Continuum Stewart-Gough Teleoperation

With our work to speed up the inverse kinematics, we are in a good position to control a robot. We will implement a *teleoperation* control scheme where a human operator has interactive control over the end effector pose. As part of my graduate studies I have done this for a robot using an Xbox controller as the input device and an Arduino to control the robot's linear actuators (video). I don't want construction of a robot and possession of an Xbox controller to be prerequisites for this code example, so we'll control a virtual robot using keyboard input. The robot model will be visualized in real-time using a Qt application.

We will use Qt charts to show the robot shape, and we will rely on four different views. When creating a new "Qt Widgets Application" project, by default a class "MainWindow" is created to represent the application window. In the main window header file, we add three lines declaring the chart elements we will use:

```cpp
QtCharts::QChart charts[4];
QtCharts::QLineSeries legs[4][6];
QtCharts::QLineSeries ee[4];
```

There are four charts, one for each view of the robot. There are also line series objects which will represent the robot legs and end effector in a chart. These objects are persistent and the points they contain are updated periodically as the robot moves. We declare a function to setup the charts when the application is started:

```cpp
private:
    void initializeCharts();
```

This is called in the main window constructor, and it is defined in "mainwindow.cpp". It begins by adding chart views to the window:

```cpp
void MainWindow::initializeCharts(){
    QtCharts::QChartView* view_front = new QtCharts::QChartView(&charts[0]);
    QtCharts::QChartView* view_top = new QtCharts::QChartView(&charts[1]);
    QtCharts::QChartView* view_side = new QtCharts::QChartView(&charts[2]);
    QtCharts::QChartView* view_ortho = new QtCharts::QChartView(&charts[3]);
    charts[0].setTitle("<b>Front</b>");
    charts[1].setTitle("<b>Top</b>");
    charts[2].setTitle("<b>Side</b>");
    charts[3].setTitle("<b>Orthographic</b>");
    ui->leftPane->addWidget(view_top);
    ui->leftPane->addWidget(view_front);
    ui->rightPane->addWidget(view_ortho);
    ui->rightPane->addWidget(view_side);
```

This creates a view for each chart, gives the charts titles, and then adds them to the main window. The "ui->leftPane" and "ui->rightPane" are defined in "mainwindow.ui", which was created using Qt's form editor. We let the QChartView pointers escape into the ether because they don't need to be destroyed until the application ends.

In the next section of "initializeCharts", we format the line series and add them to the charts:

```
QPen leg_pen(QColor::fromRgb(88,89,91)), ee_pen(QColor::fromRgb(255,130,0));
leg_pen.setWidthF(1.5);
ee_pen.setWidthF(2);
for(int i = 0; i < 4; i++){
    for(int j = 0; j < 6; j++){
        legs[i][j].setPen(leg_pen);
        charts[i].addSeries(&legs[i][j]);
    }
    ee[i].setPen(ee_pen);
    charts[i].addSeries(&ee[i]);

    charts[i].createDefaultAxes();
    charts[i].legend()->hide();
}
```

The outer loop iterates over the four different views, and an inner loop iterates over the line series for each leg. We also initialize the chart axes, which we edit in the next step:

```
charts[0].axes(Qt::Horizontal).back()->setRange(-0.25, 0.25);
charts[0].axes(Qt::Vertical).back()->setRange(0, 0.6);
charts[0].axes(Qt::Horizontal).back()->setTitleText("x (m)");
charts[0].axes(Qt::Vertical).back()->setTitleText("z (m)");

charts[1].axes(Qt::Horizontal).back()->setRange(-0.25, 0.25);
charts[1].axes(Qt::Vertical).back()->setRange(-0.25, 0.25);
charts[1].axes(Qt::Horizontal).back()->setTitleText("x (m)");
charts[1].axes(Qt::Vertical).back()->setTitleText("y (m)");

charts[2].axes(Qt::Horizontal).back()->setRange(-0.25, 0.25);
charts[2].axes(Qt::Vertical).back()->setRange(0, 0.6);
charts[2].axes(Qt::Horizontal).back()->setTitleText("y (m)");
charts[2].axes(Qt::Vertical).back()->setTitleText("z (m)");

charts[3].axes(Qt::Horizontal).back()->setRange(-0.2, 0.2);
charts[3].axes(Qt::Vertical).back()->setRange(0, 0.6);
charts[3].axes(Qt::Horizontal).back()->hide();
charts[3].axes(Qt::Vertical).back()->hide();
```

There are several lines of code here to handle the specifics of each view, but the logic is straightforward. Finally, we end the chart initialization function by plotting the robot shape:

```
    refreshCharts();
}
```

