Simple Smooth Stepper Motor Motion Control

I.J. McEwan

Forward and History

The context and impetus for this project and white paper was a friend's custom 3D printed CNC tool which used a single stepper motor with a pulley and belt, similar to the configuration in many 3D printers, and a sprung loaded tool to perform a repetitive move-and-part tool motion. Unfortunately they weren't able to find a suitably sized stepper motor that was both powerful enough to perform the full motion flawlessly but small enough to not vibrate the printed chassis to pieces. The project also had a pretty underwhelming controller meaning any code solution needed to be very tight.
I always planed to revisit this and update it to also be able to use splines or NURBS, but ... intentions ...

Abstract

Stepper motors can produce very high torque, but without profiling and smooth control they can produce significant vibrations, and strain. Using piecewise polynomial profiles can greatly elevate these problems without incurring much additional overhead. Here we provided a simple profiling and code generation system to quickly design programable position, velocity, and torque profiles for a simple stepper-pulley-belt-spring system.

1 Requirements

The final implementation needs to meet the following requirements :

1. Very quick end to end motor movements.
2. Minimize vibrations and tool wear.
3. Smooth velocities and at least piecewise continuous accelerations.
4. Feed forward control only, feed back is limited to position calibration points such as end-stop detectors.
5. Creation of motor torque/speed profiles to help in motor selection.

2 Profiles - Discussion and code description

Requirement 3 can be met using functions that be repeatably differentiated and integrate nicely. So for this I have chosen simple polynomials with rational coefficients, as they are particularly easy to work with. Specifically : python handles both rationals and polynomials with easy; polynomials are easy to implement in the generated C code; and should meat all the requirements listed. Alternatives such as splines or NURBS (Non-Uniform Rational B-Splines) are a possible up-scope or expansion for later.
The simulation and generation code executes as follows: First it reads a configuration file for the polynomials (outlined in Section 3, and plotted in Table 1), the desired movement profile (e.g. Listing 1 and Listing 3) and any motor data provided for comparison [`generate.py:418`].
From this it computes various timing zones, and linearly transforms the requested polynomial so that the overall distance and velocity functions are smooth and the acceleration function is at least piecewise continuous on these zone boundaries; and the expected accelerations, forces, velocities, and displacements versus time [`generate.py:425`].
Next, it plots the various time vs displacement, velocity, acceleration and, force functions (seen in section 6), as well as the torque vs motor speed plots for component comparison and selection [`generate.py:429`].
The use of polynomials make this straight forward. For example computing the overall force function at any point is given by:

$$\begin{aligned} F(t) &= F_{mass}(t) + F_{spring}(t) \\ &= m\ddot{x}(t) + kx(t) + F_{x_0} \end{aligned}$$

where m is the mass, k is the spring constant, F_{x_0} is the initial spring force, and $\ddot{x}$ and x are the existing acceleration and displacement polynomials. The result is just another polynomial.
Similarly the torque and motor speed functions are just scaled (by the radius of the pulley R_{pulley}) versions of

the force and velocity functions.

$$\tau(t) = R_{pulley} F(t)$$
$$\omega(t) = \frac{1}{2\pi R_{pulley}} \dot{x}(t)$$

As are the displacement and motor speeds in 'step' units :

$$X(t) = \frac{N}{2\pi R_{pulley}} x(t)$$
$$\Omega(t) = N\omega(t)$$

Where N is the number of steps per revolution of the stepper motor.
It also calculates the points in time that correspond with any calibration points, such as the location of an end-stop switch or optical marker, given as displacements. i.e. it solves for t in $x(t) = x_{cal}$ for all "calpoint.name" values in the given profile.
The generator then builds the needed C functions using its computed scale factors and the initial polynomials (not the derived polynomials - this is a time/space trade off, one additional multiply saves an explosion of several polynomials for every zone). Also the resultant constants are fairly nice rational numbers which would make porting this to a non-fpu, integer system much easier [`generate.py:440`].
Lastly it builds the plots and resultant data into the reports shown in Section 3 [`generate.py:444`].

2.1 The generated code

The generated motor driving functions (e.g. Listings 2 and 4) make use of the way a stepper motor is usually driven, that is, with two signals : a direction bit and a clock-like step bit, so the frequency of the step bit is proportional to the speed of the motor.
If we compute the desired displacement of the motor at time steps $t_i, t_{i+1}, t_{i+2}, \ldots$, rounded to the nearest step, and at a high enough frequency, then the lowest bit of those computed displacement will clock at exactly the correct frequency for a stepper motor to follow the programmed motion profile. The direction can be found by differencing the displacements.
This gives the following relationships at time t_i for the clock and direction signals:

$$x_i = X(t_i)$$
$$\mathrm{clk}_i = \mathrm{floor}(2x_i) \,\&1$$
$$\mathrm{dir}_i = \mathrm{sign}(x_i - x_{i-1})$$

Where $X()$ is the displacement profile as a function of time given above, and &1 denotes a 'bit-wise and' with the lowest bit.
This method has several advantages : It's simple; the CPU load is constant and deterministic; Multiple motors can be computed on the same timer.
Its major disadvantage though is that the jitter in pulse timing is proportional to the timer's frequency. Higher frequency gives less jitter, at the expense of higher (but still deterministic) load. (see Section 4.3).
Note: There are other ways to do this such as solved for t_{i+1} in $X(t_{i+1}) = X(t_i) + 1$. This has the advantage of very high velocity fidelity, and maximizes the time between outputs, at the cost of an somewhat non-deterministic iterative solve.

