Analysis and Design of a Resonant Robotic Rolling Polyhedron

Masters of Engineering Project
Patrick J. Lingane
Advisor: Professor Hod Lipson
Department of Mechanical and Aerospace Engineering, Cornell University

May 21, 2010

Abstract

As a way to create an extremely durable form of locomotion, a polyhedron is caused to roll using internal resonance. This paper outlines the analysis of such a polyhedron and the design of the best possible polyhedron for rolling, including all of its geometries and dynamics. A polyhedron such as the one described here will be able to move by actively moving a mass inside it suspended by springs from the frame. Such a mass will always move in an ellipse, and this mass will also hold all the electronics and actuators for the system. A theoretical analysis of such a system is completed here for the case where the edges of the polyhedron do not leave the ground, and the coordinates of frame of reference do not tilt. This system is then optimized using a genetic algorithm. A computational simulation is created where the object is free to roll about and where the forces acting on it are much more similar to what they would be in the real world, and this is further optimized. Finally, a prototype is built. Although the prototype reproduces some prior results of hopping in a forward direction, rolling was still not achieved.

Table of Contents

Introduction

The goal of the project is to create a polyhedron capable of moving itself using only an internal resonating mass. The advantage of such a form of locomotion is that when enclosed, this robot will be able to travel in very adverse conditions where other robots with joints and external moving parts would clog and get stuck. Such a robot could potentially be used under water or in some other fluid or plasma, and the control electronics and actuators would be completely shielded from electrical disturbances.

There are three modes by which such a polyhedron may move. It may roll onto consecutive faces, it may repeatedly hop forward, or it may slide and shuffle. Of the three modes, the first two are more controllable, but only the second two have so far been demonstrated. Thus the goal of my portion of the project is to create a polyhedron capable of rolling using an internal resonating mass.

Previous work on this project was completed by Jonas Neubert, who successfully made a tetrahedron hop forward in a controlled way. Originally his goal was to create the two forms of controllable motion, rolling and hopping, but the former proved to be more difficult. The system used was a central moving mass with actuators attached, where each actuator was attached through a sprung cable to a vertex of the polyhedron. To minimize the number of actuators, a tetrahedral frame was used.

Through iterative simulation using a genetic algorithm, the motion of these actuators was optimized for both rolling and hopping. Both sets of iterations produced working results in simulation. In both cases, the speed of motion was optimized. The speed of the hopping motion in simulation was 4 meters per minute.

When this control was implemented in the robot, the hopping algorithm produced working results but the rolling algorithm did not. The hopping algorithm produced motion with a speed of 2 meters per minute. The rolling algorithm caused the robot to slide but not roll. Some possibilities of future work were presented as possibly increasing the friction or otherwise enabling the robot to roll, as well as increasing the speed of hopping.

The present work is much the same as the prior work, and the design will take the same general form. This work includes a theoretical analysis of a system, theoretical optimization, computational simulation, optimization of the simulation, and finally construction of a prototype. In each case, the optimized parameter was some measure of how well the system would roll. Hopping is ignored in this project and is considered undesirable as the system is rolling. Every parameter is examined including the number of sides of the polyhedron and the mass of each component. Once reasonable limits are set for each parameter, the system is optimized to be able to roll without slipping.

Analytical Calculations

In this analysis the polyhedron is constructed of an outer frame inside which a mass is able to move about. This moving mass is connected to the frame by strings and springs in series such that each spring is connected to the frame and each string is connected to an actuator on the moving mass. These actuators are servos with pulleys attached, and they are able to pull on each string individually to

actuate the moving mass in any direction. Further, because the springs are arranged symmetrically and are all of equal characteristics, an equal spring constant will act on the moving mass in every direction. Therefore, there is a single natural frequency at which the moving mass may oscillate around its rest position which will be (disregarding gravity) at the frame's centroid. It will always oscillate in an ellipse around that point, since all oscillating bodies oscillate in ellipses within the coordinates in which they have freedom. Since this system has three degrees of freedom (disregarding translation of the frame and all rotation) then the ellipse will occur in any plane that intersects the rest position.

A first step to understanding how such a polyhedron could roll is an analysis of the forces and moments necessary to make it roll, and what conditions must be met. Two free body diagrams of the system are shown in Figure 1 to help illustrate what forces act on the bodies. Note that the spring force F_s acts on both objects equally but in opposite directions. The spring force is applied to the frame through the springs and acts on the frame from the center of mass of the moving mass. The gravitational force on the frame is applied at the center of mass of the frame.

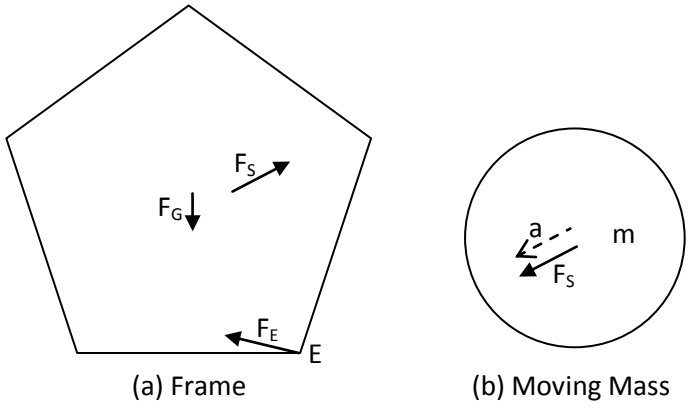


Figure 1: Free body diagrams of components

The moving mass is assumed to be accelerating relative to the inertial coordinate system and the frame is assumed to be fixed relative to the inertial coordinate system. This is no longer true when the moment applied by the spring and gravity forces about the edge is enough to cause rolling. Note that this analysis is therefore only valid when the frame is in contact with the ground. It is still possible however to determine if the frame will start to tip up for a given set of parameters. In Figure 1, the frame is shown rolling to the right, and thus it is rotating in the negative direction. Thus the moment applied must be negative. Because the frame has no acceleration, the sum of forces acting upon it must equal zero and the force on the edge about which it rolls, F_E emerges.

