Buckling Link and Spring System

Katharin Jensen

Contents

Section 1. Introduction

Purpose

This document describes systems of springs and buckling links. These systems are developed to demonstrate the effects of aleatory uncertainty in system morphology with simple, easy to understand models. The application of novel binning techniques to reduce computation time is also described.

Problem

The problem is to write a code that can apply the equations of statics and dynamics to solve a system of buckling links and springs

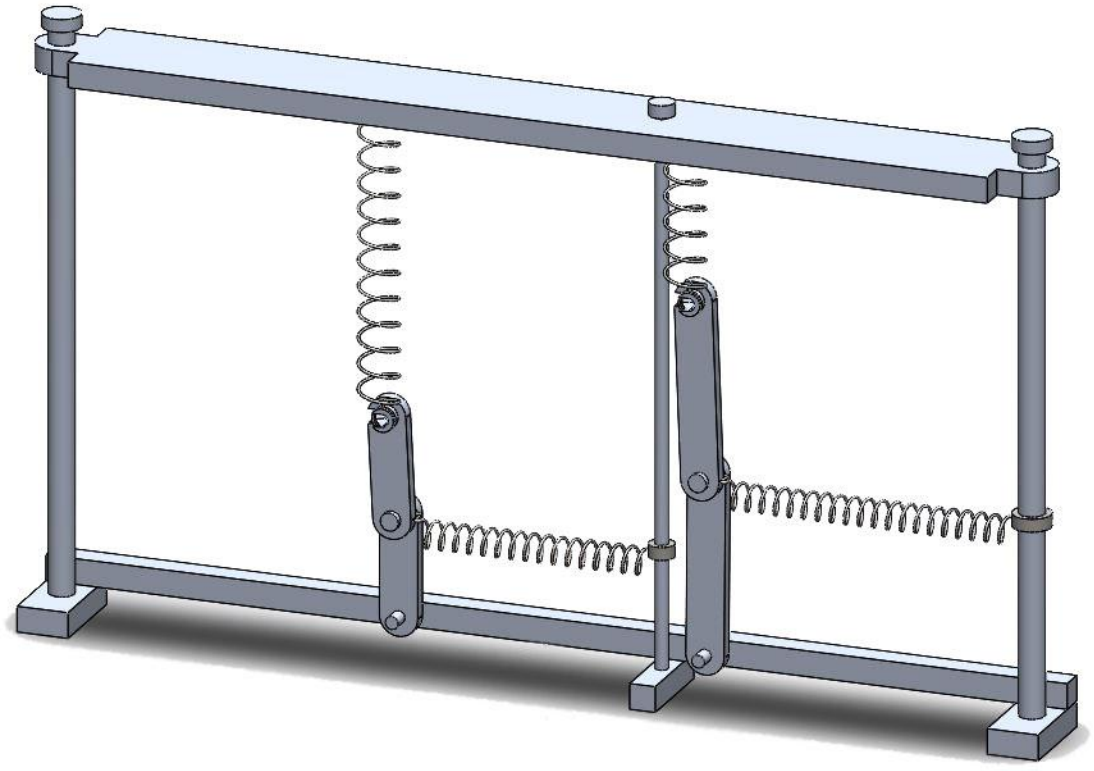


Figure 1: A CAD drawing of the setup of the system

The system consists of a number of these buckling units arranged in parallel with each other. As shown in Figure 1, each unit is made up of two rigid links, pinned together so that they can rotate in plane. One end of the connected links is pinned to a base, and the other end is pinned to a non-linear spring. The non-

linear spring is treated like a bar that is unable to buckle, so that its spring constant is EA/h, where E is the Young's Modulus for the material, A is its cross-sectional area, and h is the length of the bar. The specifics of what E and A are aren't important to the model, just the value of EA, so for the remainder of this document EA is referred to as one entity, considered to be a spring constant-like quantity for the top spring. This spring is vertical, and its other end is attached to a plate, upon which the force is applied. There is another, linear spring attached to the joint between the links. It is horizontal, and its other end is attached to a part which allows the spring to move vertically with the links and thus stay completely horizontal. The top of the joined links – where the spring is attached – can only move vertically, as can the plate where the force is applied. The links are set up so that they are forced to stop buckling once they have folded to be completely horizontal.

Scope

At its most basic, the system models quasistatic motion. However, it is capable of modeling dynamics, and can take into account the effects of damage and irreversibility. The effects of friction are neglected.

Section 2. Governing Equations

This section derives the governing equations of the buckling system. There are 3 phases to the buckling process: before buckling, during buckling, and after buckling. Each phase has its own equation for the applied force P in terms of the displacement d.

Basic Equations for Each Part

The force equations for each part – the two springs and the link – are derived here

The Springs

The top spring is modeled as a non-linear spring. The equation used for it is

$$F_t = EA \tanh^{-1}\left(\frac{\delta}{h}\right) \tag{1}$$

Where δ is the amount the spring has compressed from equilibrium, h is the uncompressed spring length, and F_t is the spring force. Using this equation allows the spring force to be linear for small δ, but approach infinity as δ approaches h, ensuring that F_t will always reach the critical buckling force before the spring completely compresses.

The side spring is a linear spring, so it is modeled by Hooke's Law

$$F_s = kx \tag{2}$$

The Links

To find the force required to keep the links in static equilibrium when they're buckling, first a free body diagram of the bent links is required as shown in Figure 2. Applying a force balance gives the equation

$$\sum F_x = A_x + C_x - F_s = 0 \tag{3}$$

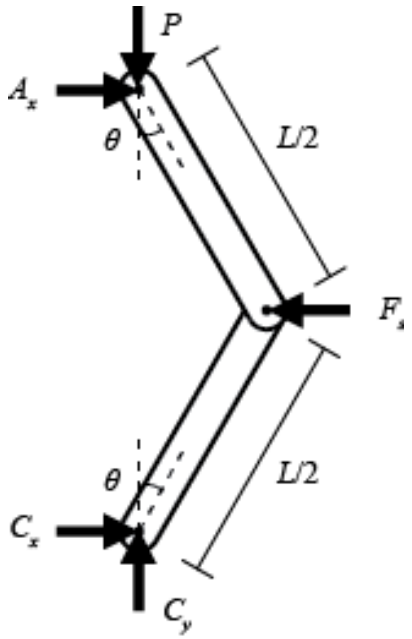


Figure 2: FBD of the buckling links

Because the bent link is symmetric, moment balance implies that $A_x = C_x$. Substituting A_x for C_x in Equation 3 and rearranging gives $F_s = 2A_x$, or, solving for A_x,

$$A_x = \frac{F_s}{2} \tag{4}$$

Drawing a free body diagram of just one link, as seen in Figure 3, putting in Equation 4 for A_x and applying a moment balance about point B gives the equation

$$\sum M_B = P\left(\frac{L}{2}\sin\theta\right) - \frac{F_s}{2}\left(\frac{L}{2}\cos\theta\right) = 0 \tag{5}$$

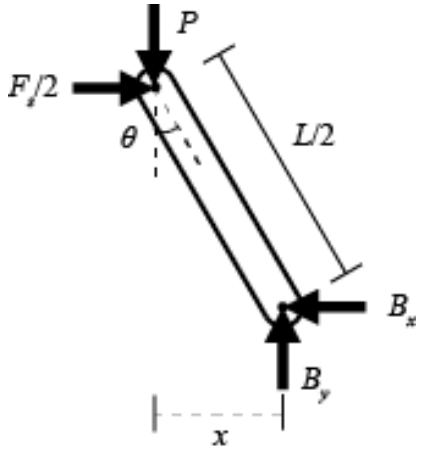


Figure 3: FBD of one link

When the link is bent at the angle θ, the displacement of the side spring is x. From Figure 3, x can be found to equal $\frac{L}{2}\sin\theta$. Substituting that for x in Equation 2 and then substituting for F_s in Equation 5 gives the equation

$$P\left(\frac{L}{2}\sin\theta\right) - \frac{k\frac{L}{2}\sin\theta}{2}\left(\frac{L}{2}\cos\theta\right) = 0 \tag{6}$$

Equation 6 can be simplified and rearranged, producing the equation

$$P = \frac{kL}{4}\cos\theta \tag{7}$$

The Critical Buckling Force

The critical buckling force is the force that must be applied to the link in order for it to buckle. It can be found by setting θ in Equation 7 equal to zero, giving

$$P_{crit} = \frac{kL}{4} \tag{8}$$

Phase 1: Before Buckling

In the system's initial state, the link is unbuckled and the top spring is uncompressed. As shown in Figure 4, the total length of the link is L, the total height of both the link and top spring is H, the uncompressed length of the top spring is h, and the applied force is P (which is initially zero). From Figure 4, it's obvious that

$$h = H - L \tag{9}$$

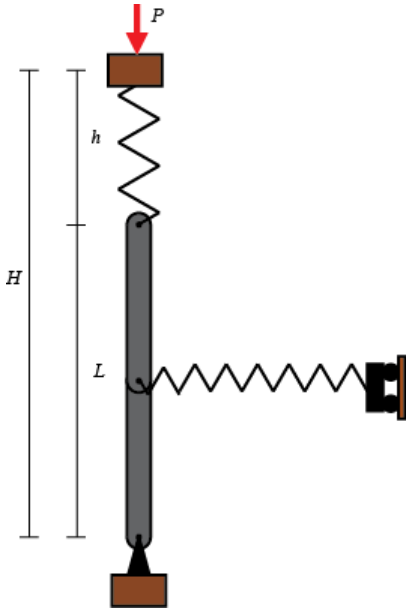


Figure 4: An unbuckled, uncompressed link

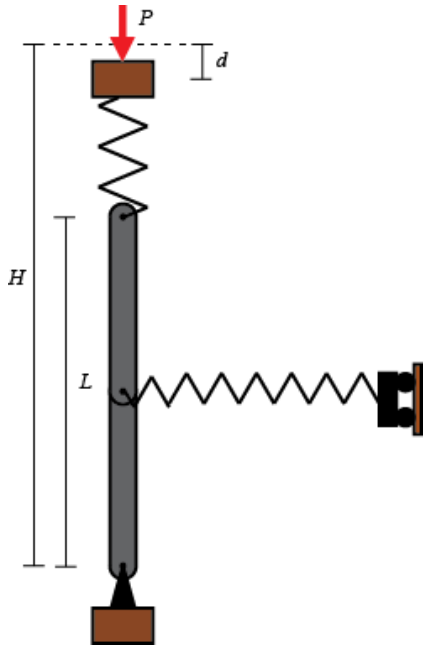


Figure 5: An unbuckled, compressed link

Once the spring begins to compress, P becomes non-zero. P must also equal F_t. Since the links have not yet buckled, $\delta = d$, as can be seen in Figure 5. Substituting P for F_t, d for δ, and Equation 9 for h gives the final equation:

$$P = EA \tanh^{-1}\left(\frac{d}{H-L}\right) \tag{10}$$

Phase 2: During Buckling

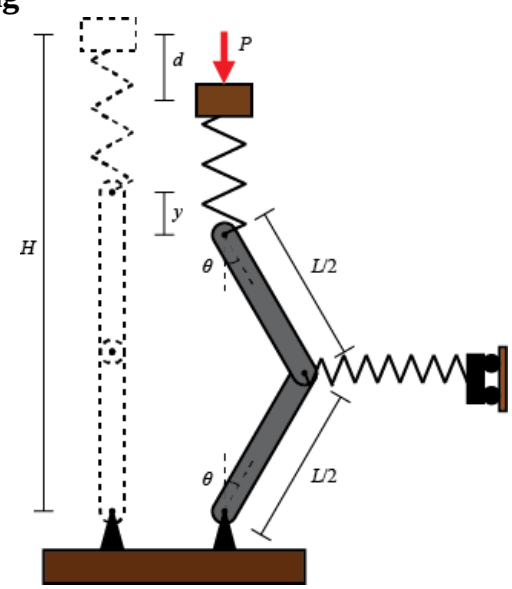


Figure 6: A buckling unit

Once the link begins to buckle, a new variable *y* is introduced, which is the distance the top of the buckling link has displaced from its original position, as shown in Figure 6. For quasistatic motion, *P* still must equal F_t. From Figure 6, an expression for $L\cos\theta$ can be found:

$$2\left(\frac{L}{2}\cos\theta\right) = L - y \tag{11}$$

Simplifying and substituting for $L\cos\theta$ in Equation 7 gives

$$P = \frac{k}{4}(L - y) \tag{12}$$

Because P is also the top spring force,

$$P = EA\tanh^{-1}\left(\frac{\delta}{H - L}\right) \tag{13}$$

Rearranging Equation 12 and Equation 13 to solve for *y* and δ gives the equations

$$\delta = (H - L)\tanh\left(\frac{P}{EA}\right) \tag{14}$$

$$y = L\left(1 - \frac{4P}{kL}\right) \tag{15}$$

From Figure 6, it can be seen that $\delta = d - y$. Solving for *d* and substituting Equation 14 for δ and Equation 15 for *y* gives

$$d = L\left(1 - \frac{4P}{kL}\right) + (H - L)\tanh\left(\frac{P}{EA}\right) \tag{16}$$

Equation 16 must be solved numerically for *P*.

Phase 3: After Buckling

After a link has completely collapsed, *P* is once again equal to the force in the top spring. In this case, $\delta = d - L$, so

$$P = EA\tanh^{-1}\left(\frac{d - L}{H - L}\right) \tag{17}$$

Expanding to multiple units

For a system that contains multiple units, the equation is simple. Just find *P* for each individual unit, and sum them all to get P_{total}.

Section 3. Variability

In order to examine the effects of applying a statistical distribution to one of the variables involved in the governing equations, both a statistical distribution and the parameter to vary are needed. For this model the parameter being varied is the length of the links. Because of this, two statistical distributions are being used: a Weibull distribution and an exponential distribution. To apply the distributions, first a set of random numbers between zero and one is generated. The a Weibull distribution is created by putting each random number into the equation

$$l = \left(\frac{\ln\left(\frac{1}{R}\right)}{\ln 2} \right)^{\frac{1}{Wm}} \tag{18}$$

Where *R* is the random number and *Wm* is the Weibull modulus. The issue with this particular distribution is that it's possible to get lengths that are longer than the maximum possible. To fix that, a second, exponential distribution is used:

$$L = H\left(1 - e^{-l}\right) \tag{19}$$

The *H* is in there in order to scale the values to the desired height of the system. This gives the final link lengths used in the force calculations.

Section 4. Binning

While it is fairly easy to run a simulation that contains a small number of links, running one that has upwards of thousands can be difficult for the computer and take a long time. To remedy this, a binning scheme is applied, as described in "An Efficient Binning Scheme with Application to Statistical Crack Mechanics" (Huq, F., L. Graham-Brady, and R. Brannon. *Int. J. Num. Meth. Engr.*). In essence, the large distribution of randomly generated numbers is replaced with a smaller, representative, weighted sample of those numbers, which when used in the simulation produces a force-displacement curve very similar to the force-displacement curve that would be generated without binning the random number set. Because there is only one parameter being varied, the one-dimensional scheme is used.

Section 5. Damage

Using solely the equations described in Section 1, the loading process is reversible. The force displacement curve will be exactly the same no matter how it is loaded and unloaded. To make it behave more like a real material, where the force vs. displacement curve is not the same for both loading and unloading, some sort of damage must be included. It is implemented by setting that after a link has buckled past a certain angle, it is "broken", and the side spring no longer offers any resistance, making *P* for that particular link zero. This causes the system to unload along a different path than the one it loads on, in a similar fashion to a real material.

Section 6. Dynamics

In order to give the system dynamic effects, a mass *m* is added to the system. This section derives the equations for buckling for the dynamic system both for when *m* is a point mass added between the top link and the top spring, and for when *m* is the mass of the combined links.

During Buckling

Point Mass

A free-body diagram of the links and the mass during buckling is shown in Figure 7.

