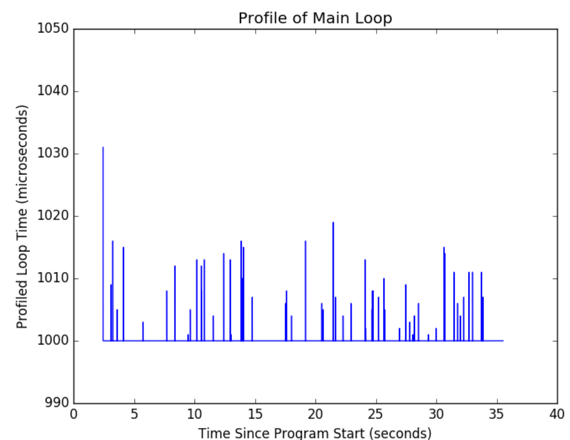


Figure 21: The figure shows the loop profile of the consolidated Main Thread after the fix to the jitter was applied. Note the scaled y-axis to demonstrate a minor amount of jitter still exists in the system.

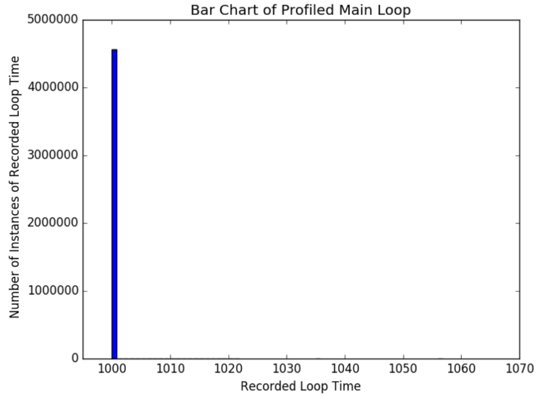


Figure 22: The figure shows the results from profiling the code for an hour.

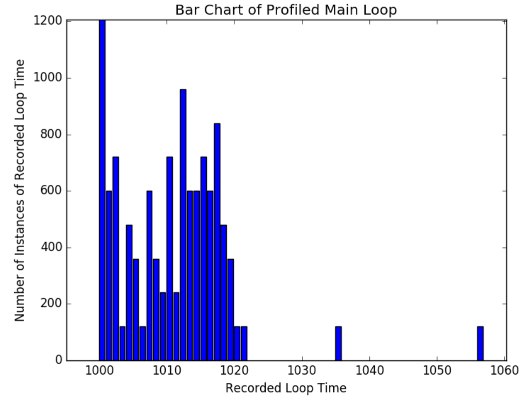


Figure 23: The figure shows the results from profiling the code for an hour, zoomed in to illustrate the other loop rates.

5.2 Validation: Robot Targeting

Robot targeting was tested independently for the Neurosurgery and Prostate Biopsy Robots. The NeuroRobot targeting abilities were validated by using the MRI Scanner. An AAPM phantom filled with distilled water was used, see Figure 24. The robot was placed in the bore of the MRI with the phantom, and an initial 3D FFE scan was taken. The DICOM images from this second scan were imported to Slicer, an entry/target point was selected, and the values were inputted in the respective fields on the web application. The robot was then moved to the desired target position, and another 3D FFE scan was taken. The DICOM images from this scan were imported to Slicer and the actual location of the probe was compared with the desired position, see Figures 25 and 26. This was repeated 7 times and the data is shown in Table 1. The results showed that there was an average RAS[7] error of 1.975mm, 0mm, 1.236mm and an overall average targeting error of 2.37mm.

Table 1: Targeting Results for NeuroAblation Robot

Trial No.	Desired Target (RAS) (mm)	Actual Probe Tip (RAS) (mm)	Error (RAS) (mm)	RMS Error
Trial 1	-3.523, 61.989, 11.74	-5.618, 61.989, 12.378	2.096, 0, -0.638	2.19mm
Trial 2	7.336, 61.989, 11.74	4.7, 61.989, 12.221	2.636, 0, -0.481	2.68mm
Trial 3	-13.954, 61.989, -10.05	-16.43, 61.989, -8.107	2.476, 0, 1.943	3.14mm
Trial 4	-14.726, 61.989, 33.837	-14.32, 61.989, 35.728	-0.366, 0, -1.891	1.93mm
Trial 5	-26.786, 62.989, 19.032	-28.664, 62.989, 20.691	1.878, 0, -1.659	2.51mm
Trial 6	-7.085, 61.989, -10.516	-4.617, 61.989, -12.061	2.468, 0, 1.545	2.19mm
Trial 7	-17.752, 61.989, 9.283	-15.844, 61.989, 8.786	1.908, 0, 0.497	1.97mm

[7] MRI Coordinate Frame – Right, Anterior, Superior

Figure 24: The figure shows the AAPM Phantom used during the NeuroRobot Targeting Tests.

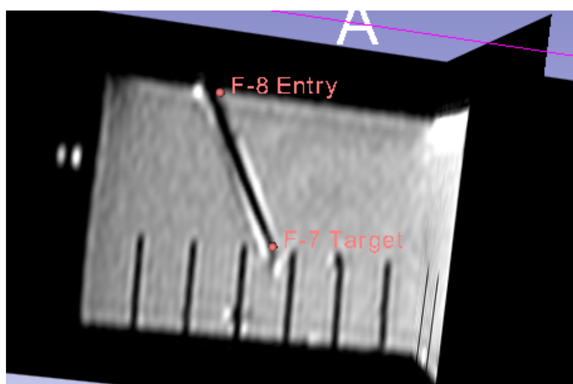


Figure 25: The figure shows the MRI scans of the probe after targeting the entry and target point shown.