In order that the frame may roll, three conditions must be met.

1. There must be enough force F_S applied on the frame to make a negative moment about E.
2. The edge E about which the polyhedron is rolling must not slip on the ground.
3. The frame must not roll backward.

Condition #1: Negative Moment

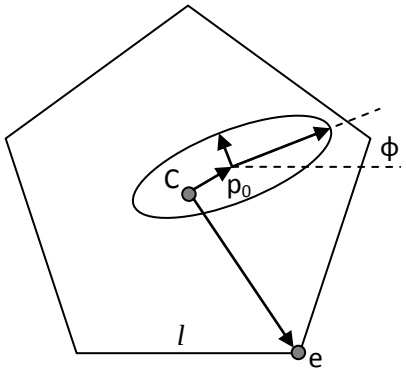


Figure 2: Geometry of the frame

In order to fulfill the first condition, an analysis of the moment about the rolling edge E is conducted. The frame will tip if the moment in Equation 1a is negative. This moment is found by taking the sum of the cross products of the location of the points of application of each force relative to the edge (see Figure 2) and the respective forces applied. Thus Equation 1b. Note that the points of application of each force lie at the centers of mass which exert the forces. Since the spring force is equal to the force that the moving mass applies on the frame, and since that is equal to the product of the mass and the sum of the acceleration and the gravitational acceleration of the moving mass; and since the force due to gravity on the frame is equal to the product of its mass and gravitational acceleration, we may write Equation 1c. Note that in these equations the cross product is used which requires two input vectors in three dimensional space and which outputs one such vector. The third coordinate (in the $\hat{k}$ direction) is zero for all the variables save the moment, for which only this coordinate has a non-zero value. Equations 1b and 1c therefore only refer to this third coordinate.

$$M_E < 0 \tag{1a}$$

$$-\vec{e}\times\vec{F}_G + (\vec{p}-\vec{e})\times\vec{F}_S + \vec{0}\times\vec{F}_E < \vec{0} \tag{1b}$$

$$-\vec{e}\times(m_m\vec{g}) + (\vec{p}-\vec{e})\times\vec{F}_S\left[m_m(\vec{a}+\vec{g})\right] < \vec{0} \tag{1c}$$

In Equation 1c, each *m* represents a mass with the subscripts denoting the frame or moving mass. $\vec{p}$ represents the position of the center of mass of the moving mass. $\vec{e}$ represents the position of the edge about which rolling will occur.

The equation of elliptical motion in the x-y plane defines the position $\vec{p}$ as shown in Equation 2. $\vec{a}$, the acceleration of the moving mass, is then the double time derivative of $\vec{p}$ as shown in Equation 3.

$$\vec{p} = \vec{p}_0 + \begin{bmatrix} r_1\cos(\omega t)\cos(\phi) - r_2\sin(\omega t)\sin(\phi) \\ r_1\cos(\omega t)\sin(\phi) + r_2\sin(\omega t)\cos(\phi) \\ 0 \end{bmatrix} \tag{2}$$

$$\vec{a} = -\omega^2 \cdot \begin{bmatrix} r_1 \cos(\omega t)\cos(\phi) - r_2 \sin(\omega t)\sin(\phi) \\ r_1 \cos(\omega t)\sin(\phi) + r_2 \sin(\omega t)\cos(\phi) \\ 0 \end{bmatrix} \quad (3)$$

In Equations 2 and 3, p_0 is the center of the elliptical motion, ω is the angular frequency of oscillation, φ is the angle of the ellipse, r_1 and r_2 are the radii of the ellipse, and t is time. It may be noted that the moving mass must always be contained inside the inscribed sphere of the frame, and this therefore puts limits on the allowed center position and radii of the elliptical motion. Considering the moving mass as a sphere with radius r_m, the centroid of the moving mass may not leave a sphere with radius r_i - r_m. Furthermore, since the moving mass hangs on springs, it will deform these springs with its weight based on the spring stiffness k. Since the system is designed to oscillate with a certain frequency, and because it is desired that this frequency be the natural frequency, the spring constant is varied based on the desired frequency. Equation 4 shows this relationship. The sag that the moving mass will experience is proportional to its weight and the inverse of this spring constant. $\vec{p}_0$ is defined relative to its maximum value, and thus the relative value varies between -1 and 1. This definition is shown in Equation 5.

The radii of the ellipse are similarly defined. Equation 6 gives the absolute radii of the ellipse in terms of the relative values and the maximum possible value.

$$k = \omega^2 m_m \quad (4)$$

$$p_{0-absolute} = \left[p_{0x}, p_{0y}, 0\right]_{relative} \cdot \left(r_i - r_m - g/\omega^2\right) \quad (5)$$

$$\left[r_1, r_2\right]_{absolute} = \left[r_1, r_2\right]_{relative} \cdot \left(r_i - r_m - \sqrt{{p_{ox}}^2 + {p_{oy}}^2}\right) \quad (6)$$

The location $\vec{e}$ of the edge E is calculated based on the geometry of the frame. Two cases for frame geometry are considered: First the frame is in the form of an equilateral polygonal prism, and second the frame is in the shape of a platonic solid (one of the five regular polyhedrons). The vertical coordinate of $\vec{e}$ is equal to the radius of the inscribed sphere of the platonic solid or the inscribed circle of the polygon forming the shape of the prism. The horizontal coordinate of $\vec{e}$ is the inscribed circle on one of the faces of the platonic solid or the one half the edge length l of the polygon in the prism. Table 1 shows the coordinates of $\vec{e}$ for each case.

