Matlab Interface for the CISST Libraries

Final Report

Group 16

Team Member: Zachary Zhou

Mentor: Anton Deguet

Background, Aims, Significance:

"The *cisst* package is a collection of libraries designed to ease the development of computer assisted intervention systems. The Surgical Assistant Workstation (SAW) is a platform that combines robotics, stereo vision, and intraoperative imaging (e.g., ultrasound) to enhance a surgeon's capabilities. The SAW package therefore consists of implemented components (e.g., interfaces to many of the devices used for computer-integrated surgery) as well as reusable applications."

Although the *cisst* library is a very powerful tool utilized in many of the medical robots in this lab, the entire library is written in C++. As a result, one requires a sufficient understanding of C++ in order to fully utilize the library. However, not necessarily all of the researchers who would like the use the cisst libraries are proficient in the C++ language.

In order to make the cisst package more accessible, it would be beneficial to port the library onto a different language. For example, MATLAB happens to be a popular language that is fairly easy to learn and utilize.

There are several advantages to utilizing MATLAB over C++. First of all, the MATLAB package includes many numerical methods for matrix manipulation and mathematical interfaces, making it a powerful package for reducing and analyzing data. In addition, MATLAB uses very loose type definitions which, in addition the command console, makes it very user friendly. Finally, researchers often reduce data acquired through the cisst libraries on MATLAB.

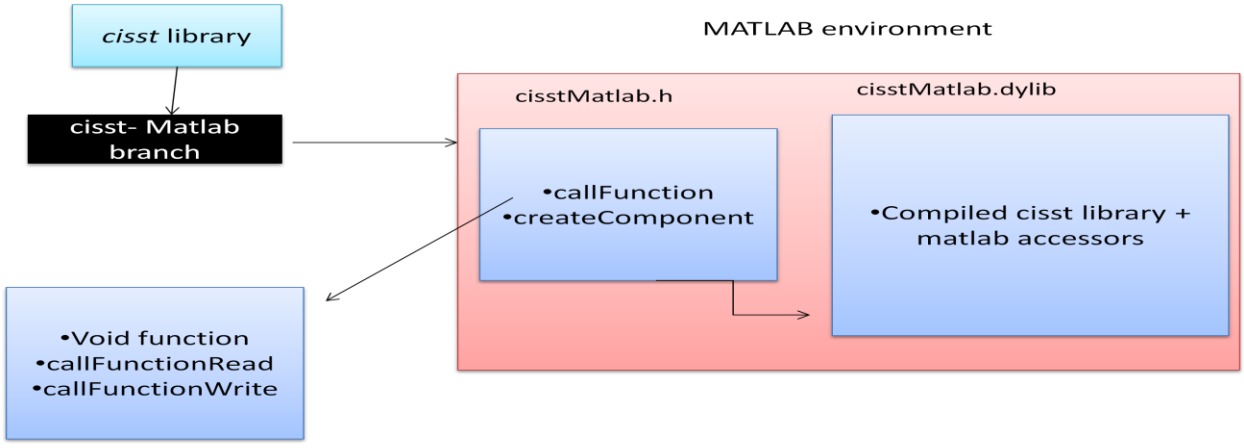
Overview of Goals:

Creation of a general purpose dynamic wrapper that will allow the use of the *cisst* packages from MATLAB. A breakdown of the goals are listed below:

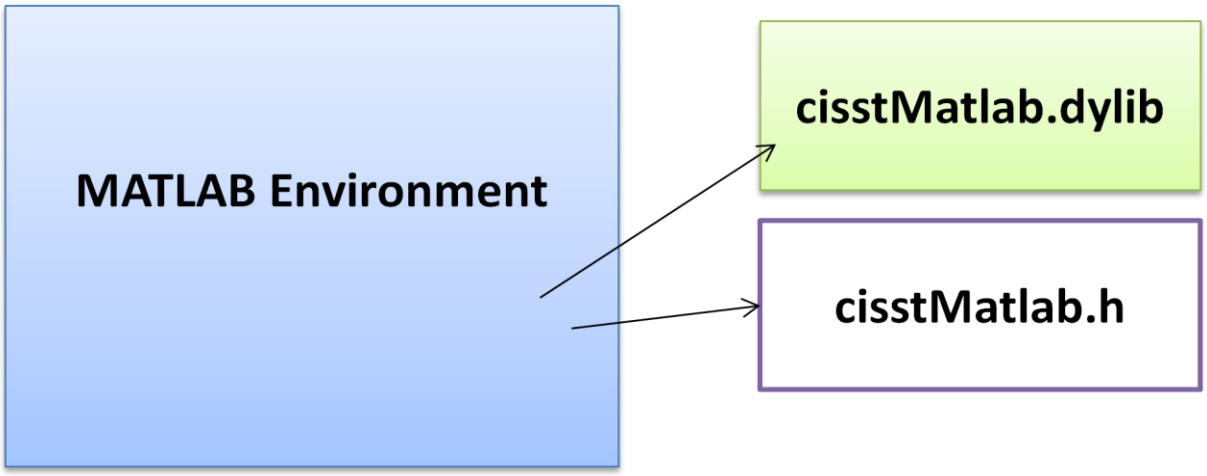
- General MATLAB wrapper for the cisst classes
- Utilize CMake to create the plug-in library
- Dynamically load cisst components onto MATLAB
- Handle data manipulation between C++ and MATLAB

Technical Approach:

In order to keep the wrapper as dynamic as possible, we chose to utilize MATLAB's ability to load dynamic libraries into the MATLAB environment (*loadlibary*) over MATLAB's compiled mex files. Utilizing shared libraries offers several advantages over mex files. First of all, MEX requires all C source code to be recompiled every time in order for MATLAB to be able to access the C libraries. Also, there is only one point of entry for MEX files: the mexfunction. This severely limits our MATLAB calls to C functions and requires significant amounts of string manipulation in order to correctly work. Finally, objects created in MEX calls do not always remain after the *mexfunction* returns. Often, variables are collected by MATLAB's garbage collector.



For this wrapper, we created a branch from the *cisst* trunk. After which, we added several C functions and MATLAB script files in order to facilitate the wrapping the library. Once compiled ultilizing CMAKE, there are two files which MATLAB is concerned with: *cisstMatlab.h* and *cisstMatlab.dylib*. *cisstMatlab.dylib* is the compiled dynamic library which contains the cisst source and additional C functions to be loaded onto MATLAB. *cisstMatlab.h* identifies which C functions MATLAB can directly interact with.

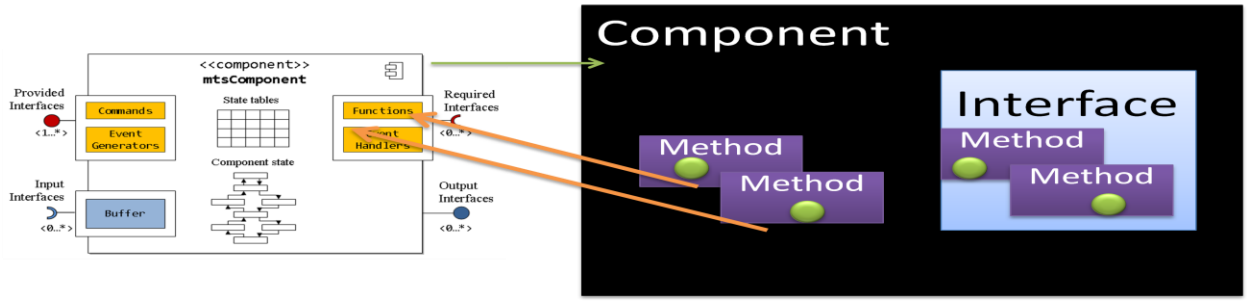


After the dynamic library is created, the cisst wrapper can be loaded onto the MATLAB interface using a call to *loadlibrary* from the MATLAB environment.

We decided to separate calls from MATLAB to C to be handled by two main C files: *creatComponent.cpp* and *callFunction.cpp*.

A call to create component will result in an object of the desired type being created on the C side, and a dynamic prop wrapping the object to be created on the MATLAB environment. Each object created in the MATLAB environment is a dynamic_prop. A dynamic_prop is a MATLAB type which mimics a class definition. However, there are several advantages in using dynamic_prop's over classes. First of all, we are able to define props on the fly and add which ever components we chose to it. Secondly, the creation of MATLAB classes requires that a file be created in the MATLAB working directory. Utilizing dynamic_props will allow us to not have to create a MATLAB class file every time new object is created.