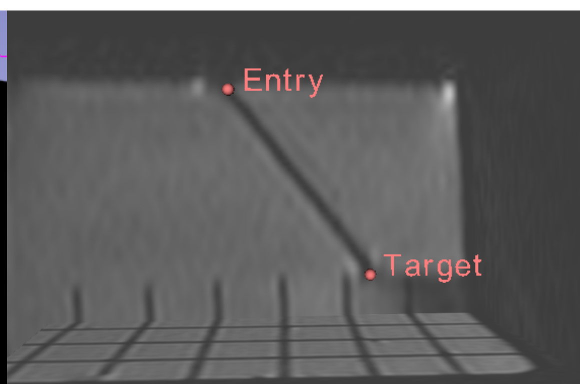


Figure 26: Another example of a NeuroRobot MRI target scan.

Figure 27: The figure shows the experimental setup used to validate the NeuroRobot targeting procedure in the MRI at UMass.

The Prostate Robot targeting abilities were validated with the Motion Capture System in the lab. The web application allows the user to select and set a target. Once chosen, the robot-base moves in the X,Y, direction to line up with this selected

target, and moves the insertion axis to the desired depth. An experiment was conducted with the robot to reach multiple particular targets to assess kinematic accuracy. To avoid needle deflection the tests were performed in air rather than in gelatin phantoms. A fiducial placed on the needle insertion shaft was tracked by the Motion Capture System and its final location was compared with the desired location set by the user. The process was repeated seven times and the results are shown in Table 2. The data shows an average X,Y,Z error of 0.422mm, 0.272mm, 0.632mm and an average overall in plane error of about 0.923 mm.

Table 2: Targeting Results for Prostate Robot

Trial No.	**Desired Target (X,Y,Z) (mm)**	**Actual Needle Tip (X,Y,Z) (mm)**	**Error (X,Y,Z) (mm)**	**RMS Error**
Trial 1	141.0, 0.0, 132.5	141.113, 0.012, 133.221	0.113, 0.012, 0.721	0.730 mm
Trial 2	150.0, 5.0, 132.5	150.671, 5.022, 131.191	0.671, 0.022, -1.309	1.471 mm
Trial 3	130.0, -5.0, 152.5	130.523, -5.108, 151.203	0.523, -0.108, -1.297	1.403 mm
Trial 4	120.0, 0.0, 112.5	120.455, 1.092, 112.616	0.455, 1.092, 0.616	1.334 mm
Trial 5	145.0, 15.0, 122.5	145.431, 14.567, 122.337	0.431, -0.433, -0.304	0.632 mm
Trial 6	147.0, -2.0, 152.5	147.466, -2.119, 152.196	0.466, -0.119, -0.304	0.569 mm
Trial 7	133.0, -2.0, 252.0	132.702, -2.119, 252.011	0.298, -0.119, 0.011	0.321 mm

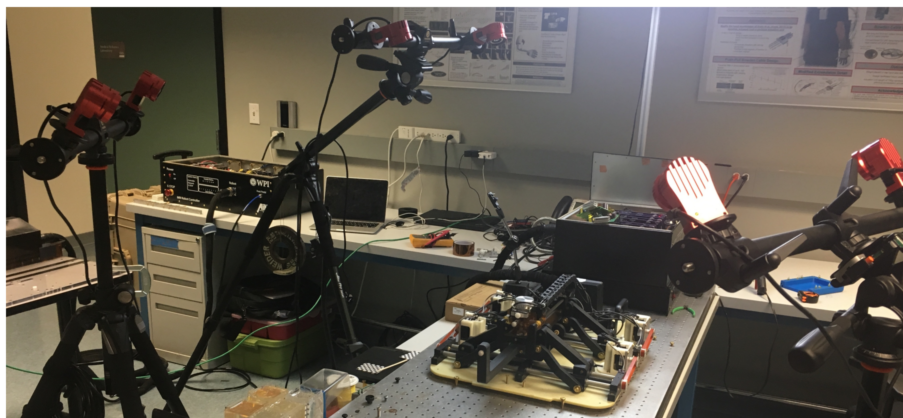


Figure 28: The figure shows another angle of the experimental setup used to validate the Prostate targeting procedure.

The results presented prove the targeting capabilities of the software architecture implemented in this thesis. The kinematic solutions for both the NeuroRobot and ProstateRobot are shown to be fully functional, however, angulation kinematics should be implemented to augment orthogonal targeting on the biopsy system. The results for the NeuroRobot show a systematic error or 2mm that can be attributed to inconsistencies in the one registration transform calculated for the experiment set. The performance, however, indicates that the treatment zone can be placed accurately with a certain degree of error, demonstrating key treatment capabilities

during MRI experimentation and pointing towards the completeness of the developed software architecture.

5.3 Validation: Robot Controllers

The robot trajectory following code and position/velocity controllers were tested in the lab using the Logger singleton object to record velocities and positions throughout the course of executing a set of paths.

Validations were performed on one axis of the NeuroRobot, the yaw motor, which is controlled via pwm values. A simple trajectory from 0 to 600 ticks was chosen. The results of the dead-band position controller are shown in Figure 30 and demonstrates that the motor is able to reach the desired position within at least 1 tick or 0.07 of a degree. This is due to the short ramp-up and down times of the piezo-electric motors used on this robotic system. As a result, a more sophisticated position controller is not needed to satisfy the current application for this research, however, the code is modular enough that a new controller could be easily implemented and interchanged. This would require either integrating the new position controller in the Robot TrajectoryFollowing() method or programming a new robot mode.

The results of the velocity validations are shown in Figure 29. It shows the desired trapezoidal trajectory for the motor to follow and the actual velocity recorded at each instant. The plot shows that the motor velocity follows a trapezoidal profile, however, there is a lot of noise in the signal. This is attributed to noise in collecting the actual velocity of the axis from the encoders. This is further made clear when comparing the velocity estimate in Figure 29 and the corresponding control output in Figure 31. The actual velocity calculation for an axis currently occurs in the C++ encoder class. The jitter arises due to the fact that the C++ code runs at 1000us, whereas the encoder counts every 1200us at the expected count rate for the yaw axis at the selected constant velocity. This can lead to a discrepancy in the time value used to calculate the current velocity in the C++ code.