Case	**X coordinate of E**	**Y coordinate of E**
Polygonal Prism side length = l, number of faces = n	$\frac{l}{2}$	$\frac{-l}{2\tan(\pi/n)}$
Regular tetrahedron edge = l	$\frac{l}{\sqrt{24}}$	$\frac{-l\sqrt{3}}{6}$

Case	X coordinate of E	Y coordinate of E
Cube edge = l	$\frac{l}{2}$	$\frac{-l}{2}$
Regular octahedron edge = l	$\frac{l}{\sqrt{6}}$	$\frac{-l\sqrt{3}}{6}$
Regular dodecahedron edge = l	$l \cdot \frac{3+\sqrt{5}}{4}\sqrt{\frac{2}{5-\sqrt{5}}}$	$-l \cdot \sqrt{\frac{2}{5-\sqrt{5}}}$
Regular icosahedron edge = l	$l \cdot \frac{3+\sqrt{5}}{\sqrt{48}}$	$\frac{-l\sqrt{3}}{6}$

Table 1: Coordinates of the edge E relative to the centroid of the frame

Note that a spherical frame is not considered. A spherical frame will obviously roll much more easily, but it is also at most marginally stable and only on flat ground. Given a slight push, a sphere will roll away, and it will be very difficult to make it entirely stop rolling. As an alternative, a shape with faces is harder to roll, but in being so is much less susceptible to being pushed and rolling away. Usually such shapes will roll a few times when pushed but will quickly halt on one face. Slight perturbations will have no effect on their position. Thus the trade off is between creating a frame which is hard to get rolling when desired (and which has few faces) and one which is susceptible to being pushed or rolling down a hill (and which has many sides). In this study a compromise is sought between these two.

Condition #2: No Slip Condition

In order to fulfill the second requirement for rolling, namely that the frame will not slip on the ground, the horizontal force applied on the edge E must be opposed by a friction force. The absolute value of such a friction force will not exceed the product of the friction coefficient of the edge against the floor and the force applied perpendicular to the floor. Thus Equation 7a is written. Expanding Equation 5a so that each force is represented by accelerations and masses yields Equation 7b. Note that since the force applied on the floor is always negative when the object is resting on the floor, the sign of this force is changed. Subscripts are used to indicate that only the vector coordinate subscripted is to be considered.

$$\left|F_x\right| < \mu F_y \tag{7a}$$

$$m_m\left|a_x\right| < -\mu \cdot \left[m_m\left(\vec{a}+\vec{g}\right)+m_m\vec{g}\right]_y \tag{7b}$$

Both conditions for rolling can now be evaluated for any given system and time increment. Because of the time variable in Equations 2 and 3, the entire range of times needs to be analyzed that fall within one period of motion. Over this range of times, the time dependant variables of acceleration, force, and moment, among others, will vary greatly.

Condition #3: Rolling Forwards Only

In order to move forwards, the polyhedron will need to roll forward repeatedly and not roll back at any time. In this analysis, rolling is impossible to determine and instead only a moment is possible to

determine. If the moment around the back edge is positive at any point (indicating that the frame will tip onto the back edge) then the difference between the moment around the back edge and the moment around the front edge must be used instead of simply the moment about the front edge. If the maximum moment around the back edge is larger than the maximum moment around the front edge, this allows the system to continuously roll backward in this model. Another option is to make the maximum forward moment zero if there is ever a positive moment around the back edge. These options are discussed in the "Methods of Optimization" section below.

Optimization of the Analytical Model

All of the fundamental variables in the problem are now known, and each is listed in Table 2. Along with each variable listing is a description of how many dimensions it has, and how it is determined. In order to give a range to the scope of the calculations, practical limits for each variable are given which are determined intuitively or by prior work. Typical units are also listed.

Name	Dimensions	Determined by	Practical Limits	Typical units	Ideal value	Importance
$\vec{g}$	3	Earth	$\langle 0,-9.81,0 \rangle$	m/sec^2	N/A	--
l	1	Components	0.2 - 0.6	m	0.4	.2
m_f	1	Components	0.2 - 1.0	kg	1.0	.4
m_m	1	Components	0.5 - 1.5	kg	1.3	.2
n	1+	Components	3-8, or platonic	--	Cube	.4
$\vec{p}_0$	3	Dynamics	$\langle \pm 1,\pm 1,0 \rangle$	--	$\langle 0,0,0 \rangle$	.8
r_1	1	Dynamics	0 - 1	--	0	.2
r_2	1	Dynamics	0 - 1	--	0	.2
r_m	1	Components	0.08-0.16	m	1.6	0.4
t	1	--	$0 - 2\pi/\omega$	sec	N/A	--
μ	1	Components	0.2 - 0.8	--	0.2	.6
φ	1	Dynamics	$0 - \pi$	rad	--	0
ω	1	Dynamics	$2\pi - 20\pi$	rad/sec	4π	1.0

Table 2: Listing of all the fundamental variables in the system

Note that all these variables save $\vec{g}$ and t are available to be changed based on the dynamics and components of the system. Also, each value has appropriate units except $\vec{p}_0$ and r which are ratios. The position of the center of the elliptical and the length of each radius are defined by Equations 4 and 5 respectively.

Methods of Optimization

Because of the number of variables, the complexity of the problem, and the fact that there are three conditions to meet, MATLAB was used to develop programs to determine the best set of parameters. There were two basic programs developed. The first was able to determine for a given set of parameters whether the system will meet the three requirements, and if it does then quantitatively how good it is compared to other systems that also meet the requirements. The second basic program calculated how well the system will perform on a continuous scale which can be maximized to hopefully find the same result as found by using the first program.

Three methods were used for optimization of the output variable of the basic program. First and most simply, hand optimization was used. Second, the basic program was expanded to allow the calculation of multiple values of each parameter at once, allowing ranges of inputs, and producing an output array if which the maximum value could be found. Finally, a genetic algorithm was implemented which used random initial inputs and then maximized an output variable by mixing and varying the inputs.

Hand Iteration

Of these methods, hand iteration was the easiest to implement, but the most difficult to fully optimize. Using plots of the parameters either in time or in space, and using intuition, it was easy to find a system that worked. This process using intuition usually took a couple of minutes to produce a working result. Initially, hand iteration was done using the first basic program which found if and how well the system worked. This program had the disadvantage that it gave no information about how bad the system was when it was not working, and eventually with the creation of the second basic program, hand iteration was more easily accomplished using this second program.

