# Tendon Robot Statics Model Variants

In the last example we implemented a relatively simple tendon robot model where the backbone was modeled as a Cosserat rod and the tendons were routed in parallel with the backbone. There is a plethora of potential modifications to the model, and three potential variations are considered in different sections here.

## 1 Kirchhoff Rod Backbone

There are significant simplifications to the model if we neglect the shear and extension modes as described in "Statics and Dynamics of Continuum Robots With General Tendon Routing and External Loading". This results in $\boldsymbol{v} = \boldsymbol{e}_3$, which is the elastic rod problem as studied by Gustav Kirchhoff prior to the Cosserat brothers. The result for the tendon robot is that one only needs to solve a 3x3 linear system for $\dot{\boldsymbol{u}}$ instead of the 6x6 linear system considered previously for $\dot{\boldsymbol{v}}$ and $\dot{\boldsymbol{u}}$.

The state vector has $\boldsymbol{n}^b$ instead of $\boldsymbol{v}$, and the ODE function is changed accordingly:

```
void kirchhoffTendonRobotOde(VectorXd& y_s_out, VectorXd& y){
    //Unpack state vector
    Matrix3d R = Map<Matrix3d>(&y[3]);
    Vector3d nb = Map<Vector3d>(&y[12]);
    Vector3d u = Map<Vector3d>(&y[15]);
```

Of course calculations involving shear or extension are modified so that $\boldsymbol{v} = \boldsymbol{e}_3$. The ODEs are also changed so that there is only a 3x3 linear system as in Equation (17):

```
    //Pack state vector derivative
    Map<Vector3d> p_s(&y_s_out[0]);
    Map<Matrix3d> R_s(&y_s_out[3]);
    Map<Vector3d> nb_s(&y_s_out[12]);
    Map<Vector3d> u_s(&y_s_out[15]);

    //ODEs
    p_s = R.col(2);
    R_s = hat_postmultiply(R,u);
    u_s = H.inverse()*(-u.cross(Kbt*u) - Vector3d::UnitZ().cross(nb) - b);
    nb_s = -u.cross(nb) - G*u_s - a - transposeMultiply(R,rho*area*g);
}
```

An inverse is used instead of a linear solver because Eigen has specialized methods to take efficient inverses of small fixed-size matrices. We also make some minor changes to the objective function since $\boldsymbol{n}^b$ is a state variable instead of $\boldsymbol{v}$:

```
y0 << p0, Map<VectorXd>(Matrix3d(R0).data(), 9), guess;

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

//Find the internal forces in the backbone prior to the final plate
Vector3d nb = Y.block<3,1>(12,Y.cols()-1);
Vector3d uL = Y.block<3,1>(15,Y.cols()-1);
Vector3d mb = Kbt*uL;
```

If we rendered the solution, we would see no visible difference compared the Cosserat model.

## 2 Non-parallel Tendon Routing

The examples so far have assumed the tendons are routed parallel with the backbone. However, the model in "Statics and Dynamics of Continuum Robots With General Tendon Routing and External Loading" is formulated for any general routing path, and helical routing strategies are demonstrated. The Cosserat backbone model is altered to have a helical routing strategy here.

There are a multitude of options for how to represent the tendon routing functions. Lambda functions might be the most straightforward way, and if we were willing to implement our own numerical integration function, the most efficient way would be to precompute all the values at grid points. The approach here is somewhere in between; we'll calculate all the vectors at a given point on the backbone:

```
//Helical routing
const double offset = 0.01506;
void getRouting(double arclength, Vector3d* r, Vector3d* r_s, Vector3d* r_ss){
    const double f = 2*pi/L; //tendons make a full revolution

    double c = offset*cos(f*arclength);
    double s = offset*sin(f*arclength);
    double fc = f*c;
    double fs = f*s;
    double ffc = f*fc;
    double ffs = f*fs;

    r[0] = Vector3d(c, s, 0);
    r[1] = Vector3d(-s, c, 0);
    r[2] = Vector3d(-c, -s, 0);
    r[3] = Vector3d(s, -c, 0);

    r_s[0] = Vector3d(-fs, fc, 0);
    r_s[1] = Vector3d(-fc, -fs, 0);
    r_s[2] = Vector3d(fs, -fc, 0);
    r_s[3] = Vector3d(fc, fs, 0);

    r_ss[0] = Vector3d(-ffc, -ffs, 0);
    r_ss[1] = Vector3d(ffs, -ffc, 0);
    r_ss[2] = Vector3d(ffc, ffs, 0);
    r_ss[3] = Vector3d(-ffs, ffc, 0);
}
```