This can be fixed in a number of different ways. One solution would be to use an encoder that can count much faster than the C++ code; with more ticks recorded, the time discrepancy can be reduced. Another solution would be to move the location of the encoder. The yaw axis is currently geared down on the NeuroRobot. Moving the encoder to a more central location can increase the number of recorded ticks. Finally, a critical contribution to fixing the problem would be to calculate velocity on the even lower level FPGA code rather than in the C++ software. The C++ code can then use this information and decide which estimate calculation to use as the current velocity of the axis.

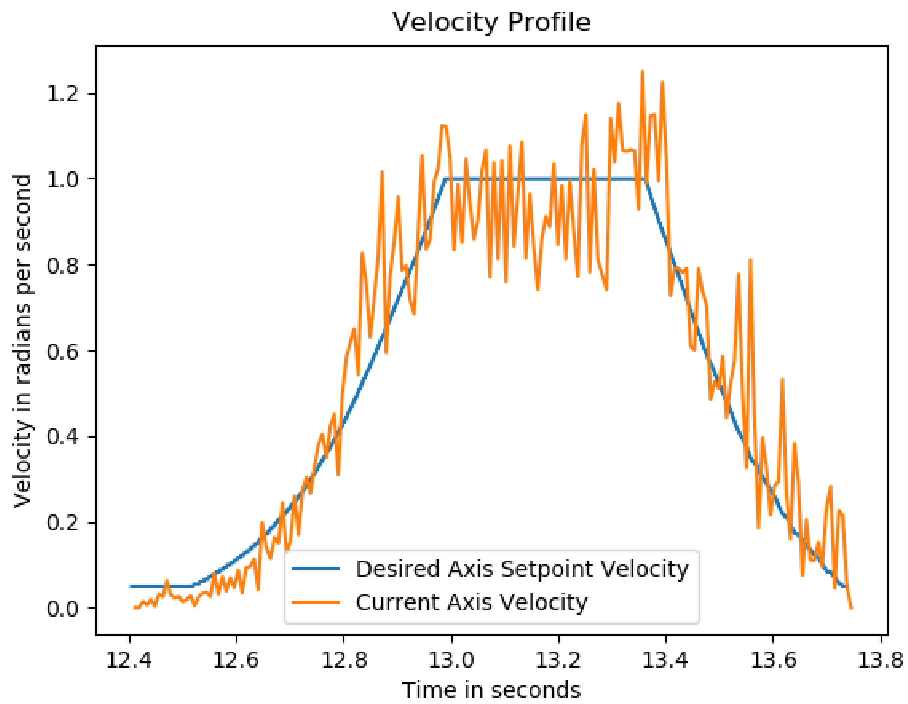


Figure 29: The figure shows the plot of the desired velocity profile and the actual velocity of the motor.

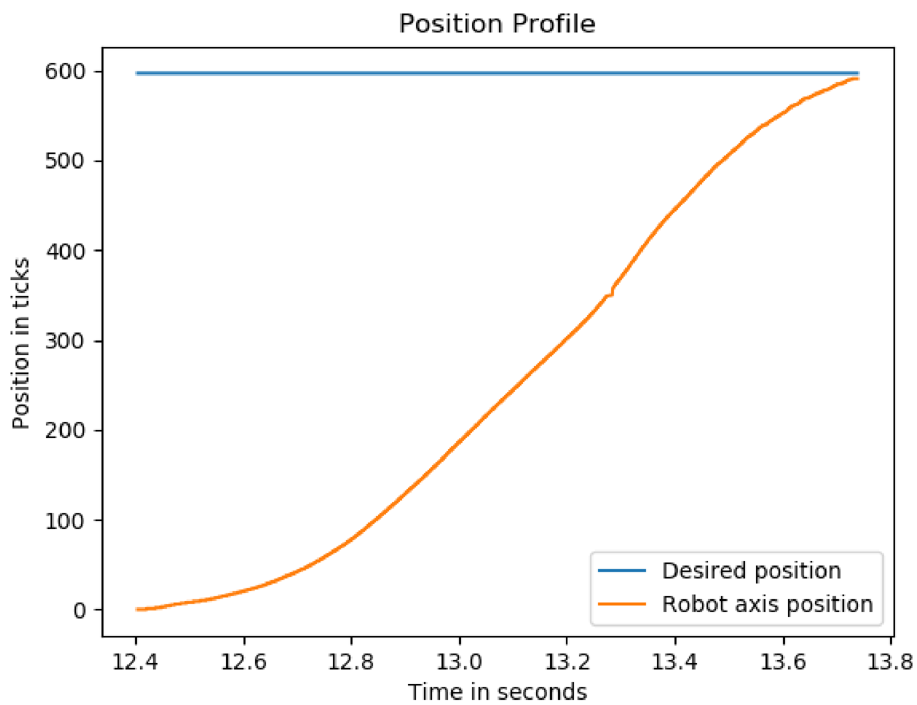


Figure 30: The figure shows the plot of the desired position and the position profile of the motor.