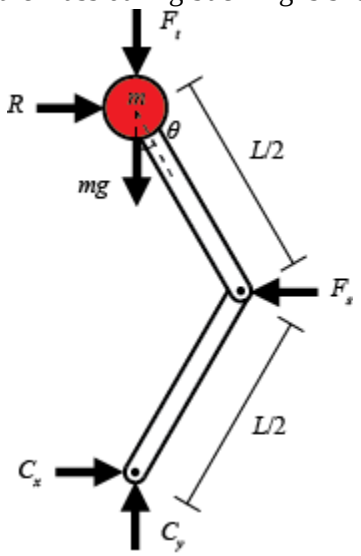


Figure 7: FBD of links and mass

Because the system as a whole is not rotating, the equation for the moment about C is

$$\sum \bar{M}_C = F_s\left(\frac{L}{2}\right)\cos\theta - RL\cos\theta = 0 \tag{20}$$

Rearranging to solve for R gives the equation

$$R = \frac{F_s}{2} \tag{21}$$

Substituting Equation 2 for F_s and $\frac{L}{2}\sin\theta$ for x in Equation 2 gives

$$R = k\frac{L}{4}\sin\theta \tag{22}$$

A free body diagram of the mass is shown in Figure 8, where F_l is the force exerted on the mass by the link. This FBD gives two equations,

$$\sum F_x = m\bar{a}_x = R - F_l\sin\theta \tag{23}$$

$$\sum F_y = m\bar{a}_y = F_l \cos\theta - F_t - mg \tag{24}$$

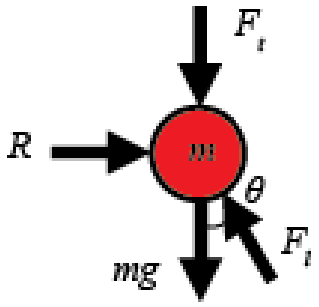


Figure 8: FBD of the point mass

Because the horizontal acceleration of the mass is zero, Equation 23 can become

$$R = F_l \sin\theta \tag{25}$$

Substituting Equation 22 for R and solving for F_l gives

$$F_l = \frac{kL}{4} \tag{26}$$

Substituting Equation 26 for F_l in Equation 24 and Equation 11 for the $L\cos\theta$ produced by that substitution, substituting Equation 13 for F_t with $d - y$ substituted for δ, and then dividing through by m, gives the equation

$$\bar{a}_y = \frac{k}{4m}(L - y) - \frac{EA}{m}\tanh^{-1}\left(\frac{d-y}{H-L}\right) - g \tag{27}$$

Because the distance y increases when $\bar{a}_y$ is negative (downward), $\ddot{y} = -\bar{a}_y$, which means

$$\ddot{y} = -\frac{k}{4m}(L - y) + \frac{EA}{m}\tanh^{-1}\left(\frac{d-y}{H-L}\right) + g \tag{28}$$

This ODE can be solved numerically to find y, which in turn can be used to find P, because

$$P = F_t = EA\tanh^{-1}\left(\frac{d-y}{H-L}\right) \tag{29}$$

Link Mass

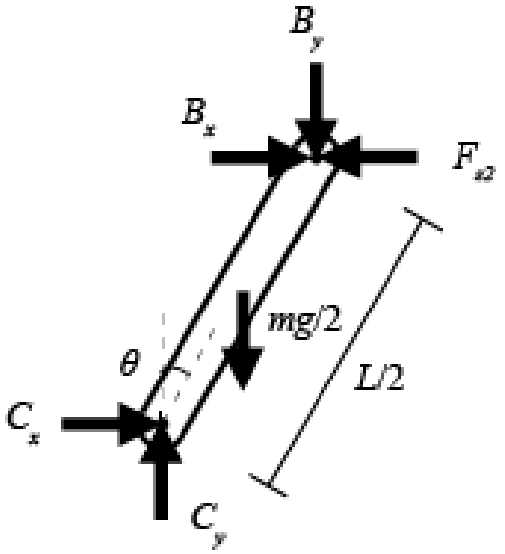


Figure 9: FBD of the bottom link

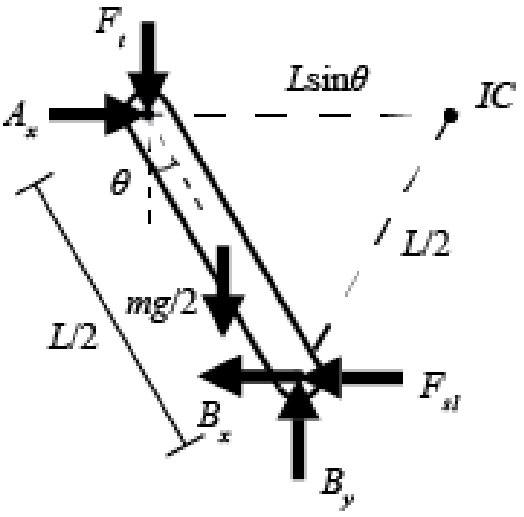


Figure 10: FBD of the top link

Figure 9 and Figure 10 show free body diagrams of the bottom and top links, respectively. F_{s1} and F_{s2} are the components of the side spring force F_s on each link, so $F_{s1} + F_{s2} = F_s$. Because the mass of both links combined is m, each link has mass $m/2$. In Figure 10, the point labeled *IC* is the instantaneous center of zero velocity. For the bottom link, its instantaneous center is at point *C*. If the top link is link 1 and the bottom link is link 2, then taking the moment about the instantaneous center of each link gives the equations

$$\sum M_{IC1} = I_{IC1}\alpha_1 = F_t L\sin\theta - B_y\frac{L}{2}\sin\theta - F_{s1}\frac{L}{2}\cos\theta - B_x\frac{L}{2}\cos\theta + \frac{mg}{2}\frac{3L}{4}\sin\theta \quad (30)$$

$$\sum M_{IC2} = I_{IC2}\alpha_2 = B_x\frac{L}{2}\cos\theta + B_y\frac{L}{2}\sin\theta - F_{s2}\frac{L}{2}\cos\theta + \frac{mg}{2}\frac{L}{4}\sin\theta \quad (31)$$

with the moment arms being found through basic trigonometry, and the positive direction for link two being clockwise instead of the conventional counterclockwise. These two equations can be rearranged into

$$B_y\frac{L}{2}\sin\theta + B_x\frac{L}{2}\cos\theta = F_t L\sin\theta - I_{IC1}\alpha_1 - F_{s1}\frac{L}{2}\cos\theta + \frac{3mgL}{8}\sin\theta \quad (32)$$

$$B_y\frac{L}{2}\sin\theta + B_x\frac{L}{2}\cos\theta = I_{IC2}\alpha_2 + F_{s2}\frac{L}{2}\cos\theta - \frac{mgL}{8}\sin\theta \quad (33)$$

which can be set equal to each other to form the equation

$$I_{IC2}\alpha_2 + F_{s2}\frac{L}{2}\cos\theta - \frac{mgL}{8}\sin\theta = F_t L\sin\theta - I_{IC1}\alpha_1 - F_{s1}\frac{L}{2}\cos\theta + \frac{3mgL}{8}\sin\theta \quad (34)$$

Rearranging, simplifying, and substituting F_s for $F_{s1} + F_{s2}$ gives the equation

$$I_{IC2}\alpha_2 + I_{IC1}\alpha_1 = F_t L \sin\theta - F_s \frac{L}{2}\cos\theta + \frac{mgL}{2}\sin\theta \tag{35}$$

The acceleration of point *A* can be found using the equation

$$\vec{a}_A = \vec{a}_B + \alpha_1 \hat{k} \times \vec{r}_{C/B} - {\omega_1}^2 \vec{r}_{C/B} \tag{36}$$

To find $\vec{a}_B$, the equation for rotational acceleration is used, with point *C* as the center of rotation.

$$\vec{a}_B = \left(\ddot{r} - r\dot{\theta}^2\right)\hat{\varepsilon}_r + (r\ddot{\theta} + 2\dot{r}\dot{\theta})\hat{\varepsilon}_\theta \tag{37}$$

Substituting $L/2$ for r, 0 for $\dot{r}$ and $\ddot{r}$, ω_2 for $\dot{\theta}$, and α_2 for $\ddot{\theta}$ and converting from polar vectors to Cartesian vectors gives the equation

$$\vec{a}_B = \left(-\frac{L}{2}{\omega_2}^2 \sin\theta + \frac{L}{2}\alpha_2 \cos\theta\right)\hat{\imath} + \left(-\frac{L}{2}{\omega_2}^2 \cos\theta - \frac{L}{2}\alpha_2 \sin\theta\right)\hat{\jmath} \tag{38}$$

$\vec{r}_{C/B} = -\frac{L}{2}\sin\theta\,\hat{\imath} + \frac{L}{2}\cos\theta\,\hat{\jmath}$, so $\alpha_1 \times \vec{r}_{C/B} = -\alpha_1 \frac{L}{2}\cos\theta\,\hat{\imath} - \alpha_1 \frac{L}{2}\sin\theta\,\hat{\jmath}$. Substituting both of these equations and Equation 38 into Equation 36 gives

$$\begin{aligned} a_A\hat{\jmath} = {} & -\frac{L}{2}{\omega_2}^2 \sin\theta\,\hat{\imath} + \frac{L}{2}\alpha_2 \cos\theta\,\hat{\imath} - \frac{L}{2}{\omega_2}^2 \cos\theta\,\hat{\jmath} - \frac{L}{2}\alpha_2 \sin\theta\,\hat{\jmath} - \alpha_1 \frac{L}{2}\cos\theta\,\hat{\imath} \\ & - \frac{L}{2}\alpha_1 \sin\theta\,\hat{\jmath} + \frac{L}{2}{\omega_1}^2 \sin\theta\,\hat{\imath} - \frac{L}{2}{\omega_1}^2 \cos\theta\,\hat{\jmath} \end{aligned} \tag{39}$$

Note that a_A only has a vertical component because point *A* is constrained to move vertically. Separating out the $\hat{\imath}$ and $\hat{\jmath}$ components into separate equations gives

$$0 = -\frac{L}{2}{\omega_2}^2 \sin\theta + \frac{L}{2}\alpha_2 \cos\theta - \alpha_1 \frac{L}{2}\cos\theta + \frac{L}{2}{\omega_1}^2 \sin\theta \tag{40}$$

$$a_A = -\frac{L}{2}{\omega_2}^2 \cos\theta - \frac{L}{2}\alpha_2 \sin\theta - \frac{L}{2}\alpha_1 \sin\theta - \frac{L}{2}{\omega_1}^2 \cos\theta \tag{41}$$

Because point *B* is on both links, $v_B = \omega_1 \frac{L}{2}$ and $v_B = \omega_2 \frac{L}{2}$, which means $\omega_1 = \omega_2$. Substituting ω_2 for ω_1 in Equation 40 and simplifying gives the result $\alpha_1 = \alpha_2$. Substituting ω_2 for ω_1 and α_2 for α_1 in Equation 41 and simplifying gives

$$a_A = -L{\omega_2}^2 \cos\theta - L\alpha_2 \sin\theta \tag{42}$$

Substituting α_2 for α_1 in Equation 35 and simplifying gives

$$\alpha_2(I_{IC2} + I_{IC1}) = F_t L \sin\theta - F_s \frac{L}{2}\cos\theta + \frac{mgL}{2}\sin\theta \tag{43}$$

Approximating both link as long thin rods, $I_{IC2} = \frac{1}{3}mL^2$ and $I_{IC1} = \frac{1}{12}mL^2 + md^2$, where here *d* is the distance from the top link's center of mass to its instantaneous center of zero velocity. Through trigonometry, that distance can be found to be $\frac{L}{4}\sqrt{9\sin(\theta)^2 + \cos(\theta)^2}$. Substituting that for *d* and putting both moments of inertia into Equation 43 gives

$$\alpha_2\left(\frac{1}{3}mL^2 + \frac{1}{12}mL^2 + m\left(\frac{L}{4}\sqrt{9\sin(\theta)^2 + \cos(\theta)^2}\right)^2\right) = F_t L \sin\theta - F_s \frac{L}{2}\cos\theta + \frac{mgL}{2}\sin\theta \tag{44}$$

Simplifying Equation 44 and solving for α_2 gives the equation

$$\alpha_2 = \frac{F_t L \sin\theta - F_s \frac{L}{2}\cos\theta + \frac{mgL}{2}\sin\theta}{\frac{5}{12}mL^2 + \frac{mL^2}{16}(9\sin(\theta)^2 + \cos(\theta)^2)} \tag{45}$$

$v_A = \omega_1 L \sin\theta = \omega_2 L \sin\theta$, so $\omega_2 = \frac{v_A}{L\sin\theta}$. Substituting this for ω_2 and Equation 45 for α_2 in Equation 42 results in the equation

$$a_A = -L\left(\frac{v_A}{L\sin\theta}\right)^2 \cos\theta - L\sin\theta\left(\frac{F_t L \sin\theta - F_s \frac{L}{2}\cos\theta + \frac{mgL}{2}\sin\theta}{\frac{5}{12}mL^2 + \frac{mL^2}{16}(9\sin(\theta)^2 + \cos(\theta)^2)}\right) \tag{46}$$

From Equation 11, $\cos\theta$ can be found to equal $\frac{L-y}{L}$. Using the Pythagorean theorem, $\sin\theta$ can then be found to be $\frac{\sqrt{2Ly - y^2}}{L}$. Substituting these two expressions for $\sin\theta$ and $\cos\theta$ in Equation 46 gives

$$a_A = -L\left(\frac{v_A}{L\left(\frac{\sqrt{2Ly - y^2}}{L}\right)}\right)^2 \left(\frac{L-y}{L}\right) - L\left(\frac{\sqrt{2Ly - y^2}}{L}\right)\left(\frac{F_t L\left(\frac{\sqrt{2Ly - y^2}}{L}\right) - F_s \frac{L}{2}\left(\frac{L-y}{L}\right) + \frac{mgL}{2}\left(\frac{\sqrt{2Ly - y^2}}{L}\right)}{\frac{5}{12}mL^2 + \frac{mL^2}{16}\left(9\left(\frac{\sqrt{2Ly - y^2}}{L}\right)^2 + \left(\frac{L-y}{L}\right)^2\right)}\right) \tag{47}$$

After much simplification, and substituting $k\frac{L}{2}\sin\theta = k\frac{L}{2}\left(\frac{\sqrt{2Ly-y^2}}{L}\right)$ for F_s and Equation 29 for F_t, the final equation becomes

$$a_A = -\frac{{v_A}^2(L-y)}{2Ly-y^2} - (2Ly-y^2)\left(\frac{\frac{mg}{2} + EA\tanh^{-1}\left(\frac{d-y}{H-L}\right) - \frac{k(L-y)}{4}}{\frac{23mL^2}{48} + mLy - \frac{my^2}{2}}\right) \tag{48}$$

Because $a_A = -\ddot{y}$ and $v_A = -\dot{y}$, Equation 48 can be used to numerically solve for y, which in turn can be used, as shown in Equation 29, to solve for P.

Before Buckling and After Collapse

The equations for before buckling and after collapse are the same as for the quasistatic model, because the mass isn't moving during those phases. However, the equation for the critical applied force to cause buckling has to be adjusted to accommodate the force of gravity on the mass, which changes it from Equation 8 to

$$P_{crit} = \frac{kL}{4} - mg \tag{49}$$

Section 7. Plasticity

To add effects of plasticity to the model, a Jenkins element is added to the side spring. It is treated as a block on a surface with friction. Once the force in the side spring reaches a critical force, the block begins to slide and the force in the spring remains constant. As the links buckle and unbuckle, this provides effects similar to plastic deformation.

Using this model, the force in the side spring is not always $k\frac{L}{2}\sin\theta$. If $x = \frac{L}{2}\sin\theta$ and x_{plas} is the distance from its starting point that the block has slid, then.

$$F_s = k\left(x - x_{plas}\right) \tag{50}$$

Application to the Quasistatic Model

To add plasticity to the quasistatic model, Equation 50 is substituted for F_s in Equation 5, which is subsequently solved for P and set equal to Equation 29. This results in the equation

$$EA\tanh^{-1}\left(\frac{d-y}{H-L}\right) = \frac{k(x-x_{plas})}{2}\frac{\cos\theta}{\sin\theta} \tag{51}$$

Substituting $\frac{L}{2}\sin\theta$ for x, $\frac{L-y}{L}$ for $\cos\theta$, and $\frac{\sqrt{2Ly-y^2}}{L}$ for $\sin\theta$ gives

$$EA\tanh^{-1}\left(\frac{d-y}{H-L}\right)=\frac{k(\frac{L}{2}\frac{\sqrt{2Ly-y^2}}{L}-x_{plas})}{2}\frac{\frac{L-y}{L}}{\frac{\sqrt{2Ly-y^2}}{L}} \tag{52}$$

Which simplifies down to

$$EA\tanh^{-1}\left(\frac{d-y}{H-L}\right)=k(L-y)\left(\frac{1}{4}-\frac{x_{plas}}{2\sqrt{2Ly-y^2}}\right) \tag{53}$$

Equation 53 can be solved numerically for *y*, which can be substituted into Equation 29 to find *P*.

Application to the Dynamic Model

To add plasticity to the dynamic model, Equation 50 is substituted for F_s in Equation 47, with $\frac{L}{2}\sin\theta$ in place of *x* and $\frac{\sqrt{2Ly-y^2}}{L}$ in place of $\sin\theta$. After simplification, this results in the equation

$$a_A=-\frac{{v_A}^2(L-y)}{2Ly-y^2}$$
$$-(2Ly-y^2)\left(\frac{(2Ly-y^2)\left(\frac{mg}{2}+EA\tanh^{-1}\left(\frac{d-y}{H-L}\right)-\frac{k(L-y)}{4}\right)+kx_{plas}\sqrt{2Ly-y^2}\frac{(L-y)}{2}}{\frac{23mL^2}{48}+mLy-\frac{my^2}{2}}\right) \tag{54}$$

which can be numerically solved for *y*, which can be substituted into Equation 29 to find *P*, as explained in Section 6: Dynamics - During Buckling - Link Mass.