Full Parameter Space Analysis

The second method of optimization involved calculating the result variable for evenly spaced values of each input variable over the entire acceptable parameter space. The output from this was an array of the optimized variable in 12 dimensions, one for each input variable. The best value could be found in this array, and correlated back to the inputs that created it.

There are two ways to compute results for varying input parameters in so many dimensions. The first uses iteration, and although simple, it would have taken approximately 8 hours to run to determine the output variable over a reasonable range of inputs. The second uses arrays in 13 and 14 dimensions, and cannot calculate as many values of each input due to system memory limitations, but it runs in under 30 seconds. However with a proper optimization scheme the entire process may be iterated either manually or automatically to achieve a very specific result in only a few minutes.

One method of automatic iteration uses 4 values for each parameter. Initially, these 4 values are chosen based on the maximum and minimum of each variable and equal spacing in between. After the first iteration, one of the four values will be chosen as the best value to continue optimizing around. If this value is a middle value, then the limits of the new 4 value set will be the two values surrounding the best value. If the best is an edge value, the new 4 value set will be contained between the two values

closest to that edge. In this way the range of each variable is shrunk to either 2/3 or 1/3 its original range in each iteration.

Unfortunately the maximum variable size possible before a memory error occurred was around 16 million elements. Considering that each 12-D array must be used at each timestep, 13 parameters must be varied. The cross product used three connected arrays of 13 dimensions for both input and output, making these inputs and outputs 14 dimensional. Since it is desired to obtain a reasonably accurate result, many timesteps must be used to model the dynamics at many points along the elliptical path of the moving mass. If 9 points through time are used in each of the 3 elements of the inputs and outputs of the cross product, the original 16 million must be divided by 27, leaving 0.6 million elements available for the original 12 dimensions. Unfortunately the twelfth root of 0.6 million is 3.03, indicating that using 4 values for each parameter is not possible. Instead, all but one or two parameters may be varied at a time, then those parameters may be varied in a separate string of calculations.

Genetic Algorithm

In order to get around the problems of conducting a full parameter space analysis, a genetic algorithm was used which optimized a variable using a much smaller set of parameters that were randomly chosen. This algorithm aims to simulate nature's process of natural selection.

In nature, a population of individual organisms exists. Those that are fit to survive do survive, and mate, and a new generation is born from the best of the first generation. Of those fit individuals in the first population, each randomly chose a mate from among that fit group, and each mate passed half of their genes to their child. Furthermore, some of the genes got altered slightly by outside forces, and thus the child is only almost a combination of its parents.

In the computational genetic algorithm used here, an original population is picked such that the traits of individuals are assigned randomly from among all the possibilities. The fitness of each individual is chosen. Those individuals whose fitness is above a threshold percentile go on to become the "fit population". The fit population is then combined with the most fit individuals from past generations, and the most fit half of the resulting population, called the breeder population, is allowed to breed and becomes the new set of most fit individuals for the next generation. A new population is assigned characteristics by randomly picking each characteristic from among the characteristics present in the breeder population. Then all characteristics are randomly varied slightly. This new population then repeats the process. This is shown schematically in Figure 3.

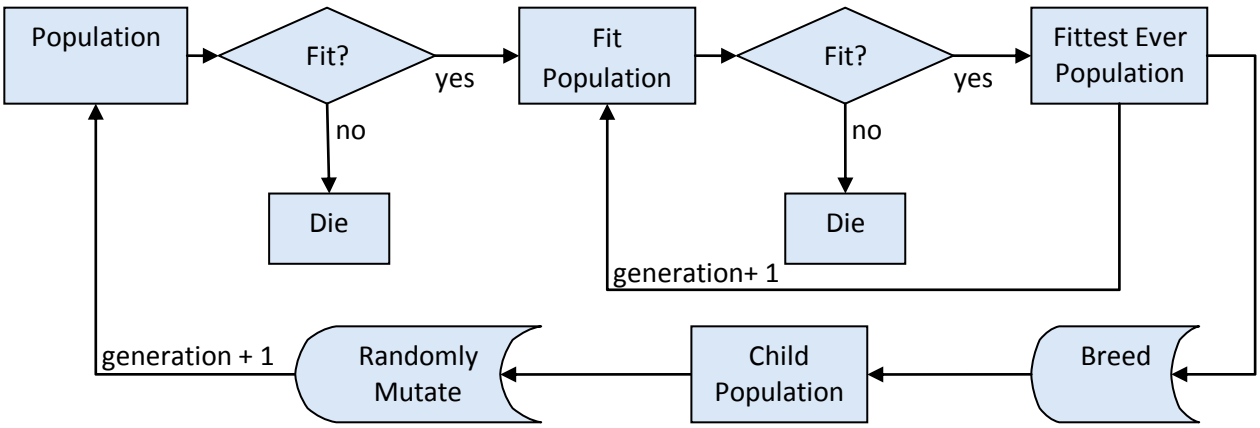


Figure 3: Genetic algorithm process flow

This genetic algorithm is a bit more complicated than most because it has a step that saves and uses older individuals to breed. Over time, this population of older individuals gradually becomes more and more fit and breeds better and better populations.

The Optimized Parameter

The output from any of these analyses has 12 dimensions. Since it is not practical to analyze such data intuitively, an algorithm was developed by which this analysis could be completed. Ideally, this algorithm will choose one value for each of the input variables that determines a system that will work well and also be reasonable to build. In order to find how easy or hard the system will be to build, an ideal value for each variable has been included in Table 2, and an importance has been indicated which denotes how important that ideal value is for that variable compared to the ideal values of the other variables.