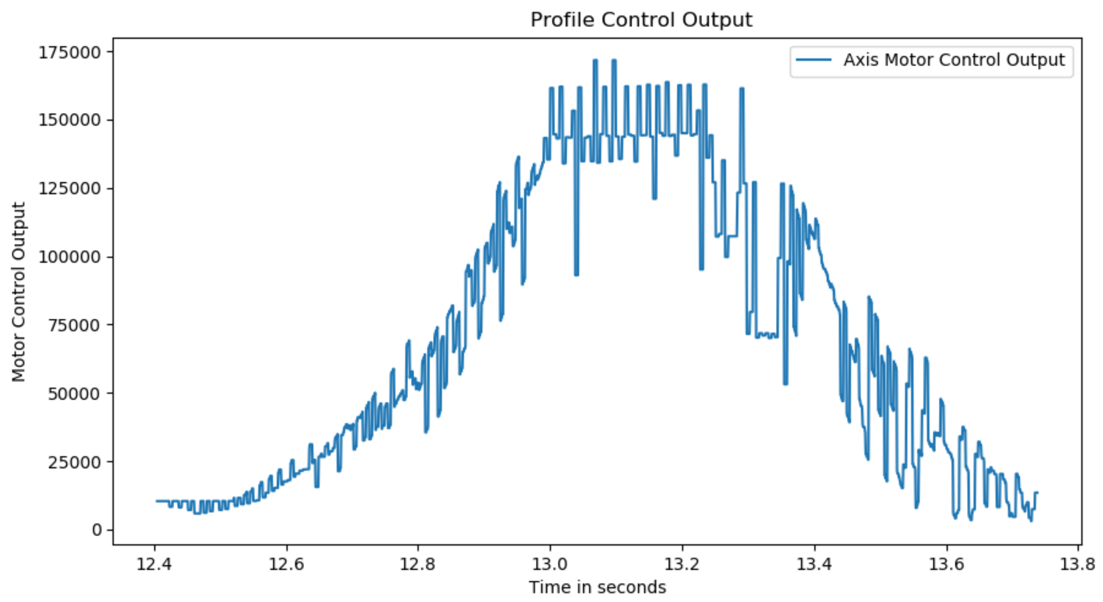


Figure 31: The figure shows the plot of the control output for the motor.

5.4 Validation: NeuroRobot Surgical Workflow

The C++ low level software was validated in the NeuroRobot surgical pipeline during animal trials at the UMass Memorial Hospital in Worcester, see the flow chart in Figure 33.

The procedure begins by homing the robot and placing the system in the MRI scanner. Then registration is performed. The animal is placed in the bore with the robot, and a 3D FFE scan of the brain is taken. The Neuro-Surgeon selects an entry/target point in Slicer on the DICOM images of the scan. The points are entered in or sent via OpenIGTLink to the web-application and the C++ program moves the robot motor set-points to target this vector. The robot is moved to line up with the entry burr hole, and the neurosurgeon inserts the ablation probe into the brain. A 3D FFE scan is taken to assess the location of the probe. The MRTI software is prepared and an ablation is performed with the Theravision system. After the procedure is complete, the probe is retracted from the brain, using the Web UI, and the robot is re-oriented such that the animal can be removed from the scanner.

Figure 32: The figure shows an example of the code being used at the UMass Medical Hospital during an experimentation.

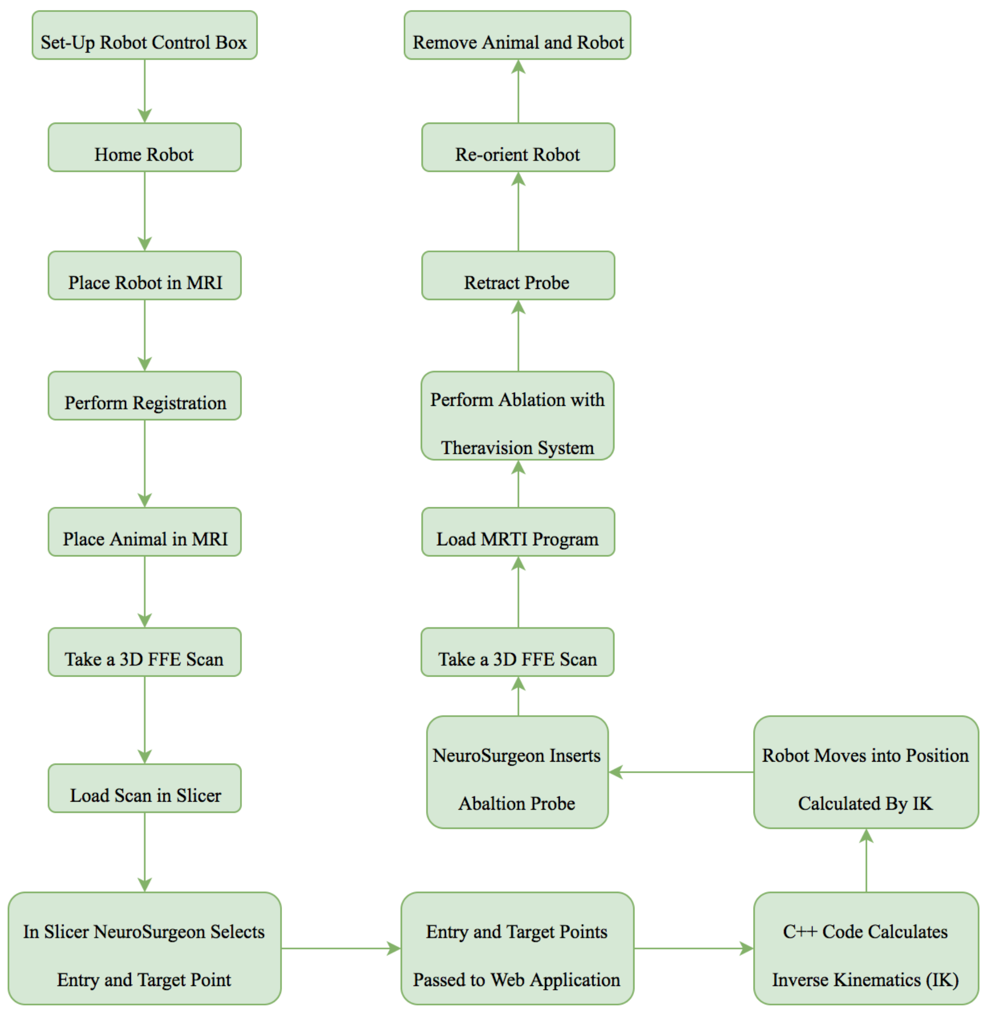


Figure 33: The figure shows a flow chart of the NeuroRobot Surgical Pipeline.