Section 8. Viscosity

To add viscosity effects, a dashpot is added either in series or in parallel with the side spring. The equation for the dashpot is written as

$$F_d=cv \tag{55}$$

where *v* is the speed of the moving part of the dashpot and *c* is the damping coefficient.

Dashpot in Parallel

If the dashpot is in parallel with the side spring, the net force exerted on the side of the links is $F_d + F_s$. This results in the equation $F_{side}=cv+kx$. Since the spring and dashpot are in parallel, the distance which the moving part of the dashpot has traveled is equal to *x*, making $v=\dot{x}$, and turning the equation for the force on the side of the links into

$$F_{side}=c\dot{x}+kx \tag{56}$$

Equation 56 can be substituted for F_s in Equation 47 in a similar manner as with plasticity, with the same substitutions for x and $\sin\theta$. Doing so and simplifying results in the equation

$$a_A = -\frac{{v_A}^2(L-y)}{2Ly-y^2} - (2Ly-y^2)\left(\frac{(2Ly-y^2)\left(\frac{mg}{2}+EA\tanh^{-1}\left(\frac{d-y}{H-L}\right)-\frac{k(L-y)}{4}\right)-c\dot{x}\sqrt{2Ly-y^2}\frac{(L-y)}{2}}{\frac{23mL^2}{48}+mLy-\frac{my^2}{2}}\right) \quad (57)$$

Note that this is almost exactly like Equation 54, if $\frac{c\dot{x}}{k}$ were substituted for x_{plas}. If $\dot{x}$ is approximated as $\frac{\Delta x}{\Delta t}$, this equation can be solved numerically for y, which can be substituted into Equation 29 to find P, as explained in Section 6: Dynamics - During Buckling - Link Mass.

Dashpot in Series

If the dashpot is in series with the side spring, then the force exerted on the links by the side spring must have the same magnitude as the force exerted on the side spring by the dashpot. This results in the equation $F_d = F_s$. If the distance the moving part of the dashpot has displaced is called l, then $F_d = c\dot{l}$ and $F_s = k(x-l)$. If these expressions are set equal to each other and solved for $\dot{l}$, it results in the equation

$$\frac{k(x-l)}{c} = \dot{l} \quad (58)$$

This differential equation can be easily solved for l.
If the expression for F_s is substituted for F_s in Equation 47 in a similar manner as with plasticity, with the same substitutions for x and $\sin\theta$, the resulting equation becomes

$$a_A = -\frac{{v_A}^2(L-y)}{2Ly-y^2} - (2Ly-y^2)\left(\frac{(2Ly-y^2)\left(\frac{mg}{2}+EA\tanh^{-1}\left(\frac{d-y}{H-L}\right)-\frac{k(L-y)}{4}\right)+kl\sqrt{2Ly-y^2}\frac{(L-y)}{2}}{\frac{23mL^2}{48}+mLy-\frac{my^2}{2}}\right) \quad (59)$$

Note that this equation is almost exactly like Equation 54 if l is substituted for x_plas. If l is solved for using Equation 58, then Equation 59 can be solved numerically for y, which can be substituted into Equation 29 to find P.

Section 9. Implementation in the Code

This section describes the basics of the code for the model. For more information on the details of the code, see Appendix C. The code is written in Python. It uses the Matplotlib and Numpy packages, so those must be installed. The code requires the user to set values for EA, k, H, the Weibull modulus, and the number of links. If the dynamics model is being run, it also requires a mass, gravity, time step, and stop time. If the model is being run with damage, a critical breaking angle must be set. If the models is

being run with plasticity, the critical side spring force for plasticity must be set. If the model is being run with a dashpot, the damping coefficient must be set as well. The user must also set whether or not they want to apply dynamics, damage, plasticity, viscosity (from the dashpots), and/or binning, as well as whether they want to make and save an animation, by setting the associated parameters as "True" or "False." Examples for setting these parameters can be found in Appendix B. Details about all the parameters can be found in Appendix C - Main Code - Initialization

In essence, the code does four things. It creates a statistically distributed list of link lengths, it iterates through a list of displacements and calculates the applied force at each, it plots the force vs. displacement curve, and (optionally) it creates an animation of the buckling links. It will perform the second and third tasks for the average link length as well, for comparison.

Creating the Distribution of Link Lengths

To get the distribution of link lengths, the code first uses Python's *random* function to generate a list of random numbers between zero and one, one number for each link. It puts each number through Equation 18 and then Equation 19, which gives the final length for each link. It also calculates the average length.

Finding the Applied Force

In order to find the applied force, the code solves the equations derived in Section 1 (for the basic quasistatic model), Section 6 (for the basic dynamic model), Section 7 (for plasticity), or Section 8 (for viscosity), for each link, and then sums all the forces to find the total, *P*. For both before buckling and after collapse, the equations are the same for all models (Equation 10 and Equation 17) and straightforward to solve, but while the link is buckling the equations must be solved numerically. Because different numerical methods are required for each model, they will be discussed separately.

The Quasistatic Model

To find the applied force during buckling for the quasistatic model, the code solves one of the quasistatic equations numerically using a root-finding method. Without plasticity, the code solves Equation 16 numerically for *P* by moving *d* to the other side and using the Newton-Raphson method. Taking the derivative of the equation isn't particularly difficult, and ends up being

$$-\frac{4}{k}+\frac{H-L}{EA}\left(1-\tanh\left(\frac{P}{EA}\right)^2\right) \tag{55}$$

Due to some issues with trying to use the pre-written Scipy Newton-Raphson function, a custom function is used instead. More details can be found in Appendix C.

Once *P* is found, it is then used to find *y* with Equation 15, which is saved for later use.

With plasticity, the code solves Equation 53 numerically for *y* by moving all terms to one side and using the bisection method, with a lower boundary of either zero or the smallest *y* value that doesn't make $tanh^{-1}$ undefined, whichever is larger, and an upper boundary with the largest *y* value that doesn't make $tanh^{-1}$ undefined. The custom function mentioned above includes the ability to use the bisection method, and is used for the numeric solver. If necessary, *x_plas* is updated.

The Dynamic Model

To find the applied force during buckling for the dynamic model, the code first numerically solves one of several possible ODEs to find *y*, which it then uses in Equation 29 to find *P*. For basic dynamics, it solves Equation 48, for plasticity it solves Equation 54, for a dashpot in series it solves Equation 59, and for a dashpot in parallel it solves Equation 57. To solve the ODE, the code uses the Runge Kutta method twice,

once to find $\dot{y}$ and once to find y, with the initial y and $\dot{y}$ being the values from the previous iteration. If doing a dashpot in series, it uses the Runge Kutta method an additional time to solve Equation 58 for l. While the code is set to only use the model where the links have mass, the model with the point mass still exists and can be easily switched to, as described in Appendix C - The Runge Kutta ODE Solver: RK4Dyn.py. However, the point mass can only be used for the basic dynamics equation; it does not support plasticity or viscosity.

Changing Phases

For both models, in order to tell when the link changes phase (e.g. from unbuckled to buckling), the code checks whether the conditions for a phase change have been met at the end of each force calculation iteration. The conditions it looks for are as follows: for changing from unbuckled to buckled, it checks to see whether the calculated force is larger than the critical buckling force; for changing from buckling to completely collapsed, it checks to see whether $y \geq L$; for changing from completely collapsed to buckling, it checks to see if the applied force is less than *-g*m* (i.e. whether the applied force can overcome gravity); for changing from buckling to unbuckled, it checks for whether $y < 0$. In the special case of the quasistatic model with plasticity, it looks for the additional criterion that $y > 10y_{previous}$. This is due to there being two roots in the function the root finding code solves. The root finding code is set up to find the correct root, but it uses a search interval that does not go below zero, so if the correct root goes below zero it will find the other root instead, and y will jump to a large value instead of going to a negative value.

Plotting the Force vs. Displacement Curve

The code uses Matplotlib's pyplot to plot the force vs. displacement curve. It sets its x-axis scale based on *H*, and its y-axis scale based on the maximum (and sometimes minimum) force for the average link length. It plots the force vs. displacement curve for the average length as well.

Animating the System

The code uses a separate animation function to animate the system. That function uses Matplotlib's path, patches, and animation libraries. It calculates the endpoints for each link, and uses that to find the points to outline the link. It then applies a PathPatch to draw the link, and plots dots on the link joints. The function calculates the bend points for each spring and plots lines between them to draw the spring. It then draws a rectangle across the top of all the links. If the dynamic model is being used, it plots a large red dot on each link. In a separate subplot, it plots the force vs. displacement curve up to that point, with a point at the end of the curve. The function then uses animation.FuncAnimation to create the actual animation. For more details on the calculations and creating the animation, see Appendix C. If the model is dynamic, the default frame rate plays the animation with a time step of 0.1s (i.e. 10fps). This can be changed by changing the t_step input. The function will then change the frame rate to play the animation with the new time step, as long as it doesn't make the frame rate exceed the maximum possible. If t_step is set to the time step value used for the calculations, the animation will play in real time. If the model is quasistatic, the frame rate defaults to 50fps, which can be changed with the fps input. The frameskip input can be used to have the animation skip frames. For instance, if frameskip=3, the animation would only play every 3rd frame.

Appendix A. Nomenclature

This appendix lists the variables referenced in this document.

Symbol	Name and Definition	SI Units	Equation or Page #
EA	Spring constant type value for the top spring	N	p.4
F_t	Force applied by the top spring	N	p.4, Equation 1
δ	Compression of the top spring from equilibrium	m	p.4
h	Unstretched length of the top spring	m	p.4
F_s	Force applied by the side spring	N	p.4, Equation 2
k	Spring constant for the side spring	N/m	N/A
x	Compression of the side spring from equilibrium	m	N/A
P	Applied force	N	p.4
L	Total link length	m	p.6
H	Total height of system	m	p.6
θ	Angle from vertical of the link	Radians	p.5, Figure 2
y	Displacement of the top of the link	m	p.8
Wm	Weibull Modulus	1	p.9
x_{plas}	Displacement of Jenkins element	m	p.15
c	Damping coefficient for the dashpots	N*s/m	p.16
l	Displacement of the dashpot	m	p.17

Appendix B. Input Examples

This appendix provides examples of the parameters to input to create various types of simulations, and the results that should be generated. For examples that include animations, GIFs of what the animations should look like can be found in the folder "exampleGIFs". (Note that, depending on the computer being used, the GIFs' timing may not exactly match the animations, but they should be relatively close.) To run the example, go to the initialization section in the code, which starts at line 59, and set all parameters to the corresponding values in the table. Any value that has (N/A) beside it is not applicable to the simulation and can be set to any value. The values in the table are in the same order and the same groups as in the code, for ease of use. Ignore the *start_time* and any *if* statements in the code. It is recommended the user read Appendix C – The Main Code: MasterAnalysis.py – Initialization for more detailed descriptions of the input parameters. For more information on how the displacement creating function works, see Appendix C – The Displacement Path Functions: dfunc.py

Example 1: A Basic 2-Link Quasistatic Simulation

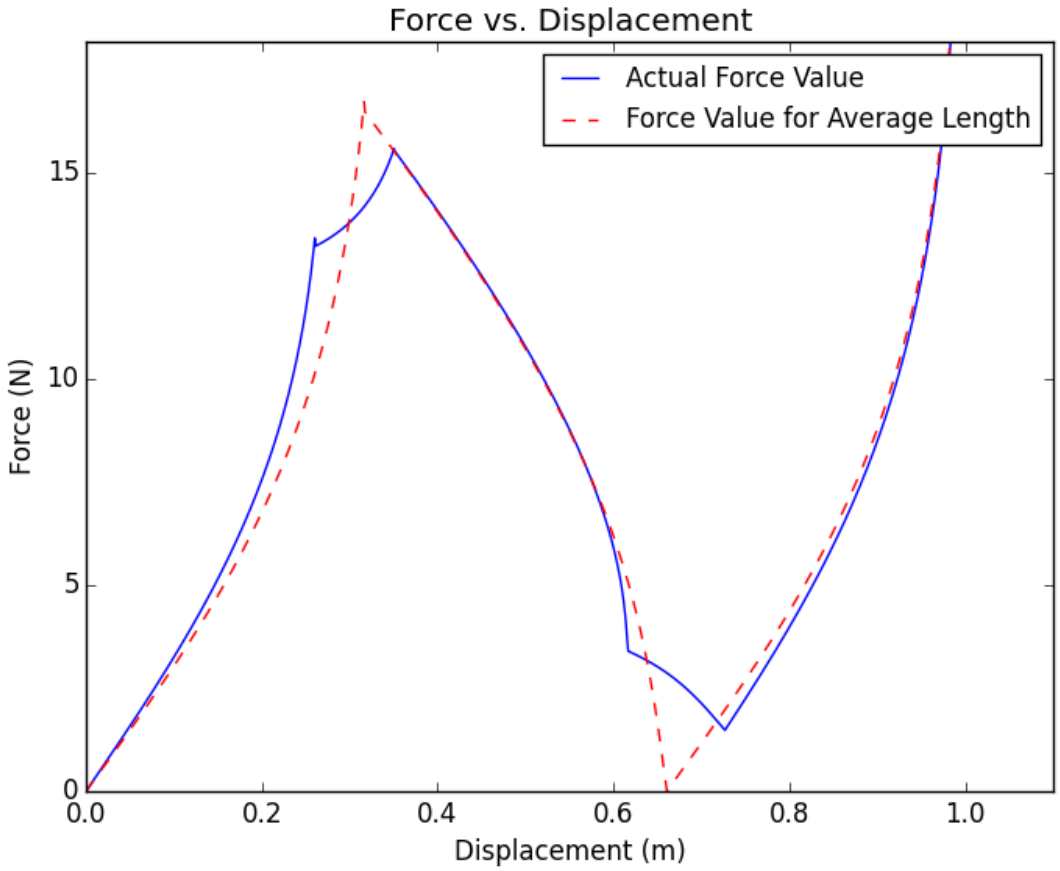
This example shows how to run a 2-link quasistatic simulation where the links are steadily completely compressed, and generate an animation.

Inputs

Inputs	***Value***	***Units***	***Description***
Type of Simulation Inputs			
dyn	False	--	Run simulation with dynamics
damage	False	--	Run simulation with damage

plasticity	False	--	Run simulation with plasticity
dashpot_par	False	--	Run simulation with parallel dashpot
dashpot_ser	False	--	Run simulation with series dashpot
bing	False	--	Run simulation with binning
testing	True	--	Run simulation in testing mode
auto_dfix	True	--	Auto-fix illegal displacements
multirun	False	--	Run simulation multiple times
Animation Options			
make_ani	True	--	Create animation
save_ani	False	--	Save animation as GIF
auto_frameskip	False	--	Auto-set *frameskip* for 10fps
Basic Parameters			
ea_t	10.0	N	Top spring constant-like value
k_t	100.0	N/m	Side spring constant
H	1.0	m	Total height
Wm	4.0	1	Weibull modulus
n_l	2	1	Number of links
steps	1000	1	Number of displacement values
Binning Parameters			
i_max	10 (N/A)	1	Number of bins
Damage/Plasticity Parameters			
theta_crit	10.0 (N/A)	degrees	Breaking angle
F_slide	5.0 (N/A)	N	Critical sliding force for plasticity
Dashpot Parameters			
c_dash	10.0 (N/A)	N*s/m	Damping coefficient
Dynamics Parameters			
m	1.0 (N/A)	kg	Mass
g	0.0 (N/A)	m/s^2	Acceleration of gravity
dt	0.01 (N/A)	s	Time step
t_stop	10.0 (N/A)	s	Total time to run simulation
Animation Parameters			
xspacing	0.5*H	m	Horizontal spacing between links
fps	50	fps	Animation speed
frameskip	1	1	Frames to skip
fname	“fn.gif” (N/A)	--	File name for saving GIF
Multirun Parameters			
runs	1 (N/A)	1	# of times to run simulation
Displacements			
d	dfunc.linepath(steps,H)	m	Displacement list

Outputs



Animation GIF "Example1"