The resultant dynamic_prop which is returned to the MATLAB environment is mostly an empty class. The prop itself corresponds to the component which was created, subcomponents and interfaces are attached as dynamic props to the main component.



Function calls are also stored as dynamic_props under the main component. However, the value for function calls are not empty. Instead, they are anonymous MATLAB functions which will call the C library and pass arguments to the C side. Each function prop holds the pointer to the mirror function on the C side. A function dynamic prop will general be defined as such:

Function= @()calllib('libcisstMatlab','mtlCallFunction', -function pointer-, args)

This allows us to call from the MATLAB side:

Result=ComponentA.interface1.DoSomething();

Also, when passing arguments from C to MATLAB, we had to wrap matricies/arrays into/from MATLAB. To do this, we would wrap arguments from C to MATLAB as *mexArrays*. When passing from MATLAB to C, we wrapped as *mtsVector* types.

This method of wrapping allows us to create and wrap cisst objects onto the MATLAB interface without having to output dummy files to support class/function declarations in the MATLAB interface.

Results:

We managed to create a MATLAB wrapper for the *cisst* libraries with partial functionality. In addition, there are several MATLAB scripts which will facilitate user's interactions with the wrapper.

After the library is compiled, the user will find a bash script which will set the directory of the MATLAB wrapper in the kernel. At this point, the user should run MATLAB directly from the kernel (open –a "MATAB path").

In the MATLAB interface, the user is able to load and unload the cisst wrapper using the provided *mtlLoad* and *mtlUnload* functions. These functions will find the cisst directory and load the cisst-MATLAB library for the user.

Also, the creation of components has been facilitated with the *createComponent* function. With a call to create component, the user can create cisst objects without having knowledge of how to use MATLAB's *calllib* function.

Conclusion:

Overall, the project was fairly successful. We were able to meet all our minimum and most of the expected deliverables by the completion of the project. We were able to create a working MATLAB wrapper which could call methods from the cisst libraries. However, we were unable to dynamically load onto MATLAB a list of components/interfaces to be created. As a result, the creations of components had to be hard coded. In the final version, we chose to hard code the component which was created and the interfaces/functions which populated the component.

One factor which limited the progress of this project was that that we had initially experimented with using both MEX functions and MATLAB class objects to wrap the cisst libraries. After it was decided to use dynamic_props to give the wrapper a feel like the python wrapper, a lot of code had to be scrapped and re-written.

Deliverables met:

In the current state of this project, we have met all minimum deliverables and most expected deliverables. The wrapper is able to load components onto MATLAB without the need of a configuration file. In addition, the library can dynamically load the cisst libraries onto MATLAB. We were able to support the conversion of basic types and simple vectors/matrices to MATLAB. In addition, we were able to population the MATLAB object with interfaces and commands. We were also able to create MATLAB object wrappers with simple string names.

Future Work:

One primary concern is to remove the hard coding of component creation and population. To do this, an improved way of passing component values from C to MATLAB must be created.

Also, we need to allow for all functions from the cisst libraries to be called. Not simply the ones which are hard coded in. This is simply a matter of populating the list of functions/interfaces to the MATLAB side as the current functions are simply wrappers for functions pointers with support for data type conversion.

In addition, a way must be found to create pointer types in the MATLAB environment. Currently, we are passing strings to and from the C side and using reinterpret_cast to extract the function pointer. However, we would like to be able to create a pointer to a location in memory in MATLAB.

Optimally, we would like to ensure that the library compiles and loads on different operating systems. The library was created on OSX, however we would like to ensure that it performs correctly on Windows and other environments.

Management Summary

I had meetings with Anton when we needed to make changes to the code or to go over significant portions of the code. Otherwise, I updated Anton with my progress using email and a SVN repository was created to help with the wrapper.

Reading List

- Vincent Chu, Ghassan Hamarneh "MATLAB-ITK Interface for Medical Image Filtering, Segmentation, and Registration". <www.cs.sfu.ca/~hamarneh/ecopy/medical_showcase2005a.pdf>

References

- https://trac.lcsr.jhu.edu/cisst
- https://trac.lcsr.jhu.edu/cisst/wiki/cisstMultiTaskTutorial
- http://www.mathworks.com/support/technotes/1600/1605.html
- http://www.cmake.org/cmake/resources/resources.html
- http://www.mathworks.com/support/tech-notes/1600/1605.html