5.5 Validation: ProstateRobot Surgical Workflow

The C++ low level software was validated in the ProstateRobot surgical pipeline during experimentation at the Brigham and Women's Hospital AMIGO Research Suite in Boston, see the flow chart in Figure 35.

The experiment begins by homing the robot and placing the system in the MRI scanner. Then registration is performed. A gelatin phantom is placed in the bore, and a MRI scan is taken. The user selects a target in the phantom through Slicer and this value is entered in the web application. Then in the dropdown menu "Cooperative Biopsy" mode is selected. The robot base is moved to line up with the target. Then the user presses on the force sensor to insert the needle into the phantom. At this time active compensation and membrane detect are active. The former steers the needle towards the target to counter-act the effect of deflection and the latter causes a small vibration in the insertion axis when a membrane has been punctured by the needle. Once the target has been reached, the insertion axis is retracted and a new target can be selected.

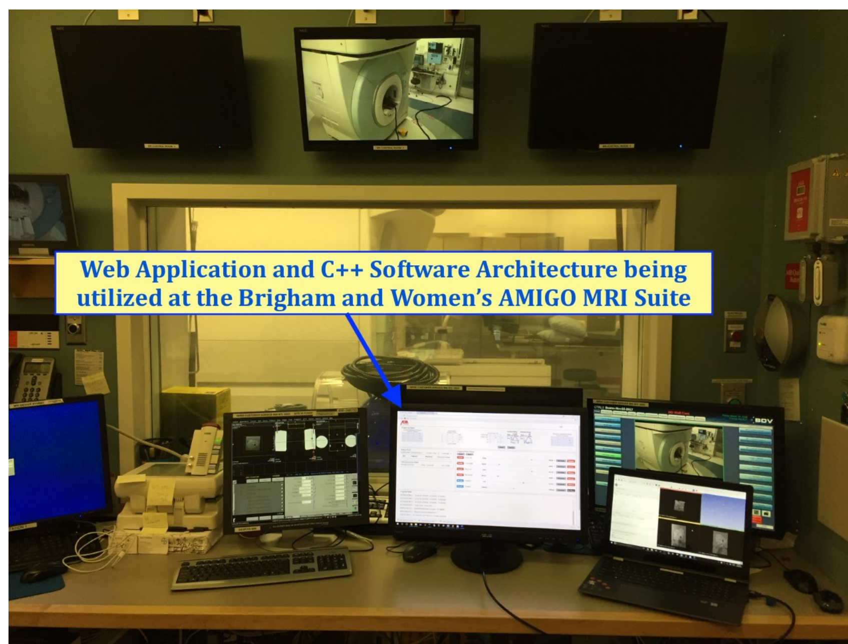


Figure 34: The figure shows the Web Application in the Scanner Console room at the Brigham and Women's AMIGO Research Suite.

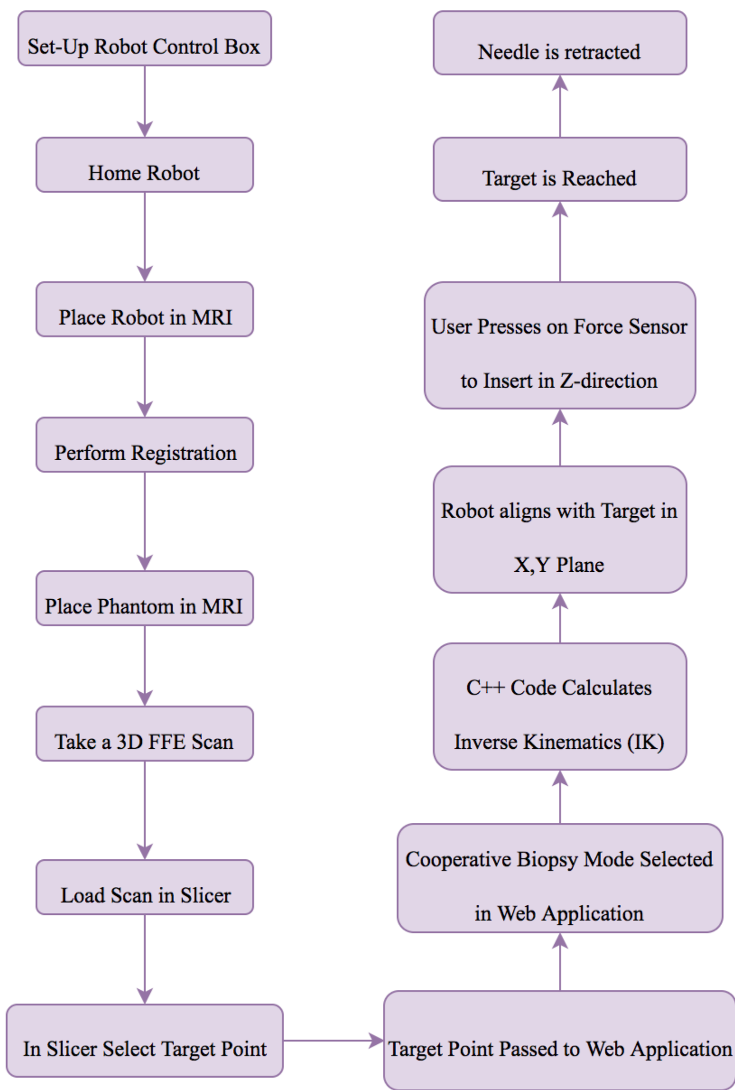


Figure 35: The figure shows a flow chart of the ProstateRobot Surgical Pipeline.