Example 2: A 50000-Link Quasistatic Simulation with Binning

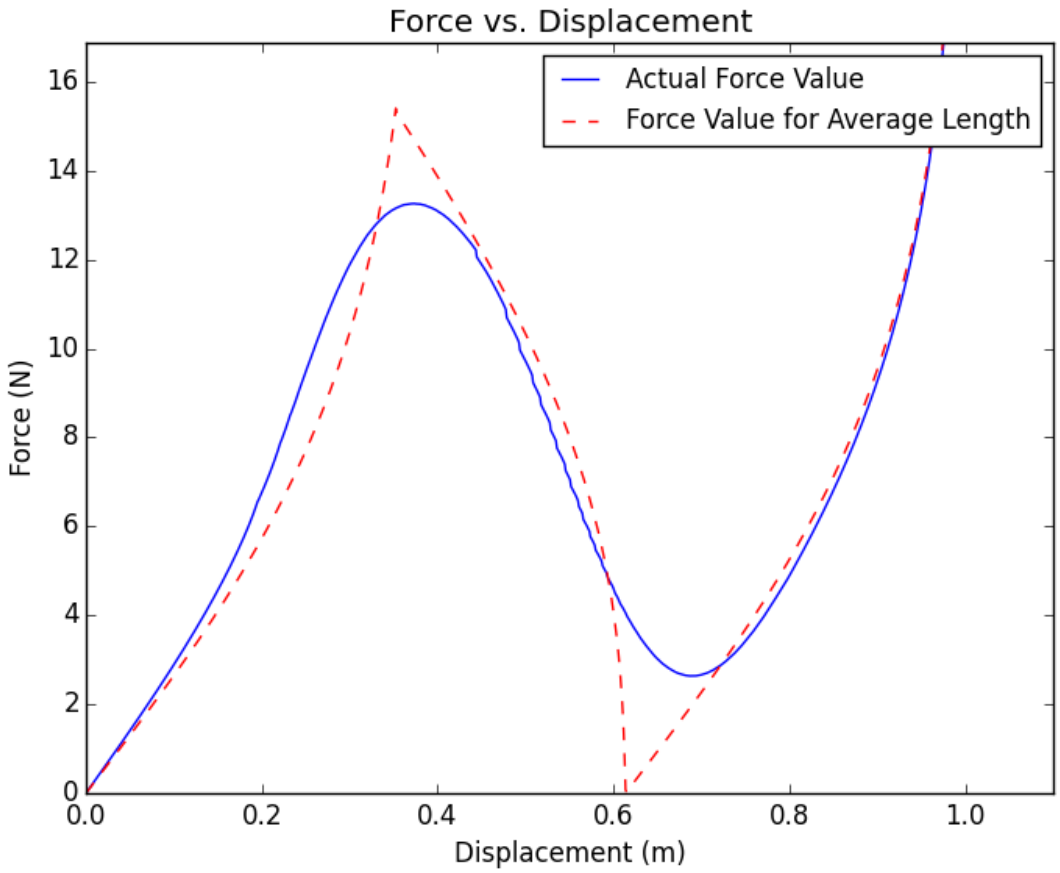
This example shows how to run a 50000-link quasistatic simulation with binning, using 21 bins, where the links are steadily completely compressed.

Inputs

Inputs	***Value***	***Units***	***Description***
Type of Simulation Inputs			
dyn	False	--	Run simulation with dynamics
damage	False	--	Run simulation with damage
plasticity	False	--	Run simulation with plasticity
dashpot_par	False	--	Run simulation with parallel dashpot
dashpot_ser	False	--	Run simulation with series dashpot
bing	True	--	Run simulation with binning
testing	False	--	Run simulation in testing mode
auto_dfix	True	--	Auto-fix illegal displacements
multirun	False	--	Run simulation multiple times
Animation Options			

make_ani	False	--	Create animation
save_ani	False	--	Save animation as GIF
auto_frameskip	False	--	Auto-set *frameskip* for 10fps
Basic Parameters			
ea_t	10.0	N	Top spring constant-like value
k_t	100.0	N/m	Side spring constant
H	1.0	m	Total height
Wm	4.0	1	Weibull modulus
n_l	50000	1	Number of links
steps	1000	1	Number of displacement values
Binning Parameters			
i_max	21	1	Number of bins
Damage/Plasticity Parameters			
theta_crit	10.0 (N/A)	degrees	Breaking angle
F_slide	5.0 (N/A)	N	Critical sliding force for plasticity
Dashpot Parameters			
c_dash	10.0 (N/A)	N*s/m	Damping coefficient
Dynamics Parameters			
m	1.0 (N/A)	kg	Mass
g	0.0 (N/A)	m/s^2	Acceleration of gravity
dt	0.01 (N/A)	s	Time step
t_stop	10.0 (N/A)	s	Total time to run simulation
Animation Parameters			
xspacing	0.5*H (N/A)	m	Horizontal spacing between links
fps	50 (N/A)	fps	Animation speed
frameskip	1 (N/A)	1	Frames to skip
fname	“fn.gif” (N/A)	--	File name for saving GIF
Multirun Parameters			
runs	1 (N/A)	1	# of times to run simulation
Displacements			
d	dfunc.linepath(steps,H)	m	Displacement list

Outputs



Example 3: A 3-Link Quasistatic Simulation with Damage

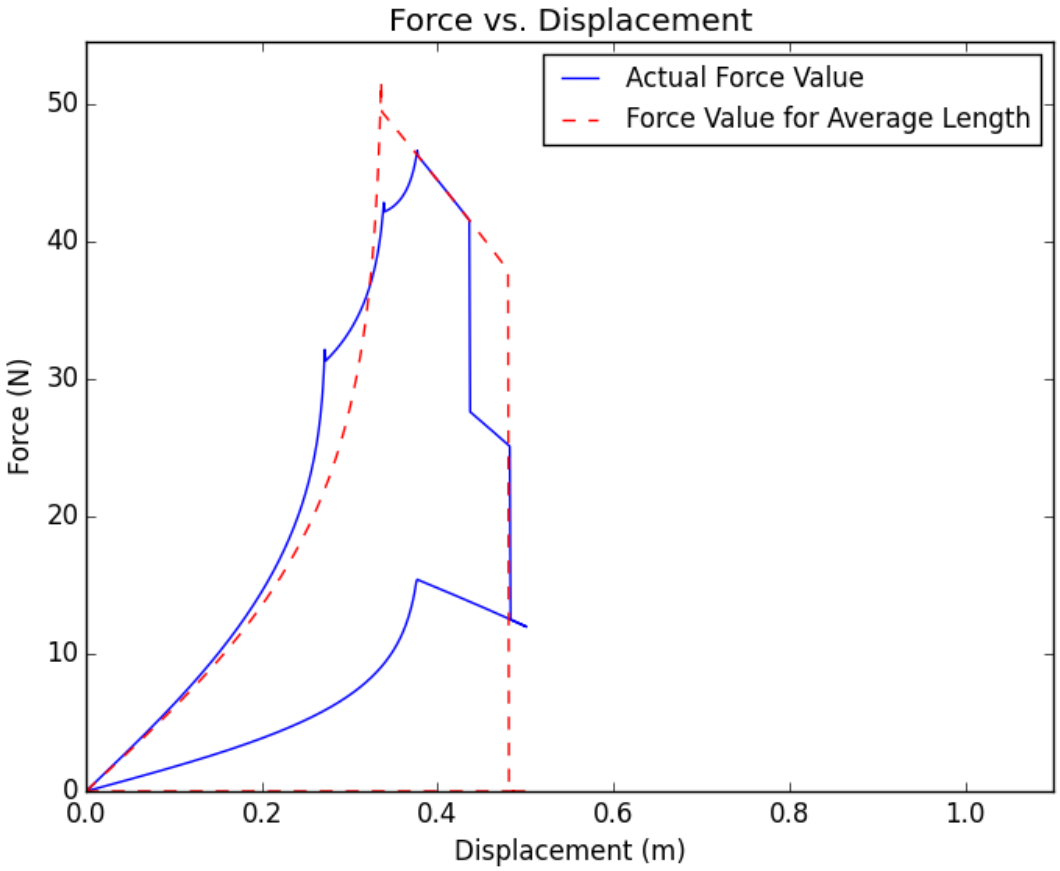
This example shows how to run a 3-link quasistatic simulation that includes the effects of damage, where the displacement steadily increases and then steadily decreases, and animate the simulation.

Inputs

Inputs	***Value***	***Units***	***Description***
Type of Simulation Inputs			
dyn	False	--	Run simulation with dynamics
damage	True	--	Run simulation with damage
plasticity	False	--	Run simulation with plasticity
dashpot_par	False	--	Run simulation with parallel dashpot
dashpot_ser	False	--	Run simulation with series dashpot
bing	False	--	Run simulation with binning
testing	True	--	Run simulation in testing mode
auto_dfix	True	--	Auto-fix illegal displacements
multirun	False	--	Run simulation multiple times
Animation Options			

make_ani	True	--	Create animation
save_ani	False	--	Save animation as GIF
auto_frameskip	False	--	Auto-set *frameskip* for 10fps
Basic Parameters			
ea_t	20.0	N	Top spring constant-like value
k_t	300.0	N/m	Side spring constant
H	1.0	m	Total height
Wm	4.0	1	Weibull modulus
n_l	3	1	Number of links
steps	1000	1	Number of displacement values
Binning Parameters			
i_max	10 (N/A)	1	Number of bins
Damage/Plasticity Parameters			
theta_crit	40.0	degrees	Breaking angle
F_slide	10.0 (N/A)	N	Critical sliding force for plasticity
Dashpot Parameters			
c_dash	10.0 (N/A)	N*s/m	Damping coefficient
Dynamics Parameters			
m	1.0 (N/A)	kg	Mass
g	0.0 (N/A)	m/s^2	Acceleration of gravity
dt	0.01 (N/A)	s	Time step
t_stop	10.0 (N/A)	s	Total time to run simulation
Animation Parameters			
xspacing	0.5*H	m	Horizontal spacing between links
fps	50	fps	Animation speed
frameskip	1	1	Frames to skip
fname	“fn.gif” (N/A)	--	File name for saving GIF
Multirun Parameters			
runs	1 (N/A)	1	# of times to run simulation
Displacements			
d	dfunc.vpath(steps,1,H,0.5)	m	Displacement list

Outputs



Animation GIF "Example3"

Example 4: A 3-Link Quasistatic Simulation with Plasticity

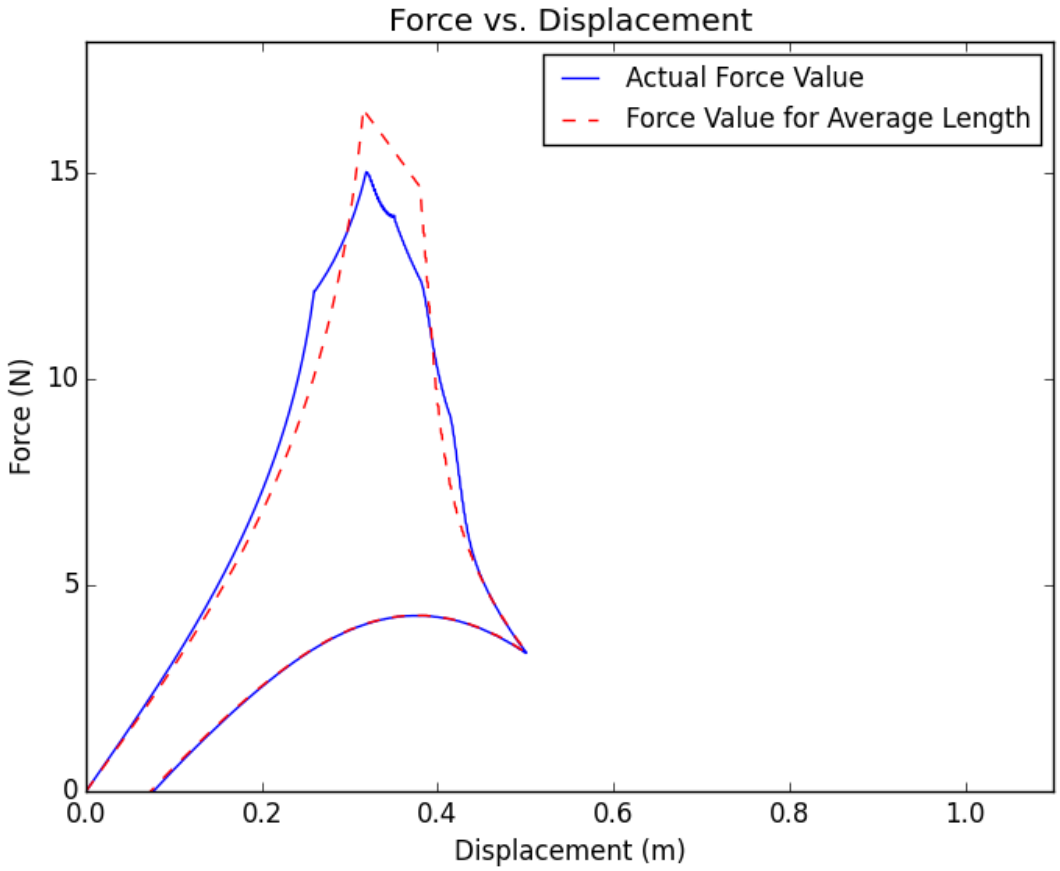
This example shows how to run a 3-link quasistatic simulation with plasticity turned on, where the displacement steadily increases and then steadily decreases, and create an animation.

Inputs

Inputs	***Value***	***Units***	***Description***
Type of Simulation Inputs			
dyn	False	--	Run simulation with dynamics
damage	False	--	Run simulation with damage
plasticity	True	--	Run simulation with plasticity
dashpot_par	False	--	Run simulation with parallel dashpot
dashpot_ser	False	--	Run simulation with series dashpot
bing	False	--	Run simulation with binning
testing	True	--	Run simulation in testing mode
auto_dfix	True	--	Auto-fix illegal displacements
multirun	False	--	Run simulation multiple times

Animation Options			
make_ani	True	--	Create animation
save_ani	False	--	Save animation as GIF
auto_frameskip	False	--	Auto-set *frameskip* for 10fps
Basic Parameters			
ea_t	10.0	N	Top spring constant-like value
k_t	100.0	N/m	Side spring constant
H	1.0	m	Total height
Wm	4.0	1	Weibull modulus
n_l	3	1	Number of links
steps	1000	1	Number of displacement values
Binning Parameters			
i_max	10 (N/A)	1	Number of bins
Damage/Plasticity Parameters			
theta_crit	10.0 (N/A)	degrees	Breaking angle
F_slide	5.0	N	Critical sliding force for plasticity
Dashpot Parameters			
c_dash	10.0 (N/A)	N*s/m	Damping coefficient
Dynamics Parameters			
m	1.0 (N/A)	kg	Mass
g	0.0 (N/A)	m/s^2	Acceleration of gravity
dt	0.01 (N/A)	s	Time step
t_stop	10.0 (N/A)	s	Total time to run simulation
Animation Parameters			
xspacing	0.5*H	m	Horizontal spacing between links
fps	50	fps	Animation speed
frameskip	1	1	Frames to skip
fname	“fn.gif” (N/A)	--	File name for saving GIF
Multirun Parameters			
runs	1 (N/A)	1	# of times to run simulation
Displacements			
d	dfunc.vpath(steps,1,H,0.5)	m	Displacement list

Outputs



Animation GIF "Example4"

Example 5: A 2-Link Dynamic Simulation

This example shows how to run a 2-link dynamic simulation with a composite displacement path where the links are steadily compressed to half the total displacement, held still, then completely compressed. An animation is also created.

