Solving a Cosserat Rod Boundary Value Problem

In the last chapter we solved a Cosserat rod initial value problem where $\boldsymbol{p}(0)$, $\boldsymbol{R}(0)$, $\boldsymbol{n}(0)$, and $\boldsymbol{m}(0)$ are known. However, in most applications we won't have a 6-dof force sensor attached at the base to measure $\boldsymbol{n}(0)$ and $\boldsymbol{m}(0)$. More commonly we know information at either end, for example if a robot uses two grippers to hold both ends of the rod, then we know $\boldsymbol{p}(0)$, $\boldsymbol{R}(0)$, $\boldsymbol{p}(L)$, and $\boldsymbol{R}(L)$. We can still solve such a problem, known as a boundary value problem (BVP), but the process is more complicated.

Since we're going to be referring to the boundary conditions (BCs) in several functions, we declare them as global variables:

```
//Boundary conditions
const Vector3d p0 = Vector3d::Zero();
const Matrix3d R0 = Matrix3d::Identity();
const Vector3d pL = Vector3d(0, -0.1*L, 0.8*L);
const Matrix3d RL = Matrix3d::Identity();
```

There's nothing too exciting here; Vector3d has nice constructor which allows us to set each element. Now we've specified the pose (position and orientation) at both ends of the rod.

There are various methods of numerically solving BVPs, but since our problem is one-dimensional and we eventually want the solution process to be fast, we'll rely on the shooting method. There are two parts to a shooting method, an *objective function* and a solver. The objective function input is a guess for the unknown state variables at one of the boundary, then the ODE is integrated to determine how badly the boundary conditions at the other end are violated. The solver minimizes this error violation, hopefully driving it to zero by some algorithm such as Newton-Raphson or Levenberg-Marquardt. The *numerical shooting method* is a well known approach with explanations readily available online, so I won't go into more detail here.

We write the *objective function* in C++ which takes the guessed base wrench (force and moment) as an input, then numerically integrates the rod to find the error in the pose (position and orientation) at the end:

```
static MatrixXd Y; //Declare Y global for shooting and visualization
VectorXd shootingFunction(VectorXd guess){
    VectorXd y0(18);
    y0 << p0, Map<VectorXd>(Matrix3d(R0).data(), 9), guess;

    //Numerically integrate the Cosserat rod equations
    Y = ode4<cosseratRodOde>(y0, L);

    //Calculate the pose (position and orientation) error at the far end
    Vector3d pL_shooting = Y.block<3,1>(0, Y.cols()-1);
    Matrix3d RL_shooting = Map<Matrix3d>(Y.block<9,1>(3, Y.cols()-1).data());

    Vector6d distal_error;
    distal_error << pL - pL_shooting, rotation_error(RL,RL_shooting);

    return distal_error;
}
```

We set the initial state including the guessed part, integrate the rod as an IVP, then compare the values of $\boldsymbol{p}(L)$ and $\boldsymbol{R}(L)$ which we found by "shooting" with the actual values of $\boldsymbol{p}(L)$ and $\boldsymbol{R}(L)$. The position error calculation is straightforward, but comparing the rotation matrices

requires a bit more effort. A rotation is a 3x3 matrix, but there aren't 9 independent error components between two rotations because each rotation already has its own set of constraints that $\boldsymbol{R}^T\boldsymbol{R} = \boldsymbol{I}$ and that the determinant of $\boldsymbol{R}$ is 1. This is why the "rotation_error" function returns a 3x1 vector expressing the error between the rotation matrices. Specifically, it calculates

$$err = \left(\log(\boldsymbol{R}_1\boldsymbol{R}_2^T)\right)^\vee$$

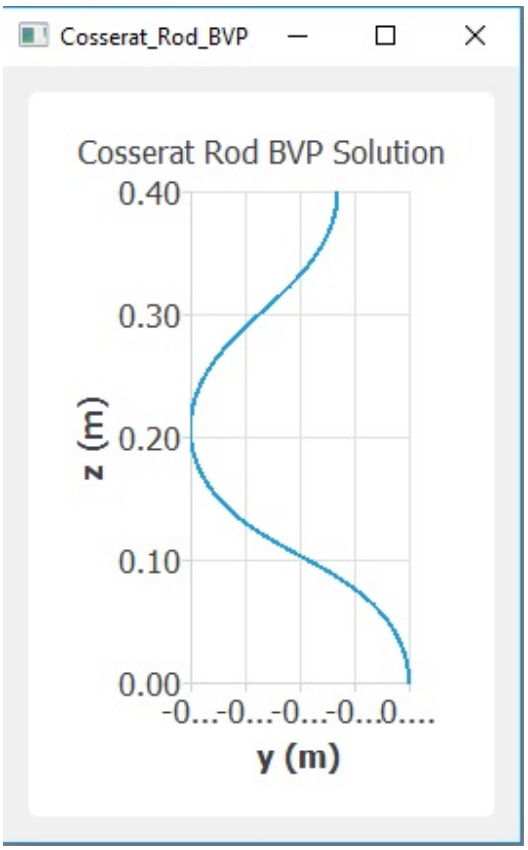
The $(\cdot)^\vee$ operator is the inverse of the hat $\widehat{\cdot}$ operator, and the "log" is a matrix logarithm.

Now we can solve the BVP by using a solver on the objective function. *Convex optimization* routines are defined in "convexoptimization.h", and we make a call to one of these algorithms:

```
int main(int, char**){
    Vector6d init_guess = Vector6d::Zero();

    //Solve with shooting method
    VectorXd wrench_soln = solveLevenbergMarquardt<shootingFunction>(init_guess);
```

The solver requires an initial guess, which we trivially supply as $\begin{bmatrix}0 & 0 & 0 & 0 & 0 & 0\end{bmatrix}^T$. The method used to find the correct base reaction wrench is the *Levenberg-Marquardt algorithm*, but while considering the rod BVP we can regard it as a blackbox solver. Now we have solved the problem and we can plot the result.



On a pedagogical note, there may not be a *unique* solution to a BVP, but if our convex optimizer finds a zero, we have found *a* solution. For instance, we would find a different solution if we set the initial guess to the following:

```
Vector6d init_guess = Vector6d::Zero();
init_guess << 0, 4, -26, 0, 2, 0;
```

The BVP fundamentally has multiple solutions, so this would be an issue even if we weren't using the shooting method. In the context of a larger project, we would need to make sure that the solution we find is appropriate.

Another note: it would be nice from a software design standpoint to always treat a convex optimization solver as a blackbox function. However, this is generally not possible since solvers are prone to diverge or settle on non-zero local minimums. It will likely be necessary to develop a working knowledge of *convex optimization* to understand a solver's behavior in more complicated use cases.