2.2 Simulations

Section 6 shows two simulated profiles and the C code generated for them.
The first profile moves the sprung loaded parting tool out of the way for four seconds and then back in three times to part and clean before restarting the motion and moving the tool out of the way again.
The second profile is the same but slightly more aggressive with a small 'wobble' overlayed to reduce the stall load on the stepper motor.
The C code generated has function of the form `<name>TimerTick()` as entry and assumes it can call a function `motorOutput()` to set the stepper motors control bit pair.

3 Base Polynomials Functions

The simulation and generation code reads in a polynomial description file that contains the roots, sign and normalization points for each named velocity polynomial set. This is really the unscaled acceleration profile, but it is better to think of it as the locations of extrema in the velocity polynomial, as it is the velocity polynomial that is the starting point for everything else. (In fact the scaled acceleration profile is obtain cheaply by differentiation later, and this poly is thrown away after initialization). The velocity profile is found by integrating and normalizing these roots.
For example the 'niceTop' profile [`polynomials.dat:18`] is given as :

```
niceCurve -1.0 auto 1.0 0.0 -1.0
```

where the first column is the polynomials name, the second and third column specify the input variable (x locations) to use to normalize the output (y values) with all subsequent columns being roots of of the polynomial. The the base velocity polynomial is found as :

$$f(t) = \int_{f(-1)=0}^{\max(f(x))=1} (t+1)(t)(t-1)\,dt$$

Table 1 shows the polynomials available for the example profiles in Section 6.

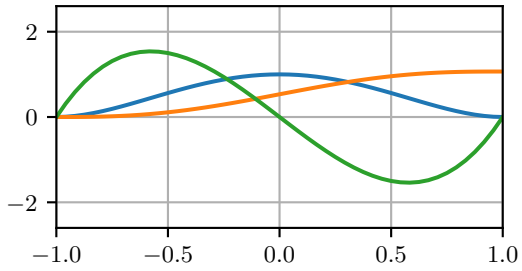
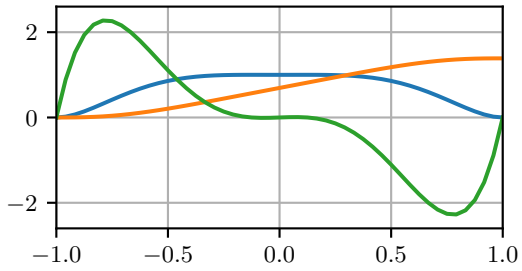
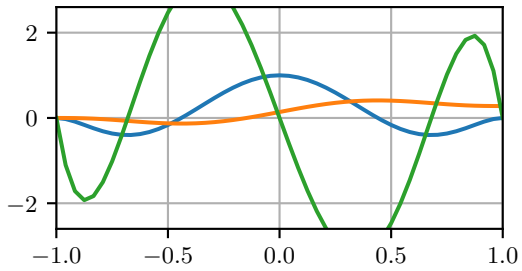
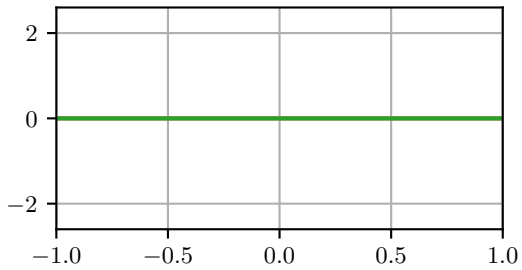
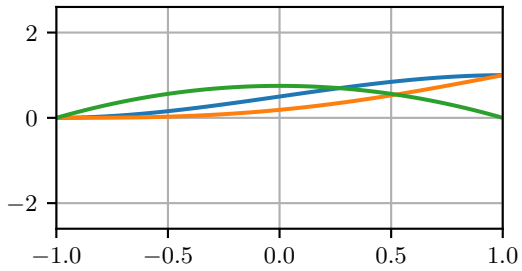
Polynomial "niceCurve"		$x(t) = \frac{1}{5}t^5 - \frac{2}{3}t^3 + t + \frac{8}{15}$ $\dot{x}(t) = t^4 - 2t^2 + 1$ $\ddot{x}(t) = 4t^3 - 4t$
Polynomial "flatTop"		$x(t) = \frac{128}{427}t^7 - \frac{39}{61}t^5 + \frac{2}{61}t^3 + t + \frac{296}{427}$ $\dot{x}(t) = \frac{128}{61}t^6 - \frac{195}{61}t^4 + \frac{6}{61}t^2 + 1$ $\ddot{x}(t) = \frac{768}{61}t^5 - \frac{780}{61}t^3 + \frac{12}{61}t$
Polynomial "wobble"		$x(t) = -\frac{625}{847}t^7 + \frac{1371}{605}t^5 - \frac{289}{121}t^3 + t + \frac{592}{4235}$ $\dot{x}(t) = -\frac{625}{121}t^6 + \frac{1371}{121}t^4 - \frac{867}{121}t^2 + 1$ $\ddot{x}(t) = -\frac{3750}{121}t^5 + \frac{5484}{121}t^3 - \frac{1734}{121}t$
Polynomial "stationary"		$x(t) = 0$ $\dot{x}(t) = 0$ $\ddot{x}(t) = 0$
Polynomial "oneSide"		$x(t) = -\frac{1}{16}t^4 + \frac{3}{8}t^2 + \frac{1}{2}t + \frac{3}{16}$ $\dot{x}(t) = -\frac{1}{4}t^3 + \frac{3}{4}t + \frac{1}{2}$ $\ddot{x}(t) = -\frac{3}{4}t^2 + \frac{3}{4}$