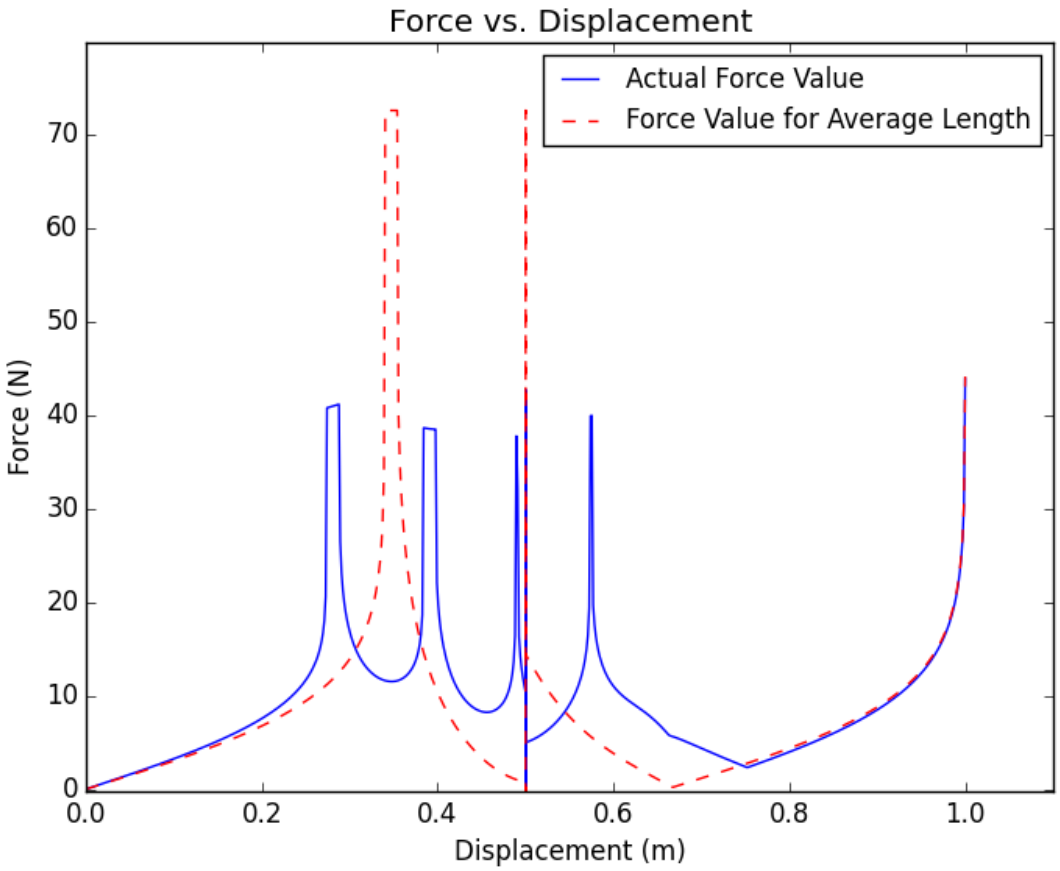
Inputs

Inputs	***Value***	***Units***	***Description***
Type of Simulation Inputs			
dyn	True	--	Run simulation with dynamics
damage	False	--	Run simulation with damage
plasticity	False	--	Run simulation with plasticity
dashpot_par	False	--	Run simulation with parallel dashpot

dashpot_ser	False	--	Run simulation with series dashpot
bing	False	--	Run simulation with binning
testing	True	--	Run simulation in testing mode
auto_dfix	True	--	Auto-fix illegal displacements
multirun	False	--	Run simulation multiple times
Animation Options			
make_ani	True	--	Create animation
save_ani	False	--	Save animation as GIF
auto_frameskip	False	--	Auto-set *frameskip* for 10fps
Basic Parameters			
ea_t	10.0	N	Top spring constant-like value
k_t	100.0	N/m	Side spring constant
H	1.0	m	Total height
Wm	4.0	1	Weibull modulus
n_l	2	1	Number of links
steps	1000 (N/A)	1	Number of displacement values
Binning Parameters			
i_max	10 (N/A)	1	Number of bins
Damage/Plasticity Parameters			
theta_crit	10.0 (N/A)	degrees	Breaking angle
F_slide	10.0 (N/A)	N	Critical sliding force for plasticity
Dashpot Parameters			
c_dash	10.0 (N/A)	N*s/m	Damping coefficient
Dynamics Parameters			
m	2.0	kg	Mass
g	0.0	m/s^2	Acceleration of gravity
dt	0.01	s	Time step
t_stop	10.0	s	Total time to run simulation
Animation Parameters			
xspacing	0.5*H	m	Horizontal spacing between links
fps	50	fps	Animation speed
frameskip	2	1	Frames to skip
fname	“fn.gif” (N/A)	--	File name for saving GIF
Multirun Parameters			

runs	1 (N/A)	1	# of times to run simulation
Displacements			
d	d1=dfunc.linepath(steps/3,0.5*H) d2=dfunc.pausepath(steps/3,0.5*H) d3=dfunc.linepath(steps/3,H,d0=0.5*H) d=d1+d2+d3	m	Displacement list

Outputs



Animation GIF "Example5"

Example 6: A 4-Link Dynamic Simulation with Damage

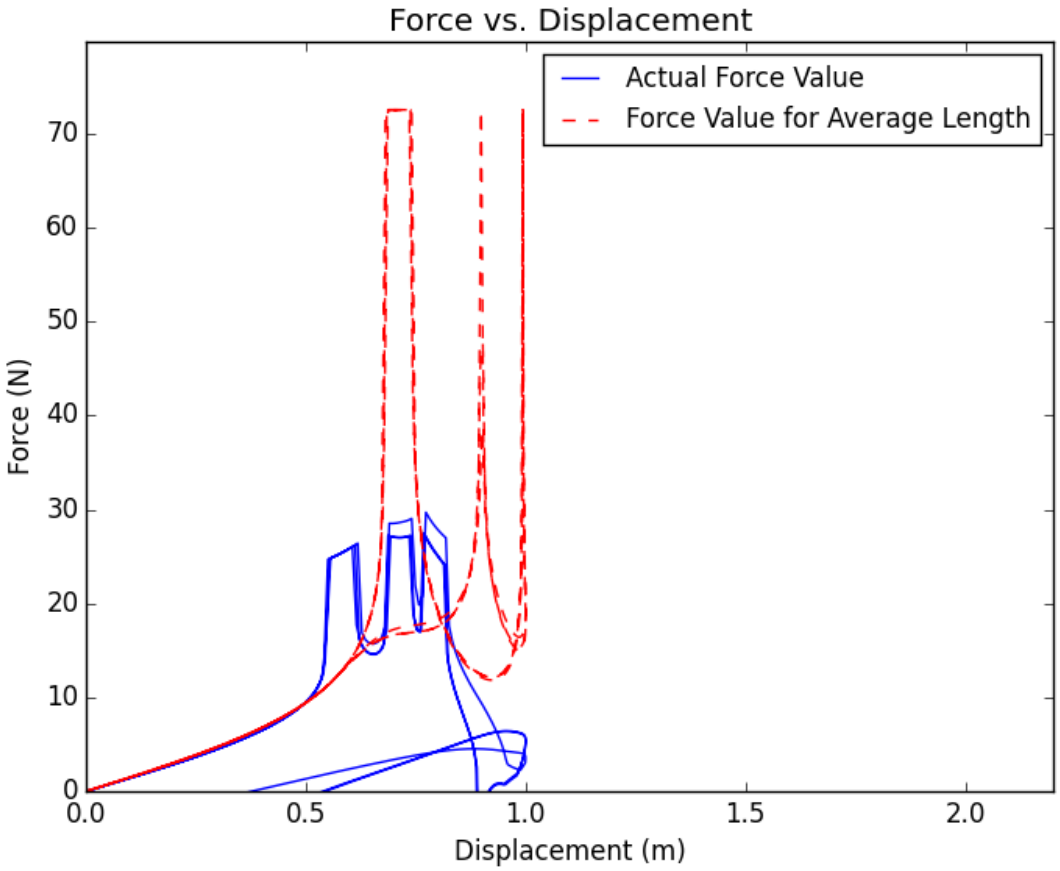
This example shows how to run a 4-link dynamic simulation with damage where the links are displaced along a cosine path, and generate and save an animation. Note that in order to save the entire animation, it must be allowed to play completely through before being closed. Otherwise only the part played before being closed will be saved.

Inputs

Inputs	***Value***	***Units***	***Description***

Type of Simulation Inputs			
dyn	True	--	Run simulation with dynamics
damage	True	--	Run simulation with damage
plasticity	False	--	Run simulation with plasticity
dashpot_par	False	--	Run simulation with parallel dashpot
dashpot_ser	False	--	Run simulation with series dashpot
bing	False	--	Run simulation with binning
testing	True	--	Run simulation in testing mode
auto_dfix	True	--	Auto-fix illegal displacements
multirun	False	--	Run simulation multiple times
Animation Options			
make_ani	True	--	Create animation
save_ani	True	--	Save animation as GIF
auto_frameskip	True	--	Auto-set *frameskip* for 10fps
Basic Parameters			
ea_t	10.0	N	Top spring constant-like value
k_t	100.0	N/m	Side spring constant
H	2.0	m	Total height
Wm	4.0	1	Weibull modulus
n_l	4	1	Number of links
steps	1000 (N/A)	1	Number of displacement values
Binning Parameters			
i_max	10 (N/A)	1	Number of bins
Damage/Plasticity Parameters			
theta_crit	30.0	degrees	Breaking angle
F_slide	10.0 (N/A)	N	Critical sliding force for plasticity
Dashpot Parameters			
c_dash	10.0 (N/A)	N*s/m	Damping coefficient
Dynamics Parameters			
m	1.0	kg	Mass
g	0.0	m/s^2	Acceleration of gravity
dt	0.02	s	Time step
t_stop	20.0	s	Total time to run simulation
Animation Parameters			
xspacing	0.5*H	m	Horizontal spacing between links
fps	50 (N/A)	fps	Animation speed
frameskip	1 (N/A)	1	Frames to skip
fname	“UserExample5.gif”	--	File name for saving GIF
Multirun Parameters			
runs	1 (N/A)	1	# of times to run simulation
Displacements			
d	dfunc.cospath(steps,3,H,0.5)	m	Displacement list

Outputs



Animation GIF "Example6"

Example 7: A 4-Link Dynamic Simulation with Plasticity

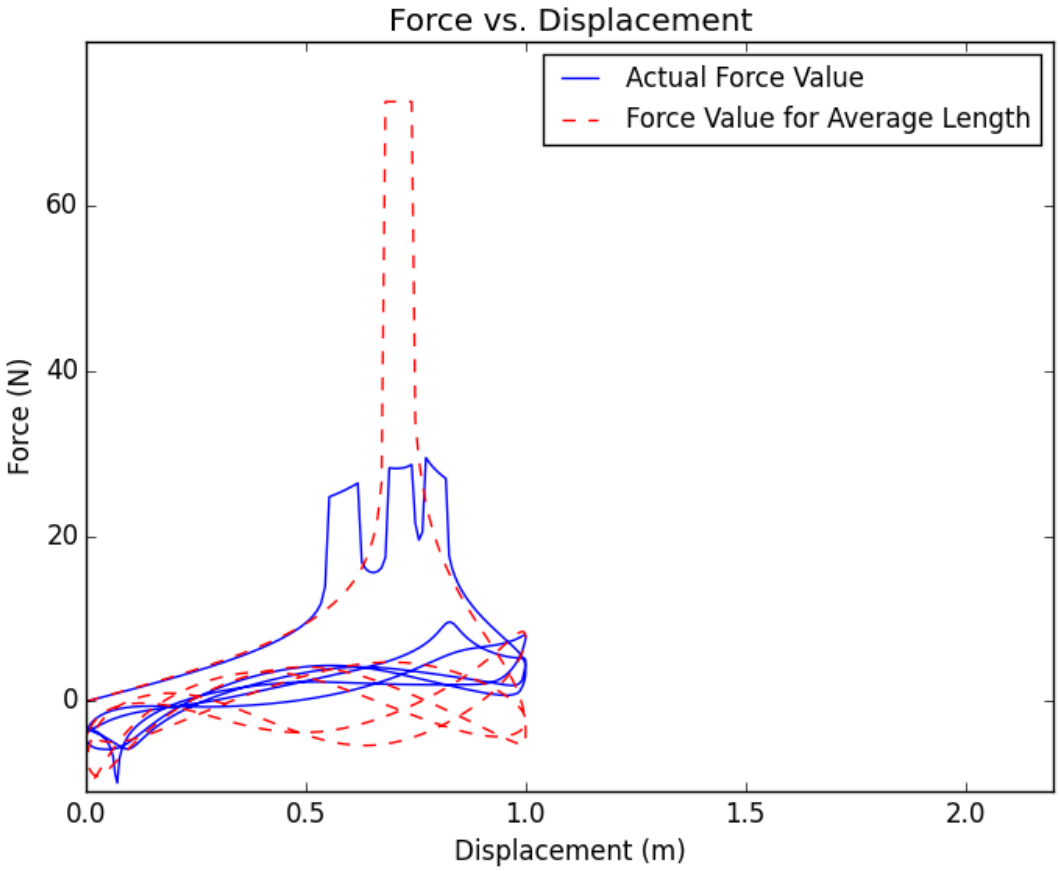
This example shows how to run a dynamic simulation with plasticity, where the displacement follows a sine wave path, and generate an animation.

Inputs

Inputs	***Value***	***Units***	***Description***
Type of Simulation Inputs			
dyn	True	--	Run simulation with dynamics
damage	False	--	Run simulation with damage
plasticity	True	--	Run simulation with plasticity
dashpot_par	False	--	Run simulation with parallel dashpot
dashpot_ser	False	--	Run simulation with series dashpot
bing	False	--	Run simulation with binning
testing	True	--	Run simulation in testing mode
auto_dfix	True	--	Auto-fix illegal displacements

multirun	False	--	Run simulation multiple times
Animation Options			
make_ani	True	--	Create animation
save_ani	False	--	Save animation as GIF
auto_frameskip	False	--	Auto-set *frameskip* for 10fps
Basic Parameters			
ea_t	10.0	N	Top spring constant-like value
k_t	100.0	N/m	Side spring constant
H	2.0	m	Total height
Wm	4.0	1	Weibull modulus
n_l	4	1	Number of links
steps	1000 (N/A)	1	Number of displacement values
Binning Parameters			
i_max	10 (N/A)	1	Number of bins
Damage/Plasticity Parameters			
theta_crit	10.0 (N/A)	degrees	Breaking angle
F_slide	5.0	N	Critical sliding force for plasticity
Dashpot Parameters			
c_dash	10.0 (N/A)	N*s/m	Damping coefficient
Dynamics Parameters			
m	1.0	kg	Mass
g	0.0	m/s^2	Acceleration of gravity
dt	0.02	s	Time step
t_stop	20.0	s	Total time to run simulation
Animation Parameters			
xspacing	0.5*H	m	Horizontal spacing between links
fps	50	fps	Animation speed
frameskip	1	1	Frames to skip
fname	“fn.gif” (N/A)	--	File name for saving GIF
Multirun Parameters			
runs	1 (N/A)	1	# of times to run simulation
Displacements			
d	dfunc.cospath(steps,3,H,0.5)	m	Displacement list

Outputs



Animation GIF "Example7"

Example 8: A 3-Link Dynamic Simulation with a Parallel Dashpot

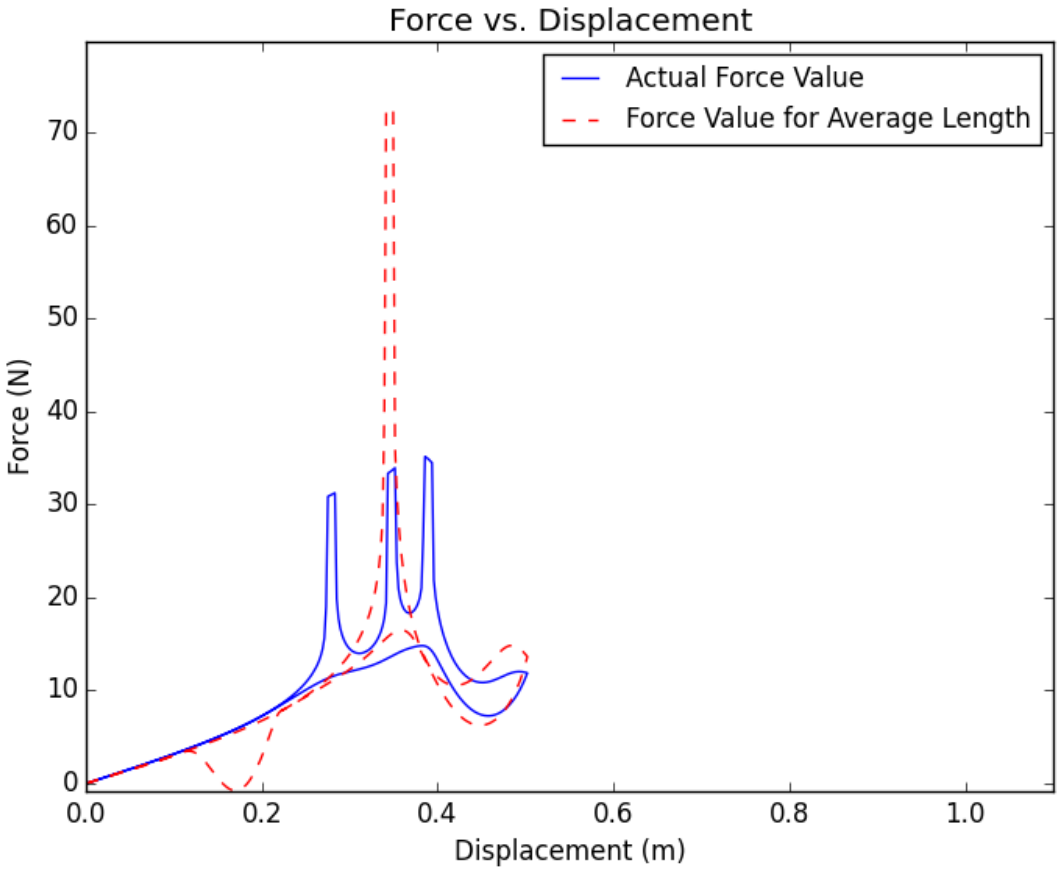
This example shows how to run a dynamic simulation where the side spring has a dashpot in parallel with it, where the displacement steadily increases and then steadily decreases, and generate an animation.

