Metrics

Complex metrics

For recording metrics that have complicated structures required for the specified validations, we went for the serialization library that comes with Boost collection. The library allows for serializing c++ structs into boost format (written in ascii). The serialized data can be de-serialized at a later stage and then converted into any format necessary for analysis and validation. A serialize function needs to be provided within structs that should be serialized. Boost serialization library provides a method to write serialization functions for data types that cannot be modified to include a serialize function such as with glm vectors, matrices that are required for recording metrics.

1. *Non-intrusive serialization for library defined data types*

To serialize data types defined by libraries such as glm without modifying them, boost::serialization library provides a way to write serialization functions externally. For logging the required metrics, we use glm vectors, quaternions and matrices within structs created for logging. The code below shows how serializing functions can be written in non-intrusively.

```
using glm::vec3;
using glm::vec4;
using glm::mat4;
using glm::quat;
namespace boost {
    namespace serialization {
        template <class Archive>
        void serialize(Archive& ar, vec3& p, const unsigned int /* version */)
        {
            ar & p.x & p.y & p.z;
        }
        template <class Archive>
        void serialize(Archive& ar, vec4& p, const unsigned int /* version */)
        {
            ar & p.x & p.y & p.z & p.w;
        }
        template <class Archive>
        void serialize(Archive& ar, mat4& p, const unsigned int /* version */)
        {
            ar & p[0] & p[1] & p[2];
        }
        template <class Archive>
        void serialize(Archive& ar, quat& p, const unsigned int /* version */)
        {
            ar & p.w & p.x & p.y & p.z ;
        }
    }
}
```

2. *Writing a struct for serialization*

```
struct my_struct {
       vec3 position;
       quat orientation;
       region_struct(vec3 position, quat orientation) :
              position(position), orientation(orientation) {}
private:
       friend class boost::serialization::access;
       template<class Archive>
       void serialize(Archive &ar, const unsigned int version) {
                     ar & position & orientation;
       }
};
```

3. *Serialize an object of struct `my_struct` to a stream*

```
boost::archive::text_oarchive oa(stream);
       oa << m;
```

`

4. *De-serializing*

```
boost::archive::text_iarchive ia(s);
       ls::log_cut_struct m1;
       ia >> m1;
```

Simple metrics

As for metrics that can be represented with one to few variables, we used the spdlog library combined with Boost format library for logging it as comma separated values to files. We chose to use this combination of libraries to log simple metrics since they were already part of the project for producing logs for debug.

Writing a row to file using spdlog

```
void Logger::log_collision_count(float col_count) {
	collision_logger->info(str(boost::format("%f,%f") % get_elapsed() % col_count));
}
```

Recording complex and simple metrics

The types of data logged are classified into three categories, resulting in total of three log files to be generated for a cutting task. For each of the required categories, we have created structs that have the required member variables.

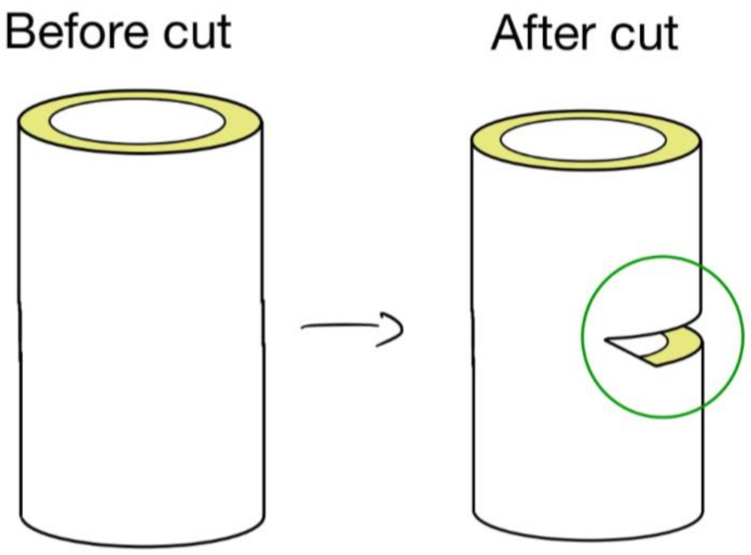
NOTE: The constructors and serialization functions have been omitted for legibility.

1. Per Cut: Data recorded after a single cut is completed. In the case of scissors, a single cut is defined from when the scissors open (t1) near a mesh to when the scissors close (t2), resulting in mesh deformation. In the case of a single blade, a single cut is defined from when a blade collides with a mesh (t1), to when it is moved away from the mesh (t2), resulting in mesh deformation.

```
struct region_struct {
	vec3 position;
	quat orientation;
	vec3 shape;
}

struct log_cut_deformation_struct {
	double time_elapsed;
	region_struct predef_top;
	region_struct predef_bot;
	region_struct postdef_top;
	region_struct postdef_bot;
	region_struct blade_swipe_region;
}
```

The variables recorded per cut includes top and bottom regions for pre deformation and post deformation. Additionally, we also record the time elapsed since beginning of simulation. To represent a region, we have created a nested struct that holds position, orientation and the vertices of the region.



2. Per Collision: Data recorded at each collision between the tool and the mesh of interest.

```
Logger::log_collision_count(float col_count) {
	collision_logger->info(str(boost::format("%f,%f") % get_elapsed() %
col_count));
}
```

Per collision, we record the time elapsed since start of simulation along with a counter that tracks unnecessary collisions. The data is stored as comma separated values, with each call to the function being written to a new line.

3. Per Time Frame: Data recorded at all times.

```
struct log_per_time_frame_struct {
	double time_elapsed;
	mat4 blade_frame;
	int exit_counter;
}
```

During each time frame, we are recording the frame of the blade in represented by a 4x4 matrix, the time elapsed since start of simulation and an exit counter that counts the number of times the tool exited the surgical scene.