Table 1: Various choices of polynomials.

823DCADED5E2D75CFBC2B7086DB4A19E
December 27, 2022

4 Comments and Thoughts.

4.1 Unfinished Todos

- Change the argument handling so it can take more than one motor command line argument, the code is already there to plot several lines, and get the caption correct (with correct color names).
- Break out code template into its own file.
- Run the generated C code in a harness, read back in and plot the resultant output for comparison, and analyzing things like timing jitter.

4.2 Other thoughts looking back at the code.

- Error reporting sucks: Many more erroneous conditions could be checked and reported, and the current errors are not exactly self explanatory!
- The code really ought to be refactored to be more modular and work with other physical configurations. It's a need-to-get-this-finished mess.

4.3 Extensions, and further work

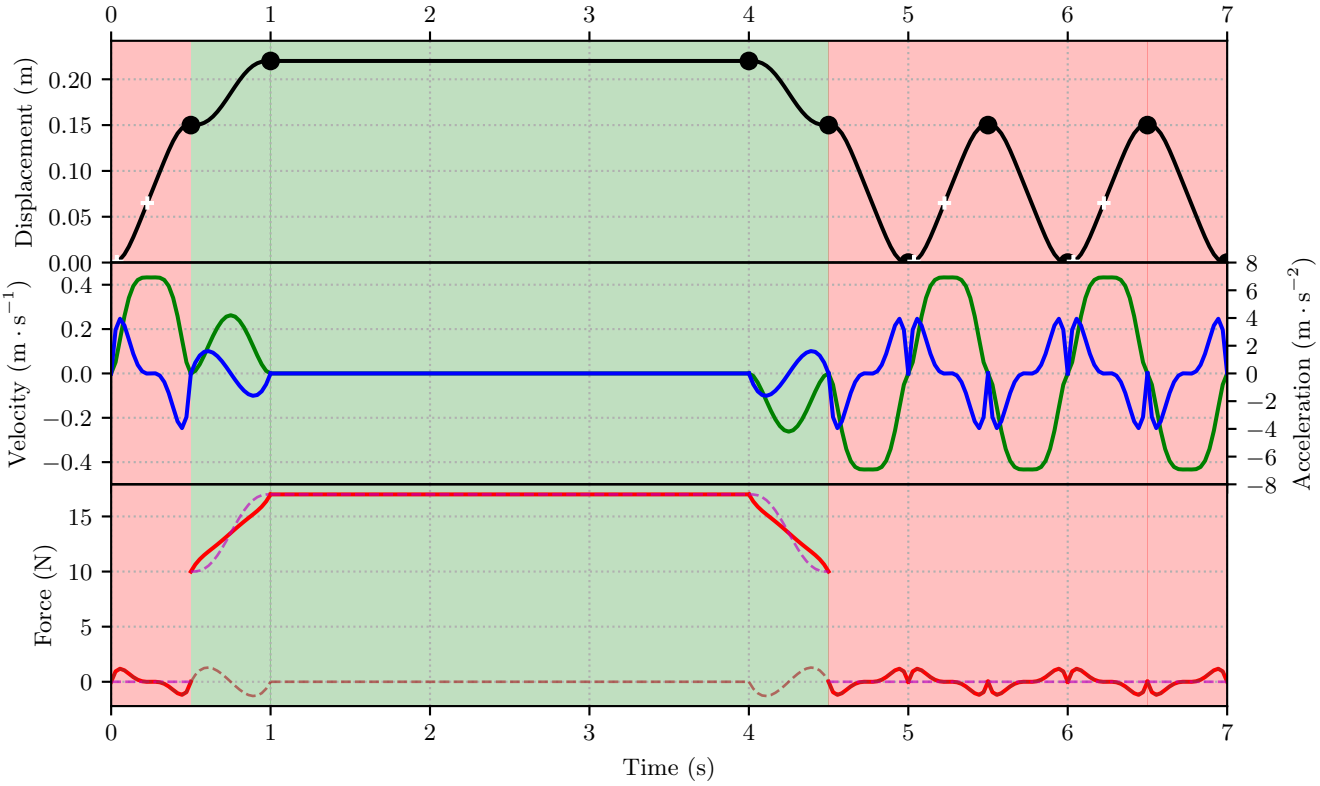
- Non-zero velocity on zone boundaries: The polynomials used are limited so that the velocity and acceleration of each zone starts at zero and returns to zero. The code already contains the bits necessary to allow velocities to be non-zero at the ends (just as position currently is) but that offset is not tracked through the profiles. Adding this would allow, for example, the motor to reverse direction under constant acceleration, rather than the current slow to a smooth stop, then accelerate in the other direction.
- Multiple motors: The code only generates one motor profile at a time. It would not be hard to allow multiple motors to controlled at the same time (by extending each line in the .in file).
- Schedule integration: The code just provides a 'tick' function to call, without regard for how the scheduler wants to do things.
- Better timing: The jitter in position and velocity of the generated code could potentially be improved in two ways,
  - When the velocity is slow, the micro-stepping fraction could be changed dynamically and to have more steps and thus higher output frequency, without effecting the timer's frequency.
  - The timer's frequency could be dynamically set to always be some multiple of the current velocity. With care taken for zero and maximum velocities.