Inputs

Inputs	***Value***	***Units***	***Description***
Type of Simulation Inputs			
dyn	True	--	Run simulation with dynamics
damage	False	--	Run simulation with damage
plasticity	False	--	Run simulation with plasticity
dashpot_par	True	--	Run simulation with parallel dashpot
dashpot_ser	False	--	Run simulation with series dashpot
bing	False	--	Run simulation with binning
testing	True	--	Run simulation in testing mode
auto_dfix	True	--	Auto-fix illegal displacements
multirun	False	--	Run simulation multiple times

Animation Options			
make_ani	True	--	Create animation
save_ani	False	--	Save animation as GIF
auto_frameskip	False	--	Auto-set *frameskip* for 10fps
Basic Parameters			
ea_t	10.0	N	Top spring constant-like value
k_t	100.0	N/m	Side spring constant
H	1.0	m	Total height
Wm	4.0	1	Weibull modulus
n_l	3	1	Number of links
steps	1000 (N/A)	1	Number of displacement values
Binning Parameters			
i_max	10 (N/A)	1	Number of bins
Damage/Plasticity Parameters			
theta_crit	10.0 (N/A)	degrees	Breaking angle
F_slide	10.0 (N/A)		
Dashpot Parameters			
c_dash	5.0	N*s/m	Damping coefficient
Dynamics Parameters			
m	1.0	kg	Mass
g	0.0	m/s^2	Acceleration of gravity
dt	0.02	s	Time step
t_stop	10.0	s	Total time to run simulation
Animation Parameters			
xspacing	0.5*H	m	Horizontal spacing between links
fps	50	fps	Animation speed
frameskip	1	1	Frames to skip
fname	“fn.gif” (N/A)	--	File name for saving GIF
Multirun Parameters			
runs	1 (N/A)	1	# of times to run simulation
Displacements			
d	dfunc.vpath(steps,1,H,0.5)	m	Displacement list

Outputs



Animation GIF "Example8"

Example 9: A 3-Link Dynamic Simulation with a Dashpot in Series

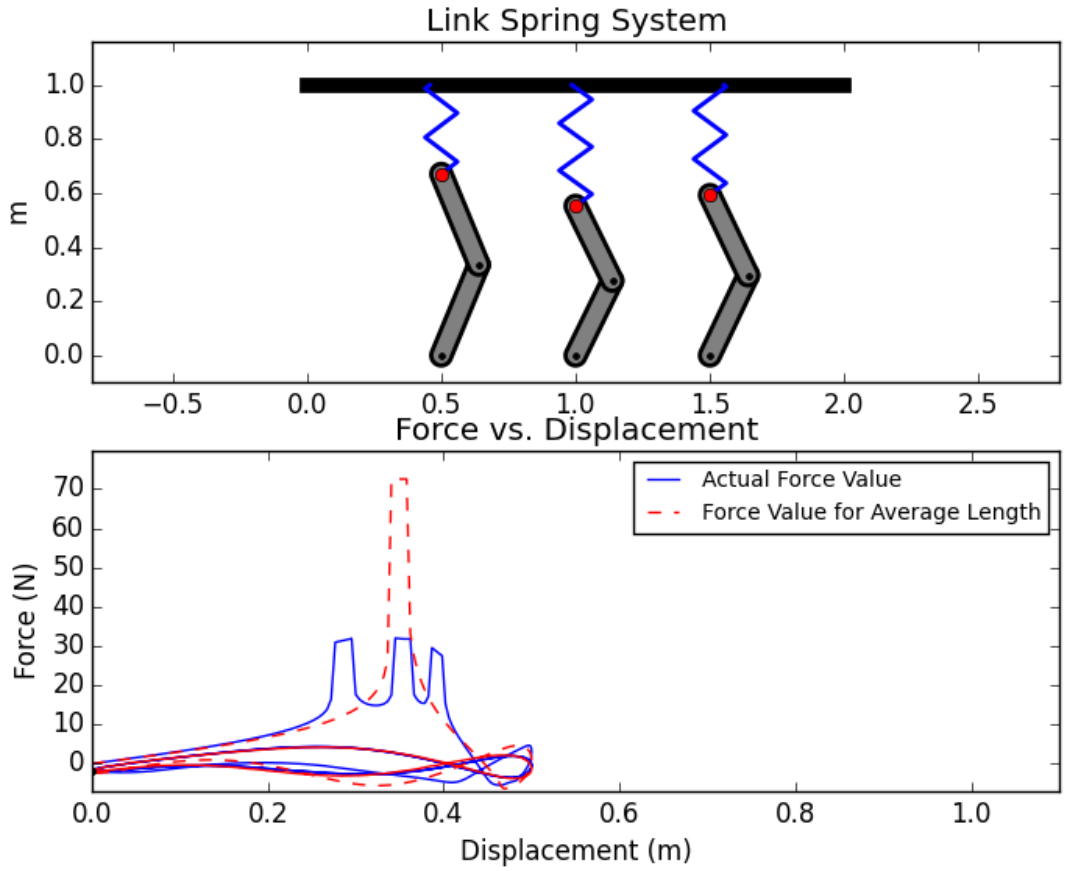
This example shows how to run a simulation where the side spring is attached in series with a dashpot, where the displacement follows a sine wave path, and generate an animation.

Inputs

Inputs	***Value***	***Units***	***Description***
Type of Simulation Inputs			
dyn	True	--	Run simulation with dynamics
damage	False	--	Run simulation with damage
plasticity	False	--	Run simulation with plasticity
dashpot_par	False	--	Run simulation with parallel dashpot
dashpot_ser	True	--	Run simulation with series dashpot
bing	False	--	Run simulation with binning
testing	True	--	Run simulation in testing mode
auto_dfix	True	--	Auto-fix illegal displacements

multirun	False	--	Run simulation multiple times
Animation Options			
make_ani	True	--	Create animation
save_ani	False	--	Save animation as GIF
auto_frameskip	False	--	Auto-set *frameskip* for 10fps
Basic Parameters			
ea_t	10.0	N	Top spring constant-like value
k_t	100.0	N/m	Side spring constant
H	1.0	m	Total height
Wm	4.0	1	Weibull modulus
n_l	3	1	Number of links
steps	1000 (N/A)	1	Number of displacement values
Binning Parameters			
i_max	10 (N/A)	1	Number of bins
Damage/Plasticity Parameters			
theta_crit	10.0 (N/A)	degrees	Breaking angle
F_slide	10.0 (N/A)	N	Critical sliding force for plasticity
Dashpot Parameters			
c_dash	10.0	N*s/m	Damping coefficient
Dynamics Parameters			
m	1.0	kg	Mass
g	0.0	m/s^2	Acceleration of gravity
dt	0.02	s	Time step
t_stop	20.0	s	Total time to run simulation
Animation Parameters			
xspacing	0.5*H	m	Horizontal spacing between links
fps	50	fps	Animation speed
frameskip	1	1	Frames to skip
fname	“fn.gif” (N/A)	--	File name for saving GIF
Multirun Parameters			
runs	1 (N/A)	1	# of times to run simulation
Displacements			
d	dfunc.cospath(steps,3,H,0.5)	m	Displacement list

Outputs



Animation GIF "Example9"

Example 10: Running a Quasistatic Simulation Multiple Times

This example shows how to run 100 5-link quasistatic simulations where the links are steadily completely compressed.

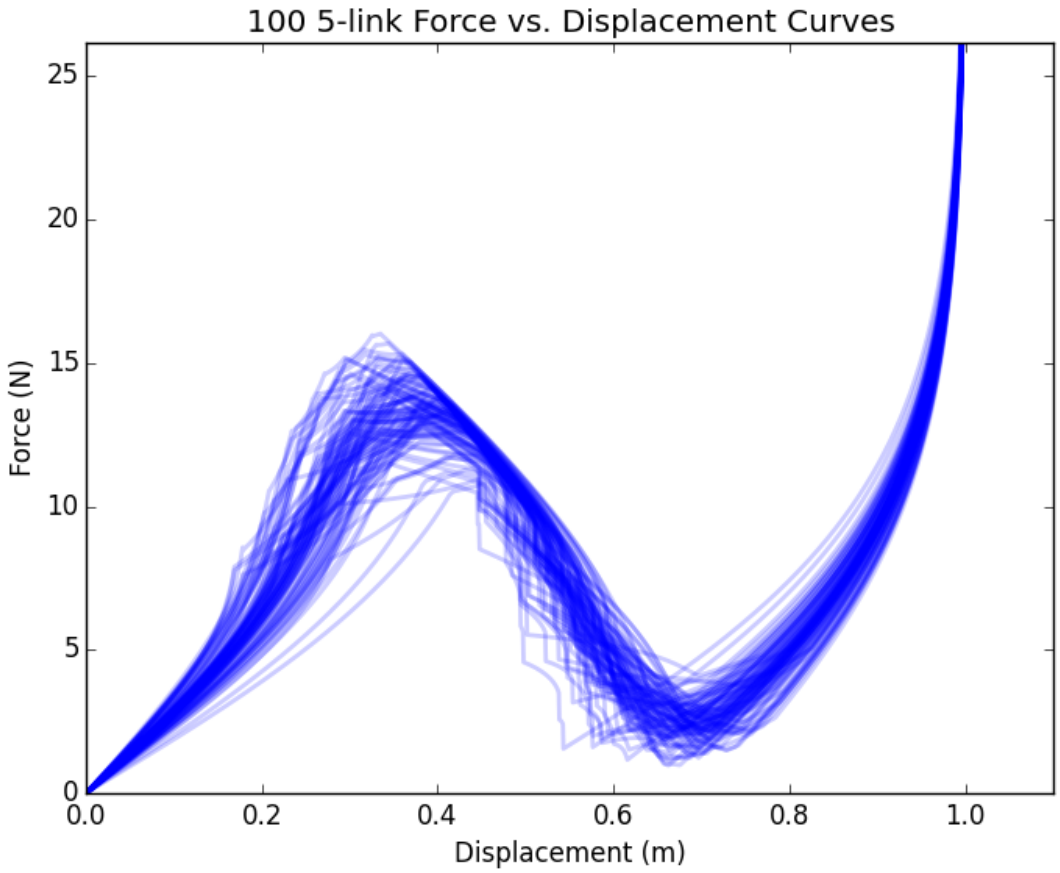
Inputs

Inputs	***Value***	***Units***	***Description***
Type of Simulation Inputs			
dyn	False	--	Run simulation with dynamics
damage	False	--	Run simulation with damage
plasticity	False	--	Run simulation with plasticity
dashpot_par	False	--	Run simulation with parallel dashpot
dashpot_ser	False	--	Run simulation with series dashpot
bing	False	--	Run simulation with binning
testing	False	--	Run simulation in testing mode
auto_dfix	True	--	Auto-fix illegal displacements
multirun	True	--	Run simulation multiple times
Animation Options			

make_ani	False	--	Create animation
save_ani	False	--	Save animation as GIF
auto_frameskip	False	--	Auto-set *frameskip* for 10fps
Basic Parameters			
ea_t	10.0	N	Top spring constant-like value
k_t	100.0	N/m	Side spring constant
H	1.0	m	Total height
Wm	4.0	1	Weibull modulus
n_l	5	1	Number of links
steps	1000	1	Number of displacement values
Binning Parameters			
i_max	10 (N/A)	1	Number of bins
Damage/Plasticity Parameters			
theta_crit	10.0 (N/A)	degrees	Breaking angle
F_slide	10.0 (N/A)	N	Critical sliding force for plasticity
Dashpot Parameters			
c_dash	10.0 (N/A)	N*m/s	Damping coefficient
Dynamics Parameters			
m	1.0 (N/A)	kg	Mass
g	0.0 (N/A)	m/s^2	Acceleration of gravity
dt	0.01 (N/A)	s	Time step
t_stop	10.0 (N/A)	s	Total time to run simulation
Animation Parameters			
xspacing	0.5*H (N/A)	m	Horizontal spacing between links
fps	50 (N/A)	fps	Animation speed
frameskip	1 (N/A)	1	Frames to skip
fname	“fn.gif” (N/A)	--	File name for saving GIF
Multirun Parameters			
runs	100	1	Number of times to run simulation
Displacements			
d	dfunc.linepath(steps,H)	m	Displacement list

Outputs

Note, the user-generated plot will not look exactly like this. However, it should look similar.



Appendix C. Code Details

This appendix details how various functions and parts of the code work.

The Main Code: MasterAnalysis.py

Initialization

After all the imports, the first main section of the code is where the user inputs all the parameters to set up the simulation. The parameters are divided into groups based on function. The first group is where the user sets the type of simulation. There are nine parameters: *dyn*, *damage*, *plasticity*, *dashpot_par*, *dashpot_ser*, *bing*, *testing*, *auto_dfix*, and *multirun*. All of them are Booleans. *dyn* sets whether the simulation will use the dynamic model (*dyn*=True) or the quasistatic model (*dyn*=False). *damage* sets whether damage will be applied (yes if True, no if False). *plasticity* does the same thing form plasticity. *dashpot_ser* and *dashpot_par* set whether to add a dashpot in series or in parallel, respectively. Note that of *plasticity*, *dashpot_par*, and *dashpot_ser*, only one may be set to True for a given simulation. *bing* sets whether binning will be applied. *testing* sets whether or not to run the code in testing mode, which means it will use a predetermined set of values for generating the link lengths, instead of generating a list of random numbers. It is mostly useful for comparing the effects of the different simulation types, where it makes sense to use the same set of link lengths – for example, comparing a run with the effects of damage included to one without. It defaults to using a list of five numbers that had been originally randomly generated, but that can be changed by going to the main analysis function and changing what *R* equals under "if *testing*:" *auto_dfix* sets what to do if the code, when it performs a check on the values of the

user-defined displacement path, finds a value that is greater than or equal to *H*. Having such a displacement value would cause the argument of Equation 17 to be greater than or equal to one, which is out of bounds for the atanh function and would cause an error in running the code, so the code cannot run unless the displacement path is fixed. If *auto_dfix* is set to True, it fixes the problem displacements automatically by setting them equal to 0.9999**H*. If it's set to False, the code will give an error and stop running, prompting the user to go back and fix the displacement path manually. *multirun* sets whether the simulation will run once or multiple times. If set to False, it will run the analysis once, generating a force vs. displacement plot with both the actual force and the force for the average length, and an animation if animation creation is on. If it's set to True, the analysis will be run the specified number of times, and a plot will be generated with all the force vs. displacement curves overlaid, with an opacity of 0.2. It will not plot the curves for the average link lengths or create animations.

The second group is where the user sets whether to make and save an animation. It has three parameters: *make_ani*, *save_ani*, and *auto_frameskip*. If *make_ani*=True, an animation will be generated – otherwise the code will only produce the force vs. displacement plot. If *save_ani*=True, the animation will be saved as a GIF. If *auto_frameskip*=True, if *save_ani* is on as well, the code will automatically set the value of *frameskip* to play the animation in the same amount of time as it plays with the user-inputted *fps* and number of frames, but at 10fps. This is mostly useful if the animation is being saved to be used on a platform that has a maximum fps at which it can play GIFs. It is set to 10fps because that's about what the maximum is for Microsoft PowerPoint.

The third group is the basic parameters used in all the models. It contains the parameters *ea_t*, *k_t*, *H*, *Wm*, *n_l*, and *steps*. *ea_t* and *k_t* are the values of *EA* and *k* before they're normalized. *H* and *Wm* have been previously defined. *n_l* is the number of links to model. *steps* is the total number of displacement values. While it isn't a necessary parameter, if used when defining the displacement path it can make it easier if the user later wants to change the length of the displacement list, because all they have to do is change *steps* and everything else will adjust automatically. If dynamics is turned on, the value of steps will automatically be changed to be *t_stop*/*dt* (since if the dynamic simulation is to go the full time specified, the number of displacement values has to equal the number of time values, which is the total time over the time step).

The fourth group is the parameter for binning. It includes only *i_max*, which is the number of bins to use.

The fifth group is for damage and plasticity. It includes *theta_crit*, which is the angle from vertical (in degrees) at which the links will "break," and *F_slide*, which is the critical side spring force before the Jenkins element slides and plastic deformation effects are applied.

The sixth group is for the dashpots. It contains only the parameter *c_dash*, which is the value of the damping coefficient.

The seventh group is the parameters for the dynamic analysis, and consist of *m*, *g*, *dt*, and *t_stop*. *m* is the mass of the point mass, *g* is the acceleration of gravity, *dt* is the time step to use when solving the dynamic model ODEs, and *t_stop* is the total time for the simulation to run.