5.6 Validation: Software Tests & Code Documentation

A test suite and build environment were created to provide quantitative verifications of the C++ low level program functions developed in this thesis. The unit test suite was written using CUTE, a unit testing library for C++. The tests focused not only on the main use case, but also on boundary cases and the functional requirements of the software. Higher level "black" box and lower level "white" box tests where also written. In black box testing, the inner working components of a function are not considered, where as they are taken into account in white box testing. Cases ranged from safety functionality such as stall detect to validations of kinematic equations. Figure 17 shows an example of a unit test written to verify one specific function in the program. The test suite was run on the NI-module in the control box and results were reported in an exported XML file. An example of the output can be seen in Figure 36 and it shows all unit tests passing. Figure 37 shows an example of what the XML file output would look like when one test is failing. Tests can be added to the unit suite by future developers by creating validation methods to the UnitTest.cpp file following standard test driven development.

Listing 17: This listing shows an example of a unit test written to validate specific robot functions.

```
//================== NeuroRobot Tests ==================
void NeuroRobot_CheckForStalls(){
  // Initialize a NeuroRobot for Testing
   Packets _packets = Packets();
   FPGA_Utilities* _fpga_util = new FPGA_Utilities();
   NeuroRobot nr = NeuroRobot(&_packets, _fpga_util, 1000);

   // Check for Stalls
   bool isStall = nr.CheckForStalls();
   ASSERT_EQUAL(false, isStall);

   // Set Stall Condition
   nr._yawRotation._enabled = true;
   nr._yawRotation._setpoint = 100;
   nr._yawRotation._stall_detect_time = 0;
   nr._yawRotation.MoveMotor()
   isStall = nr.CheckForStalls();
   ASSERT_EQUAL(true, isStall);
}
```

Doxygen, a tool for creating code documentation for C++, was used to develop a written manuscript of the software in this thesis. According to the Doxygen website [8] "You can configure doxygen to extract the code structure from undocumented source files. This is very useful to quickly find your way in large source distributions. Doxygen can also visualize the relations between the various elements by means

[8]http://www.stack.nl/ dimitri/doxygen/

of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically." The biggest benefit of using Doxygen is its ability to generate documentation from source code. This means as the software is built upon the documentation can also be constructed simultaneous.

```
 1<testsuites>
 2    <testsuite name="Surgical_Robot_Test_Suite" tests="16">
 3        <testcase classname="Surgical_Robot_Test_Suite" name="NeuroRobot_CheckForStalls"/>
 4        <testcase classname="Surgical_Robot_Test_Suite" name="NeuroRobot_ValidateForwardKinematics"/>
 5        <testcase classname="Surgical_Robot_Test_Suite" name="NeuroRobot_RunInverseKinematics"/>
 6        <testcase classname="Surgical_Robot_Test_Suite" name="NeuroRobot_IsFootPedalPressed"/>
 7        <testcase classname="Surgical_Robot_Test_Suite" name="NeuroRobot_LisMotors"/>
 8        <testcase classname="Surgical_Robot_Test_Suite" name="NeuroRobot_GetMotor"/>
 9        <testcase classname="Surgical_Robot_Test_Suite" name="ProstateRobot_CheckForStalls"/>
10        <testcase classname="Surgical_Robot_Test_Suite" name="ProstateRobot_ValidateForwardKinematics"/>
11        <testcase classname="Surgical_Robot_Test_Suite" name="ProstateRobot_RunInverseKinematics"/>
12        <testcase classname="Surgical_Robot_Test_Suite" name="ProstateRobot_IsFootPedalPressed"/>
13        <testcase classname="Surgical_Robot_Test_Suite" name="ProstateRobot_LisMotors"/>
14        <testcase classname="Surgical_Robot_Test_Suite" name="ProstateRobot_GetMotor"/>
15        <testcase classname="Surgical_Robot_Test_Suite" name="Motor_GetMinTicksGetMaxTicks"/>
16        <testcase classname="Surgical_Robot_Test_Suite" name="Encoder_SetPosition"/>
17        <testcase classname="Surgical_Robot_Test_Suite" name="Logger_SetLogLevel"/>
18        <testcase classname="Surgical_Robot_Test_Suite" name="Packets_Connection"/>
19    </testsuite>
20</testsuites>
```

Figure 36: The figure shows an example of the results of the tests exported as an xml file with all successful tests.

```
 1<testsuites>
 2    <testsuite name="Surgical_Robot_Test_Suite" tests="16">
 3        <testcase classname="Surgical_Robot_Test_Suite" name="NeuroRobot_CheckForStalls"/>
 4        <testcase classname="Surgical_Robot_Test_Suite" name="NeuroRobot_ValidateForwardKinematics"/>
 5        <testcase classname="Surgical_Robot_Test_Suite" name="NeuroRobot_RunInverseKinematics"/>
 6        <testcase classname="Surgical_Robot_Test_Suite" name="NeuroRobot_IsFootPedalPressed"/>
 7        <testcase classname="Surgical_Robot_Test_Suite" name="NeuroRobot_LisMotors"/>
 8        <testcase classname="Surgical_Robot_Test_Suite" name="NeuroRobot_GetMotor"/>
 9        <testcase classname="Surgical_Robot_Test_Suite" name="ProstateRobot_CheckForStalls"/>
10        <testcase classname="Surgical_Robot_Test_Suite" name="ProstateRobot_ValidateForwardKinematics"/>
11        <testcase classname="Surgical_Robot_Test_Suite" name="ProstateRobot_RunInverseKinematics"/>
12        <testcase classname="Surgical_Robot_Test_Suite" name="ProstateRobot_IsFootPedalPressed"/>
13        <testcase classname="Surgical_Robot_Test_Suite" name="ProstateRobot_LisMotors"/>
14        <testcase classname="Surgical_Robot_Test_Suite" name="ProstateRobot_GetMotor"/>
15        <testcase classname="Surgical_Robot_Test_Suite" name="Motor_GetMinTicksGetMaxTicks">
16            <failure message="..\src\Test_Suite\Unit_Tests.cpp:283 Motor_GetMinTicksGetMaxTicks: 7 == max expected: 7   but was:    600000  ">
17Motor_GetMinTicksGetMaxTicks: 7 == max expected:    7   but was:    600000
18            </failure>
19        </testcase>
20        <testcase classname="Surgical_Robot_Test_Suite" name="Encoder_SetPosition"/>
21        <testcase classname="Surgical_Robot_Test_Suite" name="Logger_SetLogLevel"/>
22        <testcase classname="Surgical_Robot_Test_Suite" name="Packets_Connection"/>
23    </testsuite>
24</testsuites>
```

Figure 37: The figure shows an example of the results of the tests exported as an xml file with one failing tests.