5 Conclusion

This project was intended to solve the vibration and profiling problem of a specific physical device, but the ideas and code would be easily converted to an stepper motor based repetitive motion control project.

6 Examples

6.1 Movement Profile: profile1



The above profile shows the displacement (black), velocity (green), and acceleration (blue) of the actuator, and the force needed to move the actuator alone (brown), the force from the sprint (magenta), and the overall force (red). Inputed zone boundries are marked with black circles, and any calibration points with white pluses.
The maximum torque is 0.145 N·m and the maximum velocity is 7.3 s^{-1}.
This profile has 2 zones, uses a 0.019 m pully, and a base mass of 0.3 kg :

	Name	Color	Spring (N·m)	Anchor (m)	Friction	Extra M (kg)
0	clear	red	0.0	0.0	0.0	0.0
1	spring	green	100.0	0.05	0.0	0.5

Listed calibrations points are :

Name	Disp.	Times
opt1	0.065	0.2269, 5.227, 6.227
endstop	0.001	0.0356, 5.036, 6.036

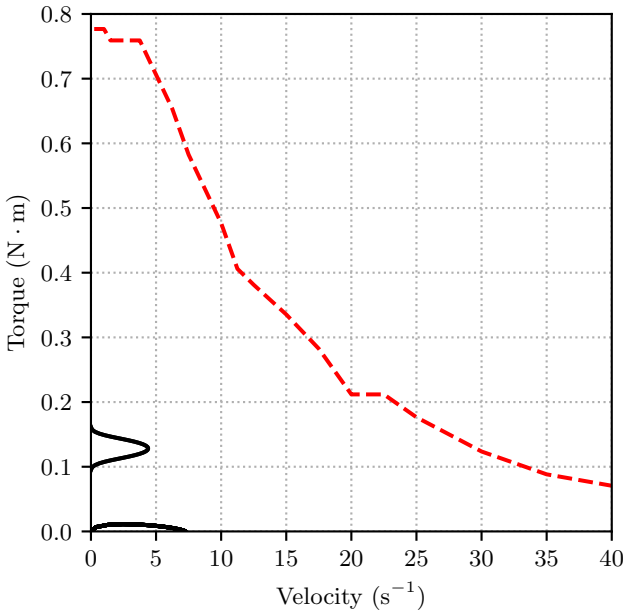


Figure 1: Predicted torque profile (in black) plotted with motor[s]: 17Y408S (red).