There are two important lines in the header to consider before getting into the details of the plotting method. The upper right view will not look directly along one of the global axes, so the header defines a rotation for this view:

```
const RowVector3d ortho_x = (Ry(pi/6)*Rz(pi*5/12)).row(1);
const RowVector3d ortho_y = (Ry(pi/6)*Rz(pi*5/12)).row(2);
```

The method "refreshCharts" is called whenever the line series data should be updated. This method is declared in the header, and then defined:

```cpp
void MainWindow::refreshCharts(){
    QList<QPointF> ee_points[4];

    for(int i = 0; i < 6; i++){
        QList<QPointF> leg_points[4];
        Matrix3Xd leg_data = getLegCenterline(i);
        int N = static_cast<int>(leg_data.cols());

        for(int j = 0; j < N; j++){
            Vector3d p = leg_data.col(j);

            leg_points[0] << QPointF(p.x(), p.z());
            leg_points[1] << QPointF(p.x(), p.y());
            leg_points[2] << QPointF(p.y(), p.z());
            leg_points[3] << QPointF(ortho_x*p, ortho_y*p+0.1);
        }
        for(int j = 0; j < 4; j++){
            legs[j][i].replace(leg_points[j]);
            ee_points[j] << leg_points[j].last();
        }
    }

    for(int i = 0; i < 4; i++){
        ee_points[i] << ee_points[i].first();
        ee[i].replace(ee_points[i]);
    }
}
```

This method loops over each point in each leg to update the line series. The final point of each leg is added to the end-effector series. After all six final points are added to the end-effector series, the first point is added again to close the shape. The one loose string here is the "getLegCenterline" function, which relies on the inverse kinematics model.

We move the CSG inverse kinematics code into a file "InverseKinematicsCSG.h". We'll make a few modifications so it fits better with the teleoperation loop we are going to create. Previously we setup the Stewart-Gough offsets using an initialization function. It would be more convenient to initialize all parameters when they are declared, which we accomplish by:

```cpp
//Hexagonal Stewart-Gough hole pattern
#define theta_b(i) ((-alpha2 + ((i+1) - (i+1)%2)*alpha2 + (i-i%2)*alpha1)/2)
#define theta_L(i) ((-alpha1 + ((i+1) - (i+1)%2)*alpha1 + (i-i%2)*alpha2)/2)
#define pb(i) ( scrib_R * Vector3d(cos(theta_b(i)), sin(theta_b(i)), 0) )
#define rL(i) ( scrib_R * Vector3d(cos(theta_L(i)), sin(theta_L(i)), 0) )
const Vector3d p0[6] = {pb(0), pb(1), pb(2), pb(3), pb(4), pb(5)};
const Vector3d r[6] = {rL(0), rL(1), rL(2), rL(3), rL(4), rL(5)};
#undef theta_b
#undef theta_L
#undef pb
```

The arrays are initialized using several macros to make the code more readable. We promptly undefine the macros so that we cannot accidentally invoke them later. The guess declaration is also altered to include an initialization:

```cpp
static VectorXd guess =
    (VectorXd(36) << VectorXd::Zero(30), pE(2)*VectorXd::Ones(6)).finished();
```

The syntax is a bit awkward, but the "finished" method allows us to return a vector from the comma initializer.

Now that the robot model is declared in an initialized state, we can write a fairly short inverse kinematics function to calculate the leg lengths given the pose:

```cpp
/*! Find the robot leg lengths to achieve a given target pose. */
inline Vector6d inverseKinematicsCSG(Vector3d pE, Matrix3d RE){
    ::pE = pE; //The double-colon syntax '::' refers to the global scope
    ::RE = RE;

    //Solve with shooting method
    guess = solveLevenbergMarquardt<shootingFunction, jacobianFunction>
                                    (guess, 1e-6, 500, 1e-4, 0.5);

    return guess.segment<6>(30);
}
```

This is the very definition of inverse kinematics: update the target pose, solve the kinematic equations, return the actuator values. The main window header has fields for the kinematics:

```cpp
Vector3d pE = 0.4*Vector3d::UnitZ();
Matrix3d RE = Matrix3d::Identity();
Vector6d L = inverseKinematicsCSG(pE,RE);
```

Setting "L" by solving the robot inverse kinematics means that the robot model is in a solved state before the "MainWindow" constructor is even called.