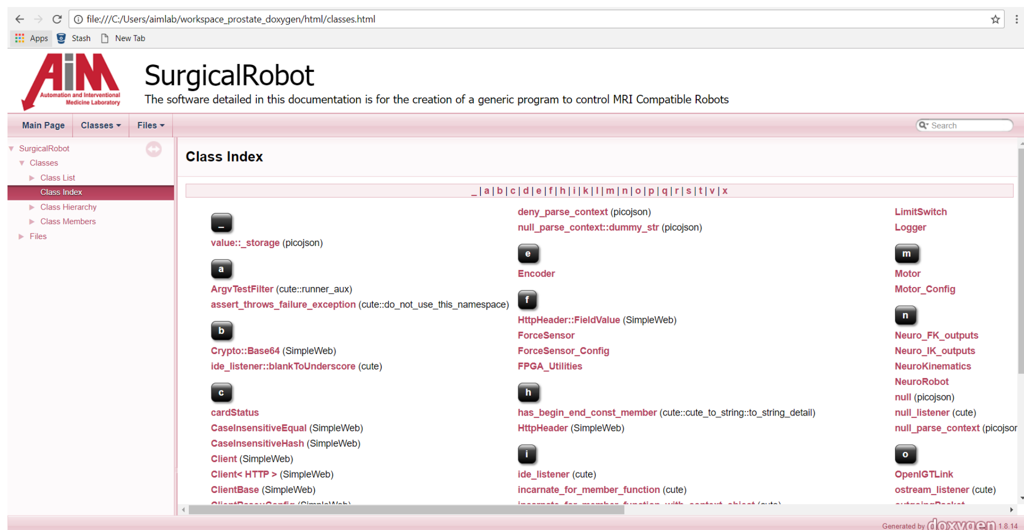


Figure 38: The figure shows an example of the Doxygen Documentation generated for the C++ software.

6 Conclusion and Future Directions

The contributions of this thesis was a designed and developed modular and extensible low-level C++ architecture. A web-based engineers console was constructed to provide a point of interface with the C++ code by developers. The software system was built to address many of the shortcoming of earlier systems, followed an object oriented approach, and included kinematic and control implementations. The architecture was validated for real-time consistency, robot targeting, control functionality, and safe-guards.

The C++ software architecture has also shown to be generically compatible with different MRI surgical robot systems by integrating with the NeuroRobot and ProstateRobot.

The program was also shown to be robust by testing the software system during experimentation at the UMass Memorial Hospital in Worcester and the Brigham and Women's Hospital in Boston. The architecture can also be considered simple as the web application is easy to use and the object oriented design of the code makes it straightforward to extend and maintain. Customizable robot objects and motor classes allow for different robot configurations and functionalities to be implemented. Software tests provide further validation, and the program documentation can serve as a guide for future developers.

Future work on the system should move past infrastructure and towards the implementation of more complex behaviors. One example could be to extend the C++ code for the NeuroRobot to allow for conformal ablation procedures. The current construction of the software also facilitates programming these more sophisticated behaviors as they can be coded within single methods, largely independent from other parts of the software. Work should also be made to improve the control implementation to reduce noise in collecting the current velocity of any given axis. This can be done by getting an encoder that counts faster than the C++ code or moving the current velocity calculation to the FPGA code. Another avenue for improvement could be to allow for Robot objects or class templates to be created from xml or text files of desired configurations. The documentation should continue to be built with changes to the code base, however, Doxygen largely facilitates this process. While the software will continue to grow, the developed architecture will serve as a solid foundation for any new robots or use cases.

A Appendix

A.1 Code Repository & Documentation

With the proper authorization, the code developed in this thesis can be found at:
http://fischerlab2.wpi.edu:7990/projects/ROBCTRL/repos/software/browse

References

[1] A. M. Tahmasebi, P. Abolmaesumi, D. Thompson and K. Hashtrudi-Zaad, "Software structure design for a haptic-based medical examination system," IEEE International Workshop on Haptic Audio Visual Environments and their Applications, 2005

[2] Denger, C., Feldmann, R. L., Host, M., Lindholm, C., & Shull, F. (2007, September). A snapshot of the state of practice in software development for medical devices. In Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on (pp. 485-487). IEEE.

[3] Chakravorty, R. (2006, March). A programmable service architecture for mobile medical care. In Pervasive Computing and Communications Workshops, 2006. PerCom Workshops 2006. Fourth Annual IEEE International Conference on (pp. 5-pp). IEEE.

[4] Eslami, S, Shang, W, Li, G, Patel, N, Fischer, G, Tokuda, J, & Iordachita, I (2016). In-bore prostate transperineal interventions with an MRI-guided parallel manipulator: system development and preliminary evaluation. The International Journal of Medical Robotics and Computer Assisted Surgery

[5] Farquhar, Adam, Richard Fikes, and James Rice. "The ontolingua server: A tool for collaborative ontology construction." International journal of human-computer studies 46.6 (1997): 707-727.