823DCADED5E2D75CFBC2B7086DB4A19E
December 27, 2022

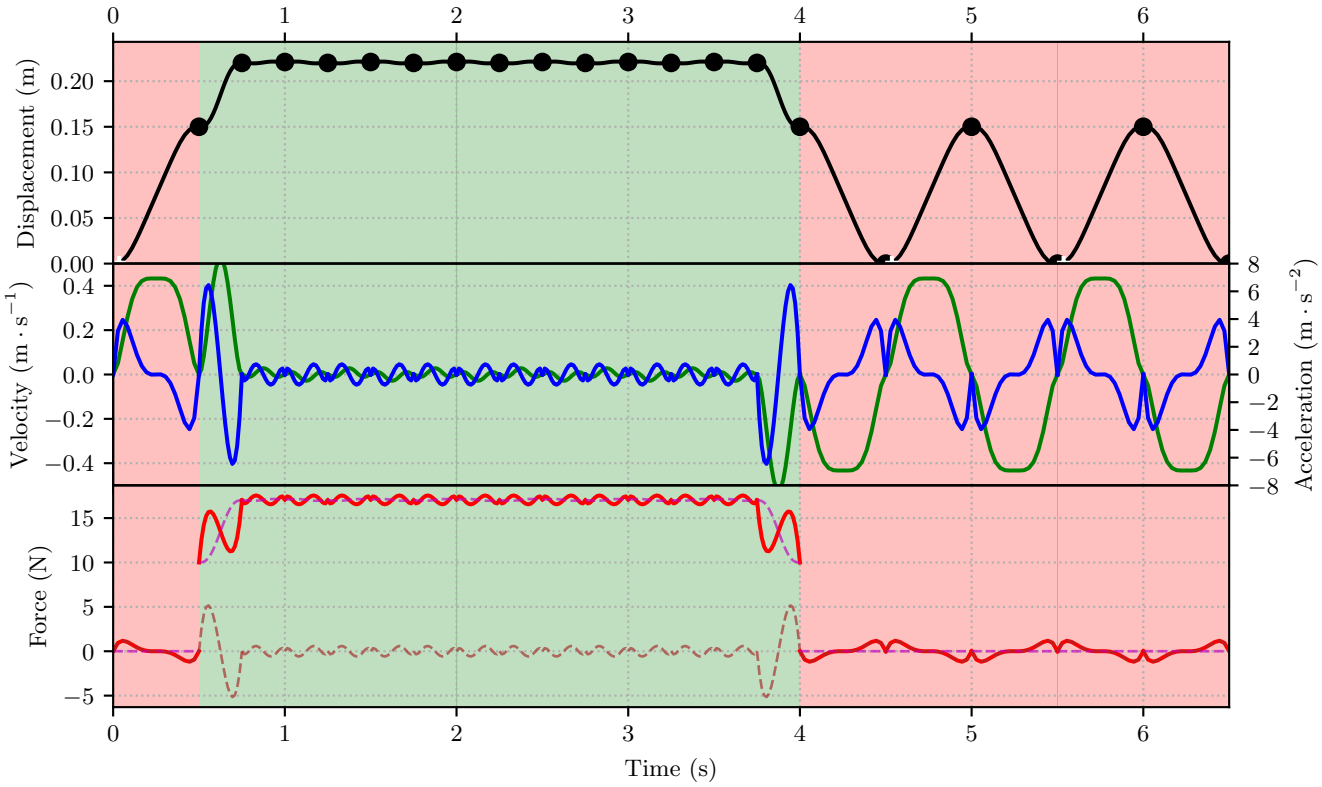
```
# Units m, kg, s

# Global Constants
global.baseMass    0.3
global.smallestDt  0.5

global.pullyDia    0.019
global.Nsteps      200

# Constants for out of the way environment
clear.springK    0
clear.springE0   0
clear.friction   0
clear.extraMass  0

# Constants for spring loaded environment
# Note springE0 set as if the spring isn't confined to the spring domain.
spring.springK    100.0
spring.springE0   0.050
spring.friction   0
spring.extraMass  0.5

# Calibration points are a set of displacements to find the corrisponding
# times for.
calpoint.opt1      0.065
calpoint.endstop   0.001

# Columns are :
# Delta x, Delta v, Delta t, Profile, Environment, Description

# Move tool out of the way.
 0.150   0.0   0.5   flatTop      clear   Part In

# Clear in 0.5 seconds, wait 3 and return in 0.5
 0.070   0.0   0.5   niceCurve    spring   Tool Out
 0.0     0.0   3.0   stationary   spring   Tool Wait
-0.070   0.0   0.5   niceCurve    spring   Tool In

# End of first part motion
-0.150   0.0   0.5   flatTop      clear    Part Out

# Two more parting motions
 0.150   0.0   0.5   flatTop      clear    Part In
-0.150   0.0   0.5   flatTop      clear    Part Out
 0.150   0.0   0.5   flatTop      clear    Part In
-0.150   0.0   0.5   flatTop      clear    Part Out
```