To calculate how easy the system will be to build for each result, the distance from the ideal to the actual value of each variable was computed, raised to some power, and multiplied by the importance factor. The powers used in this calculation were chosen using trial and error to effectively counteract the strength of the preferences of the rolling check for parameters far from the ideal value. These 12 numbers added together were called the “idealness” of the system. An equation for this idealness is shown in Equation 8. Because of the absolute value calculation, this idealness will always be greater than or equal to zero. The ideal value of the idealness is zero, with higher being worse. A value of five is on the edge of being acceptable, and lower values are preferred.

$$idealness = \sum importance \cdot \left| \frac{input - input_{ideal}}{size_{nominal}} \right|^{n} \tag{8}$$

This and how well the system rolled were combined into a single optimized variable by one of the two following methods. The first tests if the system will roll and if so how well, and if it rolled this was combined with the idealness using multiplication. This method produces many areas where the optimized parameter is simply zero. This makes using the genetic algorithm difficult, and thus a second continuous function having no zeros was developed.

Yes or No Test

This method of finding the optimized variable is designed to create a sort of phase diagram with three phases. First, if the polyhedron slips, the output variable is -1. If the polyhedron does not tip, the output variable is either 0 or -1 depending if it slips. If the polyhedron does not slip and does tip, then a tipping ratio is calculated as the length of time during which it tries to tip divided by the period of the oscillations. A backwards tipping ratio is also calculated, and difference between these redefines the tipping ratio. Because it can never leave the ground in the current analysis, how far it tips is unable to be answered, but if the tipping ratio is high enough then it is likely it will roll.

This tipping ratio is then multiplied by the difference between five and the idealness for each set of inputs to create the output variable. The idealness is a variable that ranges between zero and a high positive value, but if the system is anywhere near good the idealness will be below five. Therefore, if the polyhedron rolls and is reasonable to build, the output variable will be greater than zero where higher is better.

Unfortunately, for any kind of function which cannot search every value in a large parameter space easily, this test is difficult to use. Because there are so few options that will result in a polyhedron that rolls properly, a vast majority of the elements in the output variable are either zero or negative one. When trying to optimize near such an element it is impossible to find a direction to travel to make that element better. In the genetic algorithm this individual would die. Because an estimated 0.2% of random parameter combinations result in a system which rolls, the initial population size must be extensive. A starting population of 10,000 individuals was used, and this was decreased to 1,000 in subsequent generation.

Continuous Value Test

To overcome the disadvantages of the previous algorithm, a continuous algorithm was developed. An example of the time varying characteristics of the system such as the moment about each edge and the needed friction coefficient are plotted in Figure 4. The system represented in the Figure 4a would work, but that in Figure 4b would not, both because the moment causing it to tip backwards rises above zero (the lower black dotted line) and because the friction needed rises above the green dotted maximum friction line.

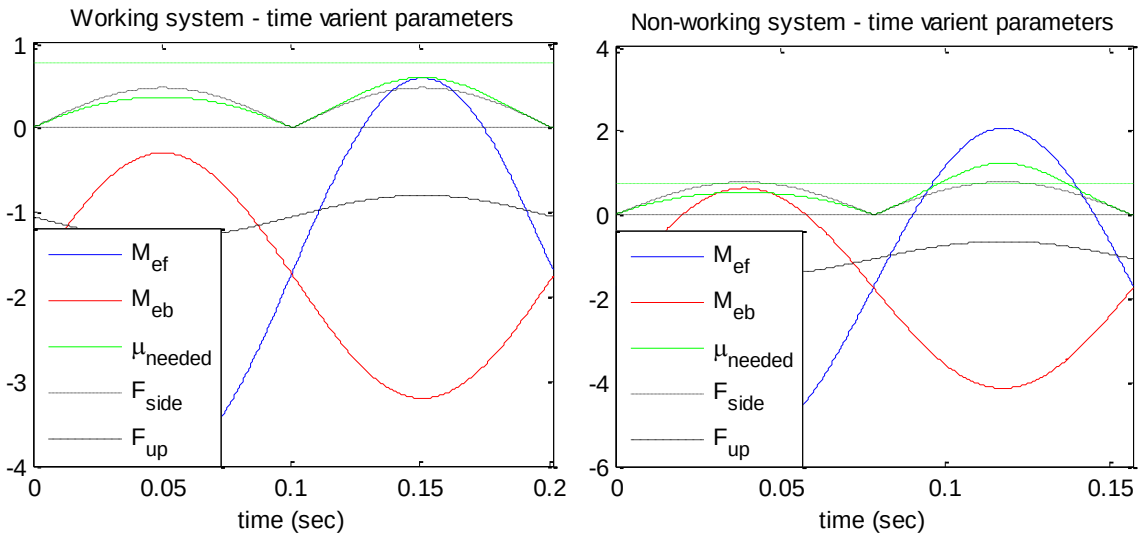


Figure 4: Time variant parameters of two example systems. Only that the left works properly.

In this algorithm, the maximum moments about each edge are calculated, and if the maximum backward moment about the back edge is positive, it is subtracted from the maximum forwards moment about the front edge. This is all divided by the amplitude of the oscillations of the forward moment about the front edge. This is shown in Equation 9.

$$M_{check} = \frac{M_{\max-forward-front} - \frac{1}{2}\left(M_{\max-backward-back} + \left|M_{\max-backward-back}\right|\right)}{Amp_{forwards-front}} \tag{9}$$

The friction check is made continuous by ensuring that the maximum value of friction coefficient that is needed at any point in time is lower than the friction coefficient provided. See Equation 10b. If the polyhedron tries to leave the ground or if the vertical force applied to the ground comes too close to being positive, a negative friction would be necessary, but instead the vertical force is artificially kept negative and made very low so that the necessary friction will be very high. See Equation 10a. The result of this modification of the vertical force is shown in Figure 5.

$$F_{vertical} = \left(F_{vertical} < F_{\max}\right)\cdot F_{vertical} + \left(F_{vertical} > F_{\max}\right)\cdot\left(F_{\max}\cdot\exp\left(\frac{F_{\max} - F_{vertical}}{10}\right)\right) \tag{10a}$$

$$\mu_{check} = \mu_{provided} - \mu_{needed} = \mu_{provided} - \left(\frac{\left|F_{horizontal}\right|}{-F_{vertical}}\right) \tag{10b}$$