[6] Fedorov, A., Beichel, R., Kalpathy-Cramer, J., Finet, J., Fillion-Robin, J. C., Pujol, S., & Buatti, J. (2012). 3D Slicer as an image computing platform for the Quantitative Imaging Network. Magnetic resonance imaging, 30(9), 1323-1341.

[7] Food and Drug Administration of USA (2016). Code of Federal Regulations Title 21 PART 820 – QUALITY SYSTEM REGULATION (FDA No. 21 CFR §820.30). Retrieved from https://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfcfr/CFRSearch.cfm?fr=820.30

[8] Garnette R. Sutherland, Isabelle Latour, Alexander D. Greer, Tim Fielding, Georg Feil, Perry Newhook; An Image-Guided Magnetic Resonance Compatible Surgical Robot, Neurosurgery, Volume 62, Issue 2, 1 February 2008, Pages 286–293,

[9] General Principles of Software Validation; Final Guidance to Industry and Staff (n.d.) 11 January 2002 U.S. Department Of Health and Human Services Food and Drug Administration Center for Devices and Radiological Health Center for Biologics Evaluation and Research

[10] International Organization for Standardization. (2016). Medical devices – Quality management systems – Requirements for regulatory purposes (ISO/DIS Standard No. 13485). Retrieved from https://www.iso.org/iso-13485-medical-devices.html

[11] International Organization for Standardization. (2016). medical device software – software life cycle processes (ISO/DIS Standard No. 62304). Retrieved from https://www.iso.org/standard/38421.html

[12] Lanfranco, A. R., Castellanos, A. E., Desai, J. P., & Meyers, W. C. (2004). Robotic surgery: a current perspective. Annals of surgery, 239(1), 14.

[13] Lee, I., Pappas, G. J., Cleaveland, R., Hatcliff, J., Krogh, B. H., Lee, P., & Sha, L. (2006). High-confidence medical device software and systems. Computer, 39(4), 33-38.

[14] Lehman, C. D., Gatsonis, C., Kuhl, C. K., Hendrick, R. E., Pisano, E. D., Hanna, L., & DePeri, E. R. (2007). MRI evaluation of the contralateral breast in women with recently diagnosed breast cancer. New England Journal of Medicine, 356(13), 1295-1303.

[15] MacDonell, Jacquelyn, Niravkumar Patel, Sebastian Rubino, Goutam Ghoshal, Gregory Fischer, E. Clif Burdette, Roy Hwang, and Julie G. Pilitsis. "Magnetic resonance–guided interstitial high-intensity focused ultrasound for brain tumor ablation." Neurosurgical focus 44, no. 2 (2018): E11.

[16] May, F., T. Treumann, P. Dettmar, R. Hartung, and J. Breul. "Limited value of endorectal magnetic resonance imaging and transrectal ultrasonography in the staging of clinically localized prostate cancer." BJU international 87, no. 1 (2001): 66-69.

[17] Miller, Randolph A. "Medical diagnostic decision support systems—past, present, and future: a threaded bibliography and brief commentary." Journal of the American Medical Informatics Association 1.1 (1994): 8-27.

[18] Nycz, C. J., Gondokaryono, R., Carvalho, P., Patel, N., Wartenberg, M., Pilitsis, J. G., & Fischer, G. S. (2017, September). Mechanical validation of an MRI compatible stereotactic neurosurgery robot in preparation for pre-clinical trials. In Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on (pp. 1677-1684). IEEE.

[19] Schorr, O., Hata, N., Bzostek, A., Kumar, R., Burghart, C., Taylor, R. H., & Kikinis, R. (2000, October). Distributed modular computer-integrated surgical

robotic systems: architecture for intelligent object distribution. In International Conference on Medical Image Computing and Computer-Assisted Intervention (pp. 979-987). Springer, Berlin, Heidelberg.

[20] Software as a Medical Device: Possible Framework for Risk Categorization and Corresponding Considerations (n.d.) 18 September 2014, IMDRF Software as a Medical Device (SaMD) Working Group

[21] Su, Hao, et al. "Real-time MRI-guided needle placement robot with integrated fiber optic force sensing." Robotics and Automation (ICRA), 2011 IEEE International Conference on. IEEE, 2011.

[22] Stoianovici, D., Kim, C., Srimathveeravalli, G., Sebrecht, P., Petrisor, D., Coleman, J., & Hricak, H. (2014). MRI-safe robot for endorectal prostate biopsy. IEEE/ASME Transactions on Mechatronics, 19(4), 1289-1299.

[23] Tokuda, Junichi, Gregory S. Fischer, Simon P. DiMaio, David G. Gobbi, Csaba Csoma, Philip W. Mewes, Gabor Fichtinger, Clare M. Tempany, and Nobuhiko Hata. "Integrated navigation and control software system for MRI-guided robotic prostate interventions." Computerized Medical Imaging and Graphics 34, no. 1 (2010): 3-8.

[24] Tokuda J. and Fischer, G. "OpenIGTLink: an open network protocol for image-guided therapy environment," Int J Med Robot, vol. 5, pp. 423–34, Dec. 2009.

[25] Wartenberg, Marek, et al. "Towards synergistic control of hands-on needle insertion with automated needle steering for MRI-guided prostate interventions." Engineering in Medicine and Biology Society (EMBC), 2016 IEEE 38th Annual International Conference of the. IEEE, 2016.

[26] Yang, B., Roys, S., Tan, U. X., Philip, M., Richard, H., Gullapalli, R. P., & Desai, J. P. (2014). Design, development, and evaluation of a master–slave surgical system for breast biopsy under continuous MRI. The International journal of robotics research, 33(4), 616-630.

[27] Yang, B., Tan, U. X., McMillan, A. B., Gullapalli, R., & Desai, J. P. (2011). Design and control of a 1-DOF MRI-compatible pneumatically actuated robot with long transmission lines. IEEE/ASME transactions on mechatronics, 16(6), 1040-1048.