Listing 1: input Profile filefor profile1

```
1
2  inline int clampint(const int x,const int x0,const int x1) { return x<x0 ? x0 : (x>x1 ? x1 : x); }
3
4  enum { N = 5, M = 5 };
5  typedef struct {
6    float c0, c1, alpha;
7    int pN;
8    } zone_t;
9
10 typedef float Marray_t[M];
11
12 static const Marray_t polynomialXCoefs[N] = {
13   {8.0/15.0, 1.0/1.0, 0.0/1.0, -2.0/3.0, 0.0/1.0} /* niceCurve */,
14   {296.0/427.0, 1.0/1.0, 0.0/1.0, 2.0/61.0, 0.0/1.0} /* flatTop */,
15   {592.0/4235.0, 1.0/1.0, 0.0/1.0, -289.0/121.0, 0.0/1.0} /* wobble */,
16   {0.0/1.0, 0.0/1.0, 0.0/1.0, 0.0/1.0, 0.0/1.0} /* stationary */,
17   {3.0/16.0, 1.0/2.0, 3.0/8.0, 0.0/1.0, -1.0/16.0} /* oneSide */
18     };
19
20 static const Marray_t polynomialDXCoefs[N] = {
21   {1.0/1.0, 0.0/1.0, -2.0/1.0, 0.0/1.0, 1.0/1.0} /* niceCurve */,
22   {1.0/1.0, 0.0/1.0, 6.0/61.0, 0.0/1.0, -195.0/61.0} /* flatTop */,
23   {1.0/1.0, 0.0/1.0, -867.0/121.0, 0.0/1.0, 1371.0/121.0} /* wobble */,
24   {0.0/1.0, 0.0/1.0, 0.0/1.0, 0.0/1.0, 0.0/1.0} /* stationary */,
25   {1.0/2.0, 3.0/4.0, 0.0/1.0, -1.0/4.0, 0.0/1.0} /* oneSide */
26     };
27
28 static const float profile1Dt = 0.5;
29 zone_t profile1TimeToZone(float t)
30   {
31   enum { L=9, K=14 };
32
33   static const zone_t zones[L] = {
34   { -1.0/1.0, 4.0/1.0, 0.4327702702702702, 1 } /* flatTop: (0.00 < t < 0.50) */,
35   { -3.0/1.0, 4.0/1.0, 0.26249999999999996, 0 } /* niceCurve: (0.50 < t < 1.00) */,
36   { -5.0/3.0, 2.0/3.0, 0.0, 3 } /* stationary: (1.00 < t < 4.00) */,
37   { -17.0/1.0, 4.0/1.0, -0.26249999999999996, 0 } /* niceCurve: (4.00 < t < 4.50) */,
38   { -19.0/1.0, 4.0/1.0, -0.4327702702702702, 1 } /* flatTop: (4.50 < t < 5.00) */,
39   { -21.0/1.0, 4.0/1.0, 0.4327702702702702, 1 } /* flatTop: (5.00 < t < 5.50) */,
40   { -23.0/1.0, 4.0/1.0, -0.4327702702702702, 1 } /* flatTop: (5.50 < t < 6.00) */,
41   { -25.0/1.0, 4.0/1.0, 0.4327702702702702, 1 } /* flatTop: (6.00 < t < 6.50) */,
42   { -27.0/1.0, 4.0/1.0, -0.4327702702702702, 1 } /* flatTop: (6.50 < t < 7.00) */
43   };
44
45   static const int timeInx[K] = { 0, 1, 2, 2, 2, 2, 2, 2, 3, 4, 5, 6, 7, 8 };
46
47   return zones[ timeInx[ clampint(t/profile1Dt, 0, L-1) ] ];
48   }
49
50 float calcPoly(const Marray_t *p, const zone_t z, float t)
51   {
52   const float *c = p[ z.pN ];
53   float s = z.c0 + z.c1 * t ;
54   return z.alpha * (((((c[4])*s + c[3])*s + c[2])*s + c[1])*s + c[0]);
55   }
56
57 float profile1TimeToDisp(float t)
58   {
59   return calcPoly( polynomialXCoefs, profile1TimeToZone(t), t );
60   }
61
62 float profile1TimeToVel(float t)
63   {
64   return calcPoly( polynomialDXCoefs, profile1TimeToZone(t), t );
65   }
66
67 float xLast, tNow;
68
69 void profile1Reset()
70   {
71   xLast = 0.0;
72   tNow  = 0.0; /* could also be a tStart = getNow() function */
73   }
74
75 void profile1TimerTick()
76   {
77   float xNow;
78
79   static const float kMotor = 6701.260761764014;
80
81   tNow += profile1Dt; /* could also be tNow = getNow() - tStart */
82
83   xNow = kMotor * profile1TimeToDisp(tNow) ;
84   motorOutput(  (int)(xNow) & 1, (xNow-xLast < 0) );
85
86   xLast = xNow ;
87   }
```

Listing 2: Generated C Code for profile1

6.2 Movement Profile: profileWobble



The above profile shows the displacement (black), velocity (green), and acceleration (blue) of the actuator, and the force needed to move the actuator alone (brown), the force from the sprint (magenta), and the overall force (red). Inputed zone boundries are marked with black circles, and any calibration points with white pluses.

The maximum torque is 0.167 N·m and the maximum velocity is 7.3 s^{-1}.