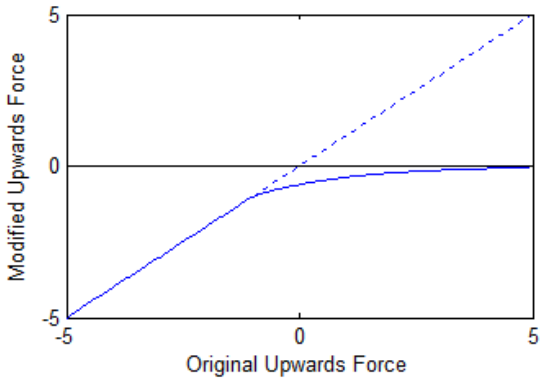


Figure 5: Modification of the upwards force for use in friction calculations

The variable determining if the system will roll is determined by the friction check and the moment check. Its absolute value is the absolute value of the product of these two, but its sign is negative unless both of these values are positive. When both of these values are positive the system will roll properly. If one of them is negative, the equations will still be able to determine how bad the system is, and what needs to happen to improve it will still be obvious.

The output variable of this system is simply a negative multiple of the idealness added to this rolling check. The idealness ranges up from zero, and zero is preferred. Typical values are around 3.5. Typical values of the moment check are around 0.03. In order to make these sizes comparable, the idealness is multiplied by -160. The output variable usually hovers somewhere around zero, with the best systems having a positive output variable.

Results of the Analytical Optimization

The results of the optimization were inconclusive. Figure 6 shows a compilation of many of the tests run with the genetic algorithm. In each plot in Figure 6, the fitness is shown as a function of one of the parameters, or as a function of a few of the parameters combined using an equation. These plots may be compared to the limits listed in Table 2.

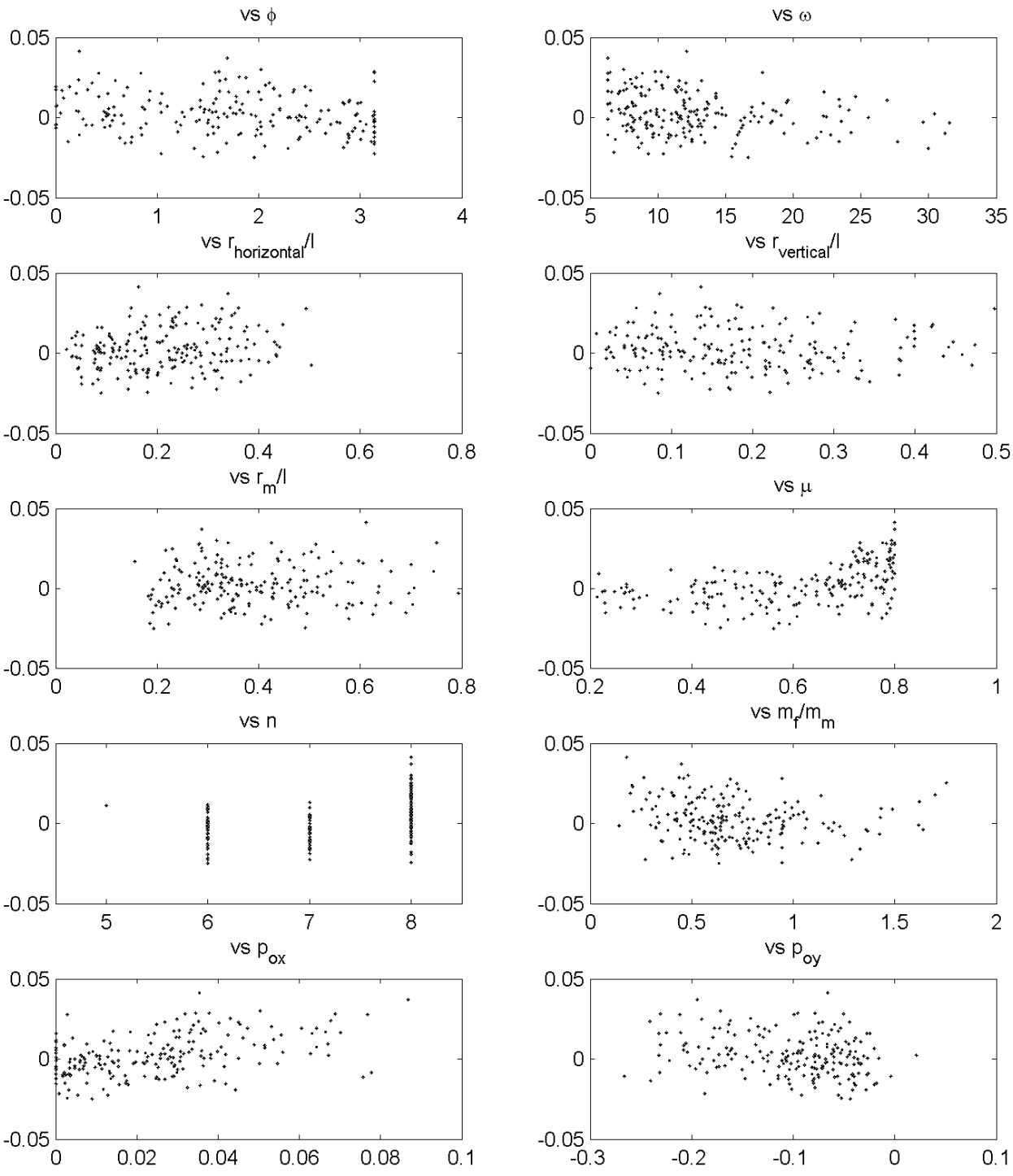


Figure 6: A Compilation of the results of the genetic algorithm

From Figure 6 we may make a few conclusions. As φ is varied, maximum fitnesses are reached around φ values of 0, π/2 and π radians. This indicates that either oscillating directly horizontally or directly vertically, or close to it is ideal. It is difficult to tell if the oscillations are horizontal or vertical from this metric however, because the smaller radius of the ellipse could be either one.