With the kinematics done, a function in "InverseKinematicsCSG.h" gets the centerlines:

```cpp
/*! Returns a matrix where each column is the 3D position of a grid point. */
inline Matrix3Xd getLegCenterline(int i){
    Vector3d n0 = guess.segment<3>(5*i);
    Vector2d m0xy = guess.segment<2>(5*i+3);
    double L = guess(30+i);
    y << p0[i], 1, 0, 0, 0, 1, 0, 0, 0, 1, n0, m0xy, 0;

    return ode4<cosseratRodOde,N>(y,L).block<3,N>(0,0);
}
```

This is used by "refreshCharts" to display the robot as covered earlier. If we ran the program using everything covered so far, four plots of the robot in the initial configuration would appear.
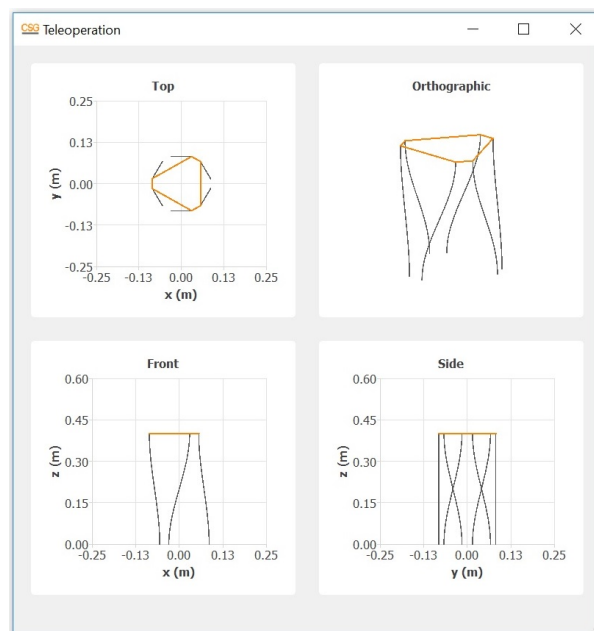
However, the program will not respond to user input yet.

To achieve keyboard input, we override the "keyPressEvent" function in the main window:

```cpp
void MainWindow::keyPressEvent(QKeyEvent* event){
    const double v_incr = 1e-4;
    const double w_incr = 1e-3;
    const int steps = 25;

    Vector3d vE = Vector3d::Zero();
    Vector3d wE = Vector3d::Zero();

    switch(event->key()){
        case Qt::Key_A: vE.x() -= v_incr; break;
        case Qt::Key_D: vE.x() += v_incr; break;
        case Qt::Key_S: vE.y() -= v_incr; break;
        case Qt::Key_W: vE.y() += v_incr; break;
        case Qt::Key_Q: vE.z() -= v_incr; break;
        case Qt::Key_E: vE.z() += v_incr; break;
        case Qt::Key_I: wE.x() -= w_incr; break;
        case Qt::Key_K: wE.x() += w_incr; break;
        case Qt::Key_J: wE.y() -= w_incr; break;
        case Qt::Key_L: wE.y() += w_incr; break;
        case Qt::Key_U: wE.z() -= w_incr; break;
        case Qt::Key_O: wE.z() += w_incr; break;
        case Qt::Key_Space: resetKinematicsSolver();
    }

    try{
        for(int i = 0; i < steps; i++){
            pE += vE;
            RE += hat(wE)*RE;
            L = inverseKinematicsCSG(pE,RE);
        }
    }catch(...){
        resetKinematicsSolver();
    }
    refreshCharts();
}
```

Depending on which key is pressed, there is some velocity or angular velocity. We handle each different key press and then update the velocities based on the result. We use a loop to step through the kinematic solution, which is not strictly necessary, but hopefully will improve the solver robustness. Finally we update the charts based on the new configuration.

There is one problem. We are relying on convex optimization to solve the kinematics, and we may not find a minimum which meets the tolerance in a satisfactory number of iterations. If the solver ever fails to find a solution, we want to handle it gracefully. Although maybe not ideal, the most straightforward solution is to start over with the robot in the initial configuration, which we accomplish with another member function:

```cpp
void MainWindow::resetKinematicsSolver(){
    resetCSG();
    pE = 0.4*Vector3d::UnitZ();
    RE = Matrix3d::Identity();
    L = inverseKinematicsCSG(pE,RE);
}
```

In the first step, we need to reset the convex optimization guess so that we won't find ourselves in the same predicament the next time we invoke the solver. Then we solve the inverse kinematics to re-initialize the robot model. Resetting the guess is accomplished by a short function in the kinematics header:

```
/*! Reset the convex optimization guess. */
inline void resetCSG(){
    guess << VectorXd::Zero(30), 0.4*VectorXd::Ones(6);
}
```

Now if we run the program, the robot will respond to keystrokes and move around. The teleoperation loop can even run fluidly on a Raspberry Pi 3B, so the kinematics solution speed is satisfactory. The core of a control scheme is in place; Qt has capabilities for serial communication if we wanted to extend this program to communicate with robotic hardware, but we won't worry about controlling the linear actuators on a physical robot here.