This profile has 2 zones, uses a 0.019 m pully, and a base mass of 0.3 kg :

	Name	Color	Spring (N·m)	Anchor (m)	Friction	Extra M (kg)
0	clear	red	0.0	0.0	0.0	0.0
1	spring	green	100.0	0.05	0.0	0.5

Listed calibrations points are :

Name	Disp.	Times
endstop	0.001	0.0356, 4.536, 5.536

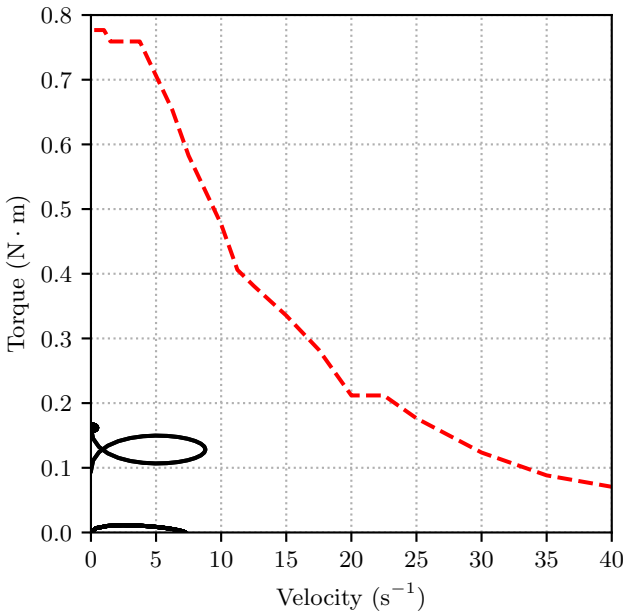


Figure 2: Predicted torque profile (in black) plotted with motor[s]: 17Y408S (red).

```
# Units m, kg, s

# Global Constants
global.baseMass   0.3
global.smallestDt 0.25

global.pullyDia 0.019
global.Nsteps    200

# Constants for out of the way environment
clear.springK   0
clear.springE0  0
clear.friction  0
clear.extraMass 0

# Constants for spring environment
# Note springE0 set as if the spring isn't confined to the spring domain.
spring.springK   100.0
spring.springE0  0.050
spring.friction  0
spring.extraMass 0.5

# Calibration points are a set of displacements to find the corrisponding
# times for.
calpoint.endstop  0.001

# Columns are :
# Delta x, Delta v, Delta t, Profile, Environment, Description

# Move tool out of the way.
 0.150  0.0  0.5  flatTop     clear  Part In

# Clear in 0.5 seconds, wait 3 and return in 0.5
 0.070  0.0  0.25  niceCurve   spring  Tool Out
 0.001  0.0  0.25   wobble      spring  Tool Wait
-0.001  0.0  0.25   wobble      spring  Tool Wait
 0.001  0.0  0.25   wobble      spring  Tool Wait
-0.001  0.0  0.25   wobble      spring  Tool Wait
 0.001  0.0  0.25   wobble      spring  Tool Wait
-0.001  0.0  0.25   wobble      spring  Tool Wait
 0.001  0.0  0.25   wobble      spring  Tool Wait
-0.001  0.0  0.25   wobble      spring  Tool Wait
 0.001  0.0  0.25   wobble      spring  Tool Wait
-0.001  0.0  0.25   wobble      spring  Tool Wait
 0.001  0.0  0.25   wobble      spring  Tool Wait
-0.001  0.0  0.25   wobble      spring  Tool Wait
-0.070  0.0  0.25  niceCurve   spring  Tool In

# End of first part motion
-0.150  0.0  0.5  flatTop     clear   Part Out

# Two more parting motions
 0.150  0.0  0.5  flatTop     clear   Part In
-0.150  0.0  0.5  flatTop     clear   Part Out
 0.150  0.0  0.5  flatTop     clear   Part In
-0.150  0.0  0.5  flatTop     clear   Part Out
```