To help determine if horizontal or vertical oscillations are better, and equation was developed for the plot labeled vs. $r_{horizontal}/l$ and vs. $r_{vertical}/l$. The equation for normalized horizontal oscillations is shown in Equation 11. These show that low vertical oscillations were preferred, while slightly higher horizontal oscillations were preferred. This result could also be guessed intuitively: if the oscillations were purely vertical, no moment would be created, and the system would start to hop which would be very bad.

$$r_{horizontal} = \frac{r_1 |\cos\phi| + r_2 |\sin\phi|}{l} \tag{11}$$

In the plot of fitness vs. ω, it may be seen that only low values of ω produced fit results. Obviously values too low would do nothing to make the system roll, and the best results seem to appear around 10 to 15 $^{rad}/_{sec}$.

The normalized radius of the moving mass is a bit harder to determine from its plot. High values of the fitness appear over a wide range of this ratio, but more results appear around a ratio of roughly 0.3. One problem with this result is that as the number of sides varies, this result will vary dramatically. In this representation, there is no control for the number of sides of the polyhedron.

It is obvious from the plot of fitness vs. μ that higher values of the friction coefficient are preferred.

The plot of fitness vs. n demonstrates that fitness increases as the number of sides increases. This was one parameter that was very difficult to optimize correctly, since the algorithms used to calculate tipping gave a strong preference for higher numbers of sides. The only reason that results are shown with lower numbers of sides is the strong preference given in the idealness calculations for a number of sides close to 4. Note that even for five sides, only one result exists, although its overall fitness is relatively high.

From the next plot, we see some preference for lower values of the ratio of the mass of the frame to the mass of the moving mass. The highest fitness appears with a very low ratio, however a higher concentration of results have a slightly higher value of this ratio. This could be in part due to the idealness calculations which act to increase this ratio.

The starting position of the results obtained shows much variation. It is generally accepted from the plots that a negative value for the vertical position is preferred, and a value of between -0.05 and -0.1 is likely best. The plot of fitness vs. horizontal position shows a trend of increasing fitness with increasing values the horizontal position, but the results become more sparse at higher values of the starting position. It is probably best to use a horizontal starting position of around 0.04.

From this compilation of results of the analysis, Table 3 is created which details what is the best range for each parameter.

Name	Practical Limits	Best Range	Typical units
l	0.2 – 0.6	--	*m*
m_f	0.2 – 1.0	$0.3 - 0.8 \cdot m_m$	*kg*
m_m	0.5 – 1.5	--	*kg*
n	3 – 8, or platonic	8	--
$\vec{p}_0$	$\langle \pm 1, \pm 1, 0 \rangle$	$\langle 0.04, {}^{-}0.5 - {}^{-}0.1, 0 \rangle$	--
r_1	0 – 1	0.1 – 0.3	--
r_2	0 – 1	0.05 – 0.2	--
r_m	0.08 – 0.16	$0.3 \cdot l$	*m*
μ	0.2 – 0.8	>0.7	--
φ	$0 - \pi$	${}^{-}0.1 - 0.1$	*rad*
ω	$2\pi - 20\pi$	10 – 15	*rad/sec*

Table 3: Listing of all the fundamental variables in the system

To view the code for these sections, and for more detail with less explanation, the reader is referred to the appendix where various versions of the Matlab code are presented in the Computer Code section.

Computational Simulation and Optimization

In order to validate the analytical work, and in order to keep such a validated model easy to modify, a computational simulation was created. Three parts were used in the simulation: the ground, the frame and the moving mass. The ground was an infinite plane, and the moving mass was modeled by a sphere. The frame took many forms depending on what shape was desired but work focused around an extruded regular octagon and a regular icosahedron. The parameters used in the simulation to control the different behaviors were identical to those described previously, and optimization of these parameters was completed using a genetic algorithm.

There are two significant differences between the capabilities of simulation and those of the analytical model. First, the simulation allows the frame to actually tip over, and second, the moving mass is attached by springs to the frame and thus does not have a rigidly defined location at any time. The impetus for creating a simulation was to test the results of the analytical model with these two constraints relaxed. Also, it would allow a real check if the frame would ever get a change to tip completely over.

This simulation thus gives a result that is hopefully much closer to reality than the result of the analytical calculations. In the planned physical prototype, the moving mass is connected to the frame by attaching servos to the moving mass, winding strings around pulleys attached to those servos, tying the other end

of those strings to springs, and fastening those springs to the corners of the frame. This allows for compliance in the system so that it cannot pull itself apart, and it also creates resonance so that the moving mass may resonate about a fixed point. This resonance is augmented by moving about the position where the springs attach to the frame such that at every time, the springs would rest in a different position. The motion of this rest position must therefore be matched to the frequency of the spring-mass system. This allows the moving mass to oscillate with higher amplitude and speed than the motors in the proposed prototype would allow if the springs were not there.

The Simulation Environment and Setup

This simulation was created in Vortex which is a set of libraries distributed by CMLabs Simulations and which runs in any C++ compiler. Its setup is allegedly similar to that of the slightly better known and open source Open Dynamics Engine (ODE), although its functionality is perhaps slightly more refined and includes more features. Its downside is that documentation of its various functions is all but missing.

In order to start using Vortex, the tutorial files were studied in depth to determine what range of features Vortex includes as well as which features would be valuable for a simulation of a rolling polyhedron. Some of the constraint features available were gravity, hinges, prismatic joints, friction, springs, cables, fluid flow, buoyancy, and of course solid-solid interactions. The solid object types available included rectangular boxes, spheres, cylinders, and convex polyhedrons. There may have been ways to constrain these various objects together, but these methods were not used in the simulation of the rolling polyhedron.