The eighth group is the parameters for the animation. It contains *xspacing*, *fps*, *frameskip*, and *fname*. *xspacing* sets the spacing between the links on the animation. *fps* sets the frame rate for the animation, in frames per second. *frameskip* tells the animation function which frames to play and which to skip – for example, if *frameskip*=3, the animation will only play every third frame. If it equals 1, it will play every frame. This parameter is mostly useful for saving GIFs. When *save_ani*=True, the frame rate is automatically set to 10fps, because that's the maximum that Microsoft PowerPoint can play. Setting *frameskip* to a number higher than one makes the GIF play reasonably quickly in PowerPoint without sacrificing the quality of the numeric solutions by having to have a larger step size. If *auto_frameskip* is set to True, the code will automatically find the necessary value of *frameskip* to get the 10fps for

PowerPoint and override the user's input value. *fname* is the filename to save as, in the format "filename.gif".
The ninth group is another one-parameter group, related to running the analysis multiple times. The parameter *runs* determines the number of times to run the analysis.
The tenth group is where the displacement path is set. It can either be done using the function *dfunc*, or the user can define their own displacement path. If the user defines their own displacement path, they must be sure that it has the type 'list'. If the type is "numpy.ndarray" or similar, it can cause problems.

Checking for Input Errors

The next section of the code checks for errors in the input that would cause the code to not work properly. If binning is on, it checks to make sure that the number of bins is not larger than the number of links. If it is, it throws an error. If damage is on, it checks to make sure that *theta_crit* is between 0 and 90 degrees. If not, it throws an error. It checks whether more than one of *plasticity*, *dashpot_par*, and *dashpot_ser* is set to true, and if so it throws an error.
The code can't create an animation if binning or multiple runs are turned on, so if both one of those and animation are turned on the code turns off the animation, displaying a message that it has done so. It will also turn off the animation if the number of links is greater than 50, because that's about the maximum number of links it can animate without them getting too small to see. If saving the animation is turned on but making the animation is not, then saving the animation is automatically turned off.
The code then checks to make sure that all the input parameters that are supposed to have the data type 'float' are in fact floats. If some of them aren't, it changes them to be floats. It does this because it's easy to accidentally enter an integer for a value that's supposed to be a float, which can cause problems with the math, as dividing two integers in Python will produce another integer that is the floor of the decimal number it's supposed to be.

The Main Analysis Function

This is the main part of the code where all the calculation happens. It is written as a function that takes the input parameters *ea_t*, *k_t*, *H*, *Wm*, *n_l*, *d*, *i_max*, *theta_crit*, *F_slide*, *c_dash*, *m*, *g*, *dt*, *t_stop*, *dyn*, *damage*, *plasticity*, *dashpot_par*, *dashpot_ser*, *bing*, and *testing*. It outputs *P*, *P_avg*, *y*, *F_crit_avg*, and *L*, which are, respectively, a list of the applied forces at each displacement, a list of the applied forces for the average link length, a 2D list of the values of *y* for each link at each displacement, the critical buckling force for the average length, and a list of the link lengths.
The first thing the function does is to normalize *ea_t* and *k_t* by dividing them each by the number of links. This ensures that the maximum force does not increase as more links are added, but only by changing the physical parameters.
The function then generates the list of link lengths. It starts by either using Python's built-in *random()* function to generate a list of random numbers *R*, between zero and one, with length equal to the number of links, or, if *testing*=True, it uses a predefined set of numbers (which should also be between zero and one). If binning is turned on, it then uses the *binning* function on *R*. To get the lengths, the code runs R through the Weibull and exponential distribution equations. It then calculates the average length.
Next the function initializes the lists for *P*, *P_avg*, *y*, and *y_avg* as lists of zeros of the size they're supposed to be (doing this is slightly faster than using *.append()* to add values to the lists). It then sets the indicator values to zero. There are two indicators, which tell the state of the links. The first is *c*, which is used to indicate which buckling phase the link is in: 0 for unbuckled, 1 for buckling, and 2 for completely collapsed. The second is *b*, which is used to indicate whether the link is broken (a value of 1), or unbroken (a value of zero). These are both lists with length of the number of links, and both are initialized as all zeros.

Next the critical buckling force is found for each link and the average length, and is stored in *F_crit* and *F_crit_avg*. *theta_crit* is then converted into radians.
If plasticity is turned on, the code finds *x_crit*, which is the maximum horizontal distance *x* which the joint between the links can move before *F_slide* is reached. It then initializes the lists for *x* and *x_plas*. If *dashpot_par* is on, it initializes the lists for *x* and *dx* (which is $\dot{x}$ in the calculations). If *dashpot_ser* is on, it initializes the lists for *x* and *x_dash* (which is *l* in the calculations).
After that, it sets up for numerically solving for the force. If the model is quasistatic and plasticity is not applied, it sets the initial root guesses as the critical buckling values (since those should be very close to what the actual force is the first time the code runs its root finding), and it sets up two functions, root_g and root_gprime, that are the function to find the root of and its derivative, respectively. If plasticity is applied, it defines one function, root_g, which is the function to find the root if plasticity is applied. If the model is dynamic, it simply initializes the velocity as zero.
The analysis function now starts the process of actually calculating the forces for each link at each displacement. It first initializes the list of forces for each link as a list of zeros. If the link is in the unbuckled state, the function first checks to see if the displacement value has jumped past where the top of the link is, and thus also jumped past where the top spring became so compressed its force approached infinity (and so definitely reached the critical buckling force). If so, it sets the link to the buckling state. Otherwise, it calculates the force applied to the link and stores it, and stores the *y* value as zero. It then checks to see whether the applied force is greater than the critical buckling force if the link is unbroken, or greater than zero if the link is broken. If so, then it sets the link to the buckling stage.
If the link is in the buckling stage, if it's using the quasistatic model and is unbroken, first *zerotol* is set. If the link is very short, *zerotol* is set to be larger than for a longer link, to prevent problematic small links from crashing the code. See The Root Finding Function: MyRootFinding.py for more details on *zerotol*. Then the force is found. If plasticity is not on, the arguments for the *MyRootFinding.newton_raphson* function are set to the current values of *d*, *L*, *k*, *H*, and *ea*, in that order. Then, *MyRootFinding.newton_raphson* is used to find the force. If plasticity is on, the current displacement of the side spring is found. The arguments for *MyRootFinding.bisection* are set to the current values of *d*, *L*, *k*, *H*, *ea*, and *x_plas*, in that order. Then the boundaries for the bisection method are set. The lower boundary is set to the larger of zero or the smallest possible *y* value that would still let atanh be defined, and the upper boundary is set to the largest possible *y* value that would still let atanh be defined. Then *MyRootFinding.bisection* is used to find the new value of *y*. Once the root finding function is run, if plasticity is on, the new force is found by using *y* in Equation 29. Then the new value of *x* is calculated. If the side spring's displacement is larger than *x_crit*, the distance *x_plas* the block has slid is updated to *x-x_crit*. If the side spring's displacement is smaller than *-x_crit*, the distance *x_plas* the block has slid is updated to *x+x_crit*. If plasticity is not on, the new force is saved and *y* is calculated and stored. If the link is broken, the force is set to zero and *y* is set equal to *d*.
If the model is dynamic and the link is unbroken, the 4th order Runge Kutta method is implemented using one of the functions in to find *y* and *v*. If *dashpot_par* is on, it saves the previous value of *x* and finds the value of the replacement for *x_plas*, as mentioned in Section 8. Viscosity - Dashpot in Parallel. It then runs *RK4Dyn.RK4_plaslink* with that replacement value as the final argument. After that it finds the new value of *x*, and uses that to calculate the new value of *dx*. If *dashpot_ser* is on, it runs *RK4Dyn.RK4_plaslink* with *x_dash* in place of *x_plas*. It then finds the new value of *x*, and uses that value in *RK4Dyn.RK4_dashser* to find the new value of *x_dash*. If plasticity is on, the displacement of the side spring is calculated, *RK4Dyn.RK4_plaslink* is run, and *x* and *x_plas* are found the same way as for the quasistatic model. The one difference is that if *y* ends up being less than zero, *x* is forced to be zero, and if *y* ends up being greater than *L*, *x* is forced to be *L*/2. If plasticity is off, the code just runs *RK4Dyn.RK4_link*. If the link is broken, the code runs *RK4Dyn.RK4_link*, but passing 0 into the Runge

Kutta function in place of *k*. (The functions for plasticity and the dashpot in series end up being the same as the one without if *k*=0, as does the function for the dashpot in parallel if the dashpot is assumed to also break, so only *RK4_link* is run.) It then makes sure *y* hasn't caused the argument for atanh to go out-of-bounds. If it has, it corrects *y* so the argument is within bounds, and then calculates the force. Otherwise it calculates the force without correcting *y*. After that, if damage is on, it checks to see if the link has broken by finding θ and checking to see if it's past the critical breaking angle. If it is, the link's break indicator is set to broken. If *y* has stepped to a negative value, making the argument for acos greater than one, theta is set to zero. Next, the function checks to see whether the link's state has changed. If *y* has become greater than or equal to the link length, *y* is corrected to be the link length, the link is set to being completely collapsed, the velocity is set to zero if the model is dynamic, and the force is re-calculated using Equation 17. If *y* has become less than or equal to zero, or, if plasticity is on and the model is quasistatic, if *y* is greater than ten times the previous *y* and the previous *y* is not zero, *y* is corrected to be zero, the link is set to being unbuckled, and the force is recalculated using Equation 10. If the argument for atanh is greater than or equal to one, it is corrected to be the largest possible value the computer can calculate atanh with. The unusual condition for switching to unbuckled if the model is quasistatic and has plasticity is due to the function used in root finding actually having two roots, near opposite ends of the domain. The *bisection* function is set up so it finds the correct one, but if *y* goes below zero it is outside the domain, and the larger root is found, so the value of *y* increases greatly.

If the link is completely collapsed, *y* is set to be the length of the link and the force is calculated using Equation 17. For dynamics, if the calculated force is less than –m*g (i.e. if the force can overcome gravity), then the link is set to buckling. For quasistatic, if the calculated force is less than zero, the link is set to buckling.

Once the code has calculated the force from each link, it adds them together. If binning is applied, it does this by initializing the total force from that displacement as zero and then looping through each link, multiplying its force by its binning weight and the total number of links and adding it to the previous value. Otherwise the link forces are all just summed. The total force value is stored in *P*.

The function then repeats the entire force-finding process for the average link length. Once it finds the force from one link of the average length, it multiplies it by the total number of links to get the total force for the average length, which is then stored in *P_avg*.

Running the Simulation

The first thing the code has to do before running the simulation is check to make sure none of the displacement values are greater than or equal to *H*, because if any are it will make the argument for atanh in the force calculations greater than or equal to one and cause an error. If *auto_dfix* is set to true, any displacements that are too large will be set to .9999**H*. Otherwise the code will raise an error, so the user knows to fix the displacements manually.

Now the code runs the analysis function. If *multirun* is set to True, it runs the analysis *runs* times, and saves the force lists and *F_crit*_avg for each run. Otherwise it runs the function once and saves *P*, *P_avg*, *y*, *L*, and *F_crit_avg*.

The code then plots the force vs. displacement curves. If *multirun* is true, it plots all the saved curves on the same plot with an opacity of .2. Otherwise it plots the curves for the distribution of lengths and the average length on the same plot. It also prints to the console the time it takes to run the code.

Animating

If making an animation is turned on, the code will run the function *ani.ani* to animate the links. If *save_ani* is turned on as well, the code will also save the animation as a GIF using Imagemagick and print to the console the amount of time it took to save.

The Binning Function: binning.py

This function follows step-by-step the algorithm outlined in "An Efficient Binning Scheme with Application to Statistical Crack Mechanics" (Huq, F., L. Graham-Brady, and R. Brannon. *Int. J. Num. Meth. Engr.*). Its inputs are *r_b* and *i_max*, which are the list of points to be binned and the number of bins, respectively. Its outputs are *W_b*, *r_b*, and *n*. *W_b* is the weight of each binned point, *r_b* is the list of binned points, and *n* is the new number of points.

The Root Finding Function: MyRootFinding.py

This function includes two sub-functions: *newton_raphson* and *bisection*. *newton_raphson* uses the Newton-Raphson root finding method to find the zero of a function. Its inputs are *f*, *x0*, *fprime*, *args* (optional, default ()), *tol* (optional, default .001), *Imax* (optional, default 1000), and *zerotol* (optional, default .01). *f* is the function to find the root of, *x0* is the initial guess for the location of the root, *fprime* is the derivative of *f*, *args* is any additional parameters needed for *f* or *fprime*, *tol* is the desired error tolerance (used as a stopping criterion), *Imax* is the maximum number of iterations (also used as a stopping criterion), and *zerotol* is how close to zero a number should be to be considered to be essentially zero. Its output is the location of the zero, or, if the zero is not found, the output is various information designed to be helpful in figuring out why the zero wasn't found.

First, the function initializes the error as 10**tol* (to make sure the error is big enough to start with), sets the counter *I* as 0, and calculates the initial value of the derivative.

In an attempt to help the method converge faster, if the initial value of the derivative is very small, the initial guess changed to 1.5**x0*, to try to get a point with a larger slope so the second guess won't end up far away from the actual value.

The function then goes into the loop that performs the Newton-Raphson method. First it stores the current guess value. Next it computes the values of the *f* and *fprime* at the current guess. Then it computes the next guess, and the approximate relative error. Finally it increments the counter variable. It repeats this until either the approximate relative error is smaller than *tol* or the maximum number of iterations is reached.

Once the loop is finished, the function checks whether the maximum number of iterations was reached. If it wasn't, the value found for the location of the root is returned. If it was, then the method didn't converge. This almost always happens because occasionally, for some link lengths, *f* winds up looking like Figure 11, where it never quite touches the x-axis.

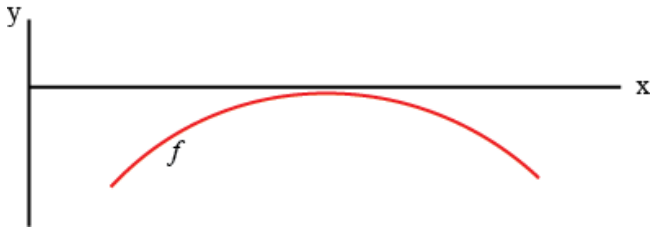


Figure 11: Example plot of f *most times the method didn't converge*

This happens when the link is on the verge of collapse, most likely because the displacement has stepped past collapse, just enough that *f(x)* = 0 no longer has a solution. However, the peak of that curve is still very close to zero, and the last guess for the location of the root is usually near that peak. This sort of nonconvergence happens often enough that there is a special exception for it built into the code. It will calculate the value of *f* at the last guess for the root location, and if the absolute value of that value of *f* is less than *zerotol*, the value is assumed to be effectively zero and function returns the location of the root. If, however, the absolute value of that value of *f* is greater than zero, the function will try to find the peak by creating a list of evenly spaced values between 0 and 2**x0*, evaluating *f* at each of those values, and

finding the maximum *f* value. If the absolute value of the maximum is less than *zerotol*, then the location of the maximum is returned as the location of the root. If it is not, then the function prints the arguments *args* to the console and displays a plot of *f* over the region [0, 2**x0*], with *x0* marked with a vertical line, so the user can try to troubleshoot what went wrong. If the function has to perform any of these exceptions, it will also print to the console what it's doing, for the user's information.
bisection uses the bisection method to find the root of a function. Its inputs are *f*, *xl*, *xu*, *args* (optional, default ()), *tol* (optional, default .001), and *Imax* (optional, default 1000). *f* is the function to find the root of, *xl* and *xu* are the lower and upper bounds, respectively, of the interval containing the root, and *args*, *tol*, and *Imax* are the same as in *newton_raphson*. Its output is *x*, which is the location of the root of *f*. The code first calculates the value of *f* at *xl*. It then initializes the error and the iteration counter, and sets the initial guess for *x* as *xl*, so it has some value to start. It then enters a while loop, which continues either until the maximum number of iterations is reached or the error gets small enough. The first thing the code does in the loop is save the previous guess for *x*. It then sets the new guess for *x* as halfway between *xl* and *xu* and evaluates *f* at the new guess. If that function value has the same sign as the function value of the lower bound, then the root is not between them and the lower bound is moved to the guess value, and the function value at the guess is stored as the new lower bound *f* value. If the function values have opposite signs, then the root is between them and the upper bound is moved to the guess value. If the guess is zero, then the root has been found and the code breaks out of the loop. The iteration counter is then incremented and the approximate relative error is calculated. After the while loop finishes, if the maximum number of iterations is reached, it displays "root not found." Otherwise it returns *x*.