Listing 3: input Profile filefor profileWobble

```
1
2  inline int clampint(const int x,const int x0,const int x1) { return x<x0 ? x0 : (x>x1 ? x1 : x); }
3
4  enum { N = 5, M = 5 };
5  typedef struct {
6    float c0, c1, alpha;
7    int pN;
8    } zone_t;
9
10 typedef float Marray_t[M];
11
12 static const Marray_t polynomialXCoefs[N] = {
13   {8.0/15.0, 1.0/1.0, 0.0/1.0, -2.0/3.0, 0.0/1.0} /* niceCurve */,
14   {296.0/427.0, 1.0/1.0, 0.0/1.0, 2.0/61.0, 0.0/1.0} /* flatTop */,
15   {592.0/4235.0, 1.0/1.0, 0.0/1.0, -289.0/121.0, 0.0/1.0} /* wobble */,
16   {0.0/1.0, 0.0/1.0, 0.0/1.0, 0.0/1.0, 0.0/1.0} /* stationary */,
17   {3.0/16.0, 1.0/2.0, 3.0/8.0, 0.0/1.0, -1.0/16.0} /* oneSide */
18     };
19
20 static const Marray_t polynomialDXCoefs[N] = {
21   {1.0/1.0, 0.0/1.0, -2.0/1.0, 0.0/1.0, 1.0/1.0} /* niceCurve */,
22   {1.0/1.0, 0.0/1.0, 6.0/61.0, 0.0/1.0, -195.0/61.0} /* flatTop */,
23   {1.0/1.0, 0.0/1.0, -867.0/121.0, 0.0/1.0, 1371.0/121.0} /* wobble */,
24   {0.0/1.0, 0.0/1.0, 0.0/1.0, 0.0/1.0, 0.0/1.0} /* stationary */,
25   {1.0/2.0, 3.0/4.0, 0.0/1.0, -1.0/4.0, 0.0/1.0} /* oneSide */
26     };
27
28 static const float profileWobbleDt = 0.25;
29 zone_t profileWobbleTimeToZone(float t)
30   {
31   enum { L=20, K=26 };
32
33   static const zone_t zones[L] = {
34   { -1.0/1.0, 4.0/1.0, 0.4327702702702702, 1 } /* flatTop: (0.00 < t < 0.50) */,
35   { -5.0/1.0, 8.0/1.0, 0.5249999999999999, 0 } /* niceCurve: (0.50 < t < 0.75) */,
36   { -7.0/1.0, 8.0/1.0, 0.028614864864864965, 2 } /* wobble: (0.75 < t < 1.00) */,
37   { -9.0/1.0, 8.0/1.0, -0.028614864864864965, 2 } /* wobble: (1.00 < t < 1.25) */,
38   { -11.0/1.0, 8.0/1.0, 0.028614864864864965, 2 } /* wobble: (1.25 < t < 1.50) */,
39   { -13.0/1.0, 8.0/1.0, -0.028614864864864965, 2 } /* wobble: (1.50 < t < 1.75) */,
40   { -15.0/1.0, 8.0/1.0, 0.028614864864864965, 2 } /* wobble: (1.75 < t < 2.00) */,
41   { -17.0/1.0, 8.0/1.0, -0.028614864864864965, 2 } /* wobble: (2.00 < t < 2.25) */,
42   { -19.0/1.0, 8.0/1.0, 0.028614864864864965, 2 } /* wobble: (2.25 < t < 2.50) */,
43   { -21.0/1.0, 8.0/1.0, -0.028614864864864965, 2 } /* wobble: (2.50 < t < 2.75) */,
44   { -23.0/1.0, 8.0/1.0, 0.028614864864864965, 2 } /* wobble: (2.75 < t < 3.00) */,
45   { -25.0/1.0, 8.0/1.0, -0.028614864864864965, 2 } /* wobble: (3.00 < t < 3.25) */,
46   { -27.0/1.0, 8.0/1.0, 0.028614864864864965, 2 } /* wobble: (3.25 < t < 3.50) */,
47   { -29.0/1.0, 8.0/1.0, -0.028614864864864965, 2 } /* wobble: (3.50 < t < 3.75) */,
48   { -31.0/1.0, 8.0/1.0, -0.5249999999999999, 0 } /* niceCurve: (3.75 < t < 4.00) */,
49   { -17.0/1.0, 4.0/1.0, -0.4327702702702702, 1 } /* flatTop: (4.00 < t < 4.50) */,
50   { -19.0/1.0, 4.0/1.0, 0.4327702702702702, 1 } /* flatTop: (4.50 < t < 5.00) */,
51   { -21.0/1.0, 4.0/1.0, -0.4327702702702702, 1 } /* flatTop: (5.00 < t < 5.50) */,
52   { -23.0/1.0, 4.0/1.0, 0.4327702702702702, 1 } /* flatTop: (5.50 < t < 6.00) */,
53   { -25.0/1.0, 4.0/1.0, -0.4327702702702702, 1 } /* flatTop: (6.00 < t < 6.50) */
54   };
55
56   static const int timeInx[K] = { 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 15, 16, 16, 17, 17, 18, 18,
       19, 19 };
57
58   return zones[ timeInx[ clampint(t/profileWobbleDt, 0, L-1) ] ];
59   }
60
61 float calcPoly(const Marray_t *p, const zone_t z, float t)
62   {
63   const float *c = p[ z.pN ];
64   float s = z.c0 + z.c1 * t ;
65   return z.alpha * (((((c[4])*s + c[3])*s + c[2])*s + c[1])*s + c[0]);
66   }
67
68 float profileWobbleTimeToDisp(float t)
69   {
70   return calcPoly( polynomialXCoefs, profileWobbleTimeToZone(t), t );
71   }
72
73 float profileWobbleTimeToVel(float t)
74   {
75   return calcPoly( polynomialDXCoefs, profileWobbleTimeToZone(t), t );
76   }
77
78 float xLast, tNow;
79
80 void profileWobbleReset()
81   {
82   xLast = 0.0;
83   tNow  = 0.0; /* could also be a tStart = getNow() function */
84   }
85
86 void profileWobbleTimerTick()
87   {
88   float xNow;
89
90   static const float kMotor = 6701.260761764014;
91
92   tNow += profileWobbleDt; /* could also be tNow = getNow() - tStart */
93
94   xNow = kMotor * profileWobbleTimeToDisp(tNow) ;
95   motorOutput(  (int)(xNow) & 1, (xNow-xLast < 0) );
96
97   xLast = xNow ;
98   }
```

Listing 4: Generated C Code for profileWobble