In order to run the simulation, the frame needed to be created, along with the moving mass, ground, springs to hold the moving mass to the frame, friction between the ground and the frame, solid-solid interactions between all relevant objects, and a way to move the moving mass about inside the frame. The simulation was started by copying one of the example files called ExFriction and creating a new directory for it in the same directory as all the other example files to ensure that any relative paths used by the library would still map to the proper location. ExFriction demonstrated the effect of simulated Coulomb friction using the scaled box friction approximation. This approximation creates a force between objects that is only slightly dependant on the speed of travel, and whose magnitude changes only slightly based on the angle of motion. The original file demonstrated how a box could be thrown in a number of directions and it would skid to a halt on the plane. This was modified so that the box was now defined as a convex polyhedron, and the simulation was only started once with zero velocity. The convex polyhedron was defined by a series of points located at its vertices. The moving mass was defined as a sphere and was initialized at a point just below the center of the frame. Because of the collision detection feature of all solid objects, this sphere had no solidity for simulation purposes but instead had a location, a mass, and a physical form for visualization only. In this way collisions between the sphere of the moving mass and the solid convex polyhedron of the frame would not be detected, and the moving mass would be able to exist inside the frame. A rest point was defined relative to the center of the frame as the position the moving mass would occupy if no gravity and no acceleration were applied to it. A spring force was added between the center of the moving mass and this rest point by computing the distance between those two and multiplying that distance by the desired spring

constant. Finally, the location of the rest point of the moving mass was adjusted around the centroid of the frame using the same parameters that were used in the analytical optimization.

Figure 7 is a screen capture of a test using a frame in the shape of an extruded octagon. The moving mass is shown in green near the center of the frame in red. Lines were also drawn behind the objects to track their centroids through space. Above the starting position at left a box was drawn with the dimensions of a meter stick as a scale.

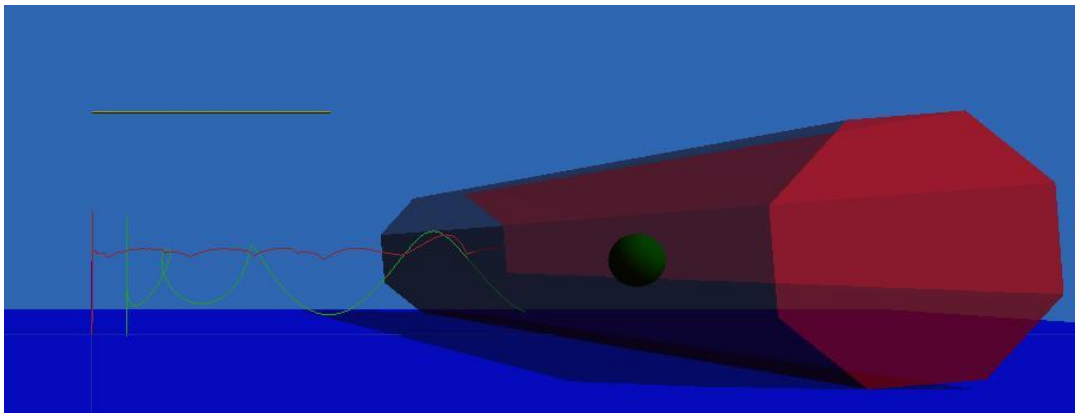


Figure 7: The extruded octahedral frame in simulation

To make all the parameters easy to change, especially for an optimization algorithm, each parameter was defined at the beginning of the script and all the subsequent parameters were calculated. The points that create the shape of the frame for example were created inside a for loop that iterated for the number of required points. The neutral position of the moving mass was changed through time by using Equation 3.

There were some limitations with this model however. First and perhaps most obviously, the frame in Figure 7 is much longer than it would be in real life in order to keep it from rotating in the horizontal plane as it moves. Such rotation could significantly invalidate the results of the simulation. Second, the neutral point of the spring is defined at all times with respect to the centroid of the frame, but this definition has a globally fixed orientation. When or if the frame rolls the neutral position of the spring with respect to the frame will be based on not only time but also on the orientation of the frame. This creates a need for orientation feedback when creating a real model.

A slightly more complicated issue is that the spring as modeled is nothing more than a restoring force directly pointed at the rest position. In reality, a spring will stretch from each vertex and will terminate on the surface of the sphere represented in green in Figure 7, and this spring will have a rest length and an initial tension. Because the moving mass has a non-zero diameter, the actual restoring force created by the combination of all the springs will be slightly different from that of the single zero length spring connecting the centers of the objects. Figure 8 represents the forces the moving mass feels at a number of locations when four springs act on it. Note that the restoring force is usually only close to pointing at the center of the four springs. Those positions that would cause a collision are not plotted.

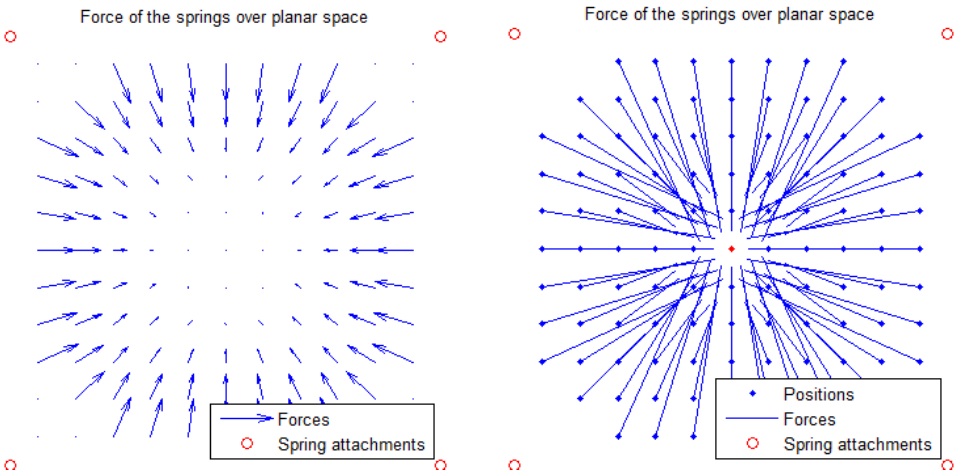


Figure 8: Two representations of the restoring spring force for four