The Runge Kutta ODE Solver: RK4Dyn.py

This function uses the 4th order Runge Kutta method to solve the dynamic model's ODE. It has four sub-functions: *RK4_point*, *RK4_link*, *RK4_plaslink* and *RK4_dashser*. The first three use the previous displacement as the beginning of their steps, and the current displacement as the end of their steps, to find the current *y* value and velocity *v*. They all have the inputs *dt*, *u0*, *u1*, *y0*, *v0*, *H*, *L*, *k*, *ea*, *m*, and *g*. *RK4_plaslink* has the additional input *x_plas*. *H*, *L*, *k*, *ea*, *m*, and *g* are described in The Main Code: MasterAnalysis.py – Initialization. *u0* and *u1* are the previous and current displacements, respectively. *y0* and *v0* are the previous *y* value and velocity. *x_plas* is the distance the Jenkins element has slid for plastic deformation. The outputs are *y1* and *v1*, which are the current *y* value and velocity, respectively.
RK4_dashser is used to find the new value of the dashpot displacement for the dashpot in series. Its inputs are *l0*, *dt*, *k*, *x*, and *c*, which are the initial dashpot displacement, the time step, the side spring constant, the displacement of the other end of the spring, and the damping coefficient, respectively. It outputs *l1*, the new dashpot displacement.
For the first three functions, the first thing they do is define the step sizes. They use two, one for the displacement, *du*, defined as the difference between the beginning and end displacements, and one for time, *h*, defined as *dt*. They then use the Runge Kutta method to find both *v* and *y*.
If for *RK4_point* Equation 28 is called *a*, for *RK4_link* Equation 48 is called *a*, for *RK4_plaslink* Equation 54 is called *a*, and for all three functions $b = -\frac{dy}{dt} = v$, then the K values for finding *v* become $K_{v1} = a(u0, y0, v0)$, $K_{v2} = a\left(u0 + \frac{du}{2}, y0 + K_{y1}\frac{h}{2}, v0 + K_{v1}\frac{h}{2}\right)$, $K_{v3} = a\left(u0 + \frac{du}{2}, y0 + K_{y2}\frac{h}{2}, v0 + K_{v2}\frac{h}{2}\right)$, and $K_{v4} = a\left(u1, y0 + K_{y3}h, v0 + K_{v3}h\right)$, and the K values for finding *y* become $K_{y1} = -b(v0)$, $K_{y2} = -b\left(v0 + K_{v1}\frac{h}{2}\right)$, $K_{y3} = -b\left(v0 + K_{v2}\frac{h}{2}\right)$, and $K_{y4} = -b(v0 + K_{v3}h)$.
All three functions calculate the K values in order, doing K_1 for both *v* and *y*, then K_2 for both, and so on. If any argument for *atanh* steps out of bounds, it is corrected to the largest or smallest possible value. For *RK4_plaslink*, if the argument for *sqrt* steps out of bounds, it is corrected to zero. The functions then find *y* and *v*, and return them as *y1* and *v1*.

If for *RK4_dashser* Equation 58 is called a, then $K_{l1} = a(l0)$, $K_{l2} = a\left(l0 + K_{l1}\frac{h}{2}\right)$, $K_{l3} = a\left(l0 + K_{l2}\frac{h}{2}\right)$, and $K_{l4} = a(l0 + K_{l3}h)$. It calculates the *K* values and uses them to find the new *l*, which it returns as *l1*.

While the code does not have a setting to switch between *RK4_point* and *RK4_link*, changing one to the other is relatively simple. Just find the four places in the code where this function is used (two in the link distribution section and two in the average link length section) and change *RK4_link* to *RK4_point*. Note that plasticity and viscosity are only applied to the situation where the links have mass, not where the mass is a point mass.

The Displacement Path Functions: dfunc.py

The functions contained in *dfunc* are designed to create a few possible lists of displacement values. The names are based on what shape a plot of the displacement vs. time (or step number) would have.

linepath is the most basic. Its inputs are *s*, *H*, and *d0* (optional, default 0), where *s* is the number of displacement values, *H* is the total distance to displace, and *d0* is the starting displacement value. Its output is a list of displacements, of length *s*, that goes steadily between *d0* and *H*.

vpath takes inputs *s*, *vs*, *H*, *per*, and *d0* (optional, default 0), where *s*, and *d0* are the same as in *linepath*, *H* is the total possible displacement, *vs* is the number of v's to make (can be a whole or half number, e.g. 2 or 2.5), and *per* is the fraction of *H* to displace. The output is a list of displacements, of length *s*, that goes steadily down to *per*H* and then steadily up to *d0*, *vs* times. If plotted vs. time, it looks like a series of v's, or a saw tooth pattern.

cospath takes inputs *s*, *w*, *H*, *per*, and *d0* (optional, default 0), where *s*, *w*, *per*, and *H* are the same as in *vpath*, and *w* is the number of cycles. It outputs a list of displacements, of length *s*, that when plotted vs. time is an inverted cosine that is shifted up to start at zero, goes to a peak height of *per*H*, and goes through *w* cycles.

pausepath takes the inputs *s* and *dval*, where *s* is the same as the other functions and *dval* is the displacement value to pause at. The output is a list of displacements, of length *s*, that all have the value *dval*, causing the displacement to stay the same for a time.

These four functions can be used individually to create a displacement path, or they can be combined. If using a combined function, it's a good idea to make sure it's continuous. A discontinuous combined function will work ok in the quasistatic model, but causes problems with the dynamic model if the discontinuity is large. The best way to combine them is to type something like d1 = *dfunc.linepath*(500,.5), d2 = *dfunc.pausepath*(500,.5), d = d1 + d2. This will create a displacement list that goes steadily from 0 to .5 in 500 steps and then pauses at .5 for 500 steps. d = d1 + d2 only works if d1 and d2 are lists. All the *dfunc* functions output lists, but if making a custom function the user must be sure the output is a list.

The Animation Function: ani.py

This function animates the links. Its inputs are *L*, *y*, *H*, *n*, *d*, *P*, *P_avg*, *xspacing*, *F_crit_avg*, *save_ani*, *fps* (optional, default 50), *frameskip* (optional, default 1), *dyn* (optional, default False), and *t_step* (optional, default .1), which are all described in The Main Code: MasterAnalysis.py - Initialization. Its output is an animation of the links.

The first part of the *ani* function is the *draw* function, which finds all the points and creates all the paths needed to animate the links. Its inputs are *L*, *y*, *H*, *n*, *d*, and *xspacing*. It outputs *bot_path*, *top_path*, *xybar*, *spr1x*, *spr1y*, *dots*, and *mass*. *bot_path*, and *top_path* are paths created by Matplotlib's *Path* function, to be used to draw the patches for the bottom and top links, respectively. *xybar* is the coordinates for the ends of the line that represents the top plate. *spr1x* and *spr1y* are the x- and y- coordinates of the

endpoints and bend points of the springs. *dots* is the coordinates for the dots at the endpoints of each link, and *mass* is the coordinates of the locations of the mass dots for each link.
The first thing *draw* does is set the link width to be .075*H*. It then uses *xspacing* to find the x-coordinate of the bottom of each link, labeled *spx*. It finds the displacement angle θ for each link, setting it to $\pi/2$ if θ is greater than $\pi/2$. Then, using θ and basic trigonometry, it finds the endpoints of the links. For the bottom of the bottom link, the x-coordinate is just *spx*, and the y-coordinate is zero. For the joint between the links, basic trigonometry says that its x-coordinate is $spx + \frac{L}{2}\sin\theta$ and its y-coordinate is $\frac{L}{2}\cos\theta$. For the top of the top link, the x-coordinate is also *spx*, and the y-coordinate is $L - y$.
Next, it finds the points outlining the links. To get the semicircles at the end of each link, first it finds what the coordinates of the points on the semicircle would be if the semicircle were centered at (0,0). For the top link, the top semicircle goes from θ to $\theta + \pi$ and the bottom one goes from $\theta + \pi$ to $\theta + 2\pi$, as shown in Figure 12. For the bottom link, the top semicircle goes from $-\theta$ to $-\theta + \pi$, and the bottom one goes from $-\theta + \pi$ to $-\theta + 2\pi$, as shown in Figure 13.

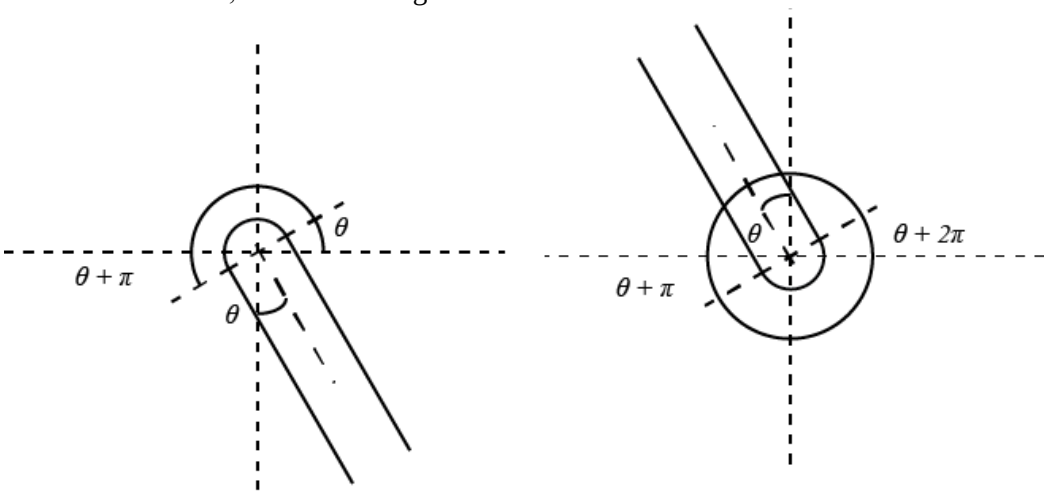


Figure 12: Illustration of the top link's semicircle angles

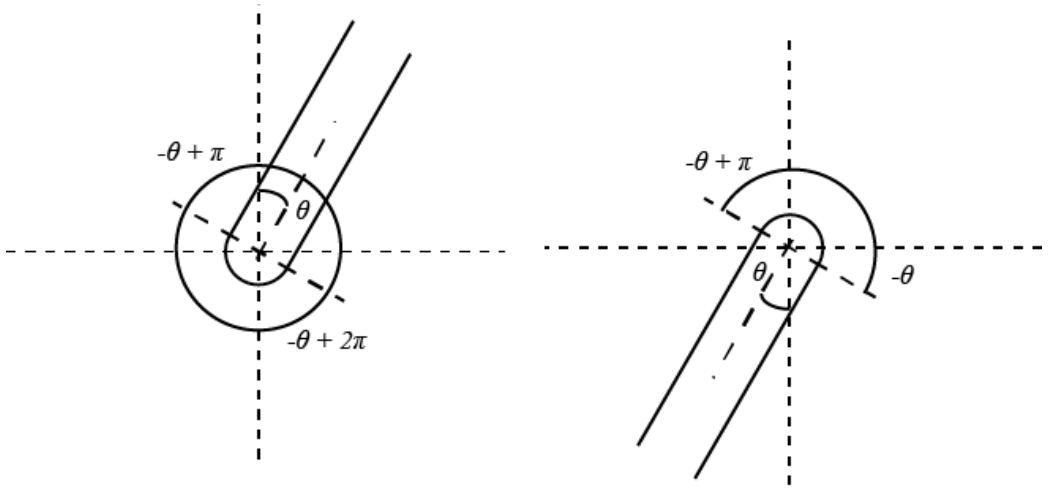


Figure 13: Illustration of the bottom link's semicircle angles

Once the semicircles centered about the origin are found, each is then translated to its actual location, with its center at the corresponding link endpoint. Then, for each link, the lists of points for the top and

bottom semicircles are combined into one list, with the first point in the list repeated at the end for Matplotlib's *Path*. The individual link lists are then combined into one list for all the top links and one list for all the bottom links. Next, the function makes a list of *Path* movement codes. The movement codes consist of one MOVETO to move to the first point of that link's list of points, LINETOs for all other points for that link except the last one, and a CLOSEPOLY for the last point, repeated for each link. Next Matplotlib's *Path* function uses the lists of points and the list of movement codes to make a path for the top links and a path for the bottom links, *top_path* and *bot_path* respectively, which are returned at the end of the *draw* function.

The *draw* function then finds the endpoints and bend points to plot the line for each spring. The first thing it does is find the number of spring segments. Each spring segment (the part between the bend points) has length twice the width of the link, and the spring is drawn so that initially each spring segment makes a 30° angle with the horizontal. (See Figure 14 for illustration. References to dots in relation to spring calculations refer to the small black dots in the figure.)

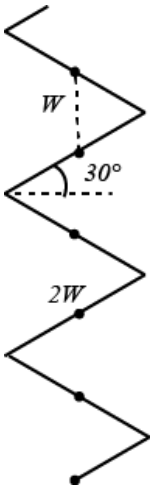


Figure 14: Illustration of the spring

This means that the number of segments (marked by the small black dots) is the length of the spring ($H - L$) divided by W, rounded up to the nearest whole number.

The code then has to find the length of the part of the spring above the topmost dot, which is $H - L - W$*(the number of dots – 1). This length is denoted *lb0*.

As the spring compresses, the function has to determine how its length has changed. The total length of the spring at a given total displacement is $H - L - d + y$. The distance between each dot is a fraction of that. To determine that fraction, the code first has to find the new length of the extra bit, *lb*, which is the fraction *lb0* was of the original total length multiplied by the current total length. It subtracts *lb* from the total length and divides that by the number of segments – 1 to get the new distance between the dots.

After it has that, it starts making the list of endpoints and bend points. The bottom endpoint is the endpoint of the top link. The x-coordinates of each bend point are the coordinate of the bottom endpoint ± $W \cos \alpha$, where α is the angle the segment makes with the horizontal and is equal to $\sin^{-1} \frac{D}{2W}$. Whether it is + or – alternates. The y-coordinate of each bend point is the coordinate for the link endpoint + $D/2$ for the first bend, and the previous y-coordinate + D for all other bends, until the bend below the topmost dot is reached.

Once the coordinates for the bend below the topmost dot are found, the code then determines whether the extra bit contains a bend or not by checking whether the remainder of the original length of the spring

divided by *W* is greater than or less than 0.5. If less than, then there is no extra bend, so the x-coordinate of the endpoint is the coordinate of the bottom endpoint $\pm \frac{lb}{\tan(\sin^{-1}(D/2W))}$ and the y-coordinate is $H - d$. If the remainder is greater than .5, there is an extra bend, whose coordinates are found like the others. The endpoint's x-coordinate is then the previous x-coordinate $\pm \frac{lb-(D/2)}{\tan(\sin^{-1}(D/2W))}$, and y-coordinate is $H - d$.

When the function makes the lists of spring coordinates, each spring is kept in a separate list. In order to be able to plot them easily, the code combines all the lists into two long lists, one for all the x-coordinates and one for all the y-coordinates, adding a 'nan' in between each link's set of points to avoid having lines plotted between the springs. These lists are returned as *spr1x* for the x-coordinates and *spr1y* for the y-coordinates.

The top plate is drawn as a very thick line. The function sets its endpoints to be (0, $H - d$), and (*xspacing*(n+1), $H - d$), so the line is drawn across the tops of all the springs. These coordinates are returned as *xybar*.

The function also puts dots at the endpoints of the links, for aesthetic purposes. Their coordinates are set to be the link endpoint coordinates, and are returned as *dots*.

Last the *draw* function has to find the coordinates of the masses. If dynamics is turned on, these coordinates are set to be the coordinates of the top endpoints of the links. If dynamics is turned off, the coordinates are set to be empty lists, so the masses won't be drawn. The mass coordinates are returned as *mass*.

In order to actually animate the links, the *ani* function first has to set up the figure. The figure has two subplots, one on top of the other. The top plot is where the links are animated. Its axes are set so that all the links will fit on the plot, and the two axes are equally scaled, so the links aren't warped. For a figure with two subplots, this means setting the total length of the y-axis to be 7/20ths the total length of the x-axis. For the bottom plot, where the force vs. displacement curve is, the axes are set to be the same as the non-animated plot.

After that, all the patches and lines are set up, with white patches off-screen and empty lines. In addition to the lines and patches output by the *draw* function, a blank white rectangle patch the size of the link plot is initialized, for clearing the screen between frames. Lines for the force vs. displacement curves for the distribution of lengths and the average length are also initialized. The figure is then labeled.

The *init* function for the animation is then set up, with all the lines and patches still blank, and it returns everything that will be drawn in the order in which it will be drawn. Then the actual animation function is set up. It first adds the plot-clearing patch, then runs the *draw* function to get the data for drawing everything else. It sets the data in the lines for the top plate and the springs, and draws the link patches. It then sets the data for the dots and the masses, sets the data for the force vs. displacement curves to plot them up to the current displacement, and plots a dot at the end of each force vs. displacement curve. The *animate* function then returns all the lines and patches, in the order in which they will be drawn.

The *ani* function then finds the time interval between frames, in milliseconds. If dynamics is on, the interval is *dt**1000. If dynamics is not on, the interval is 1/*fps*. For both cases, if the interval multiplied by the frameskip value is less than 100ms and the animation is being saved, the interval is automatically set to 100ms (which translates to 10fps, the maximum speed at which Microsoft PowerPoint can play a GIF). The *ani* function then creates the animation, with one frame for each *frameskip*th displacement value. The animation is saved as *anim*, and is returned by the function.