The tendons are still separated by 90°, but they are routed in helices around the backbone with a frequency "f". There are three arrays passed in having four vectors each. "r", "r_s", and "r_ss" are set based on the arc length. The ODE includes the derivatives of $\boldsymbol{r}$:

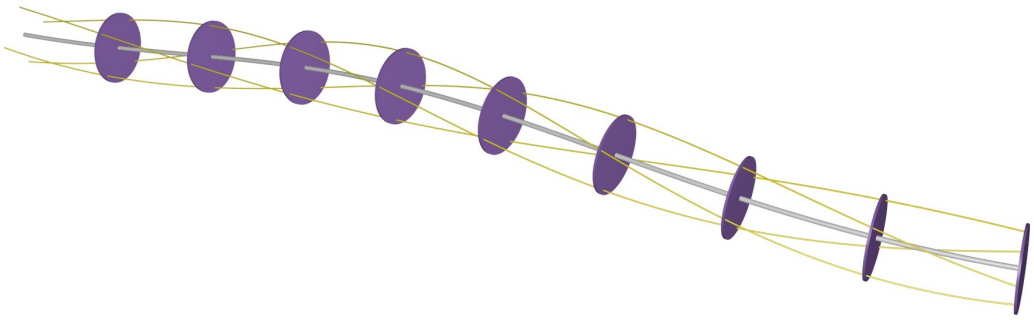
```
Vector3d r[num_tendons], r_s[num_tendons], r_ss[num_tendons];
getRouting(s, r, r_s, r_ss);

for(int i = 0; i < num_tendons; i++){
    Vector3d ri = r[i];
    Vector3d ri_s = r_s[i];
    Vector3d ri_ss = r_ss[i];

    Vector3d pb_si = u.cross(ri) + ri_s + v;
    double pb_s_norm = pb_si.norm();
    Matrix3d A_i = -hat_squared(pb_si)*(tau(i)/pow(pb_s_norm,3));
    Matrix3d G_i = -hat_postmultiply(A_i,ri);
    Vector3d a_i = A_i*(u.cross(pb_si + ri_s) + ri_ss);
```

The tendon routing vectors are calculated for the current arclength "s" before setting up the linear system. Over in the objective function, we also need to use the tendon routing vectors at

the tip when finding the equilibrium error:

```
//Find the equilibrium error at the tip, considering tendon forces
Vector3d force_error = -nb;
Vector3d moment_error = -mb;
Vector3d r[num_tendons], r_s[num_tendons], r_ss[num_tendons];
getRouting(L, r, r_s, r_ss);
for(int i = 0; i < num_tendons; i++){
    Vector3d pb_si = uL.cross(r[i]) + r_s[i] + vL;
    Vector3d Fb_i = -tau(i)*pb_si.normalized();
    force_error += Fb_i;
    moment_error += r[i].cross(Fb_i);
}
```

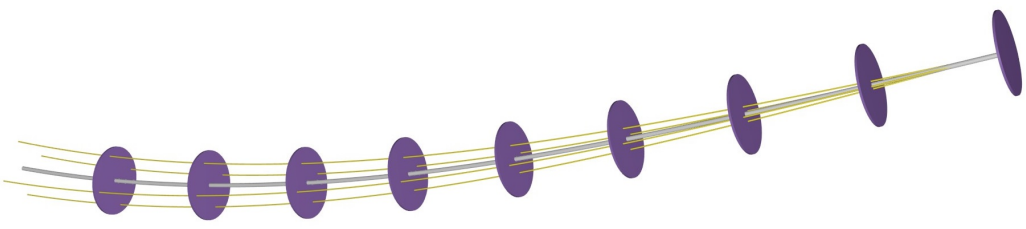
Then we can solve the problem and visualize the effects of helical routing:



We could also implement a converging tendon strategy like in the paper "Continuum Robot Stiffness Under External Loads and Prescribed Tendon Displacements". We revise the tendon function:

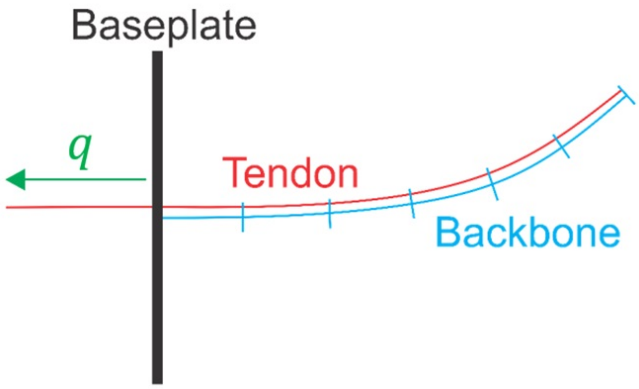
```
//Converging routing
const double offset = 0.01506;
const double routing_slope = -offset/L;
void getRouting(double s, Vector3d* r, Vector3d* r_s, Vector3d* r_ss){
    double b = (1 - s/L)*offset;

    r[0] = Vector3d(b, 0, 0);
    r[1] = Vector3d(0, b, 0);
    r[2] = Vector3d(-b, 0, 0);
    r[3] = Vector3d(0, -b, 0);

    r_s[0] = Vector3d(routing_slope, 0, 0);
    r_s[1] = Vector3d(0, routing_slope, 0);
    r_s[2] = Vector3d(-routing_slope, 0, 0);
    r_s[3] = Vector3d(0, -routing_slope, 0);

    r_ss[0] = r_ss[1] = r_ss[2] = r_ss[3] = Vector3d(0, 0, 0);
}
```

Then solving with converging routing yields the following solution:



# 3 Tendon Displacement Control

In the examples so far, the forward kinematics input has been the tendon tensions. However, most actuator control position instead of force. This scenario is also considered in the paper "Continuum Robot Stiffness Under External Loads and Prescribed Tendon Displacements". There are additional unknowns since the tendon tensions are not given, and additional constraints since the backbone must take a shape so that the tendon lengths are correct. For example, increasing the distance $q_i$ the tendon extends below the baseplate will cause the backbone to bend so that the arc length of the tendon above the baseplate is reduced, and the total tendon length remains constant (assuming an inextensible tendon).




We declare additional independent parameters for the displacement control:

```
const VectorXd q = Vector4d(5e-3, 0, -5e-3, 0);
const VectorXd l_star = Vector4d(L,L,L,L);
```

The length a tendon extends behind the baseplate is $q_i$, and the total length of a tendon before stretching is $l_i^*$. We're careful when setting the actuator displacements because it is especially easy to have an *ill-posed* problem. For instance, pulling all the tendons back by half the length of the backbone is simply not realistic, and the solver would probably fail to find a solution. The tendon tensions are now dependent variables, so they are declared as static:

```
static VectorXd tau;
```

These tensions will be guessed, so we change the guess initialization in the main function:

```
VectorXd init_guess = VectorXd::Zero(6 + num_tendons); //nb, u, and tau
```

and we unpack the last elements of the guess as the tendon tensions:

```
tau = guess.segment<num_tendons>(6);
```

Based on how the backbone deforms, each tendon will have some arc length it is routed along the backbone, with errors resulting from incorrect lengths. Calculating the errors requires modifying the ODE function to also integrate the tendon arc lengths. We extend the state vector so that the last entries are the tendon arc lengths:

```
Map<VectorXd> pb_s_norm(&y_s_out[18], num_tendons);
```

Now the change in tendon arc length with respect to the change in reference arc length is stored as part of the state vector derivative:

```
pb_s_norm(i) = pb_si.norm();
Matrix3d A_i = -hat_squared(pb_si)*(tau(i)/pow(pb_s_norm(i),3));
```

We modify the initial conditions to include the arc lengths of the tendons behind the base plate:

```
VectorXd y0(18+num_tendons);
y0 << p0, Map<VectorXd>(Matrix3d(R0).data(),9), v0, u0, q;
```

Then we have some error in the integrated lengths of the tendons versus the known lengths specified as design parameters:

```
//Find the length violation error
VectorXd integrated_lengths = Y.block<num_tendons,1>(18,Y.cols()-1);
VectorXd length_error = integrated_lengths - l_star;

VectorXd distal_error(6 + num_tendons);
distal_error << force_error, moment_error, length_error;
```

Now if we run the program, the robot is solved using tendon displacement as the forward kinematics input. However, our approach currently assumes the tendons are inextensible and capable of supporting compression. We address the former first by adding compliance as an independent variable:

```
const VectorXd C = Vector4d(1e-4,1e-4,1e-4,1e-4);
```

Then the length error is revised to include the tendon stretch, which is calculated from Hooke's law:

```
VectorXd stretch = l_star.cwiseProduct( C.cwiseProduct(tau) );
VectorXd length_error = integrated_lengths - (l_star + stretch);
```

Now the tendons will stretch under tension. Unfortunately, they will also contract under compression. To disallow compression, we include the effects of slack. Since slack and tension are mutually exclusive effects, we use a single variable to represent them both when we unpack the guess:

```
tau = guess.segment<num_tendons>(6).cwiseMax(0);
VectorXd slack = -(guess.segment<num_tendons>(6).cwiseMin(0));
```

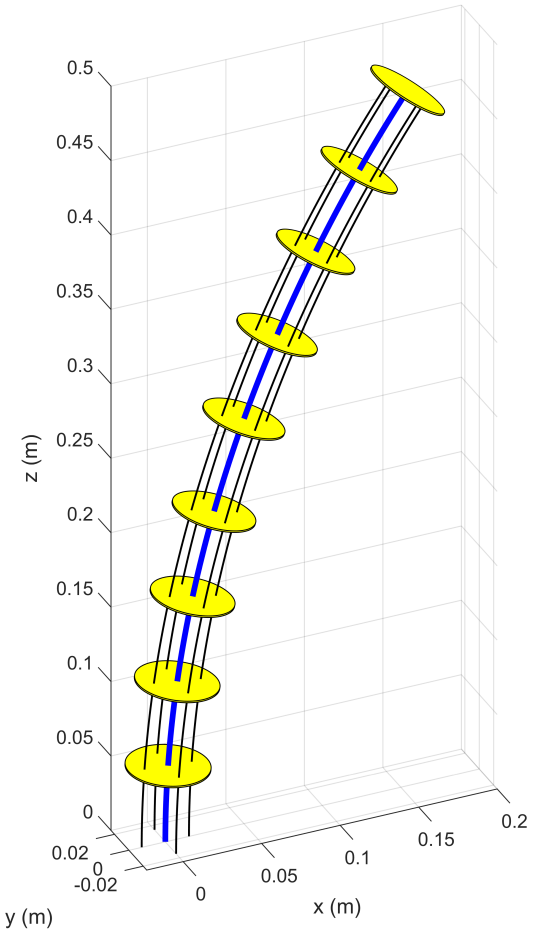
Depending on the sign, we have either tension or slack. Hopefully the shooting method optimization space is smooth around the transition; it seems to work well. Then the length error is revised to include slack:

```
VectorXd length_error = integrated_lengths + slack - (l_star + stretch);
```

Now we have a model with displacement controlled tendons which stretch under tension and go slack when necessary.

## 3.1 MATLAB example code

Thanks to Lisong Xu for contributing a MATLAB example of the displacement-controlled tendons, including a 3D visualisation.


0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
z (m)
0.02
0
-0.02
y (m)
0
0.05
0.1
0.15
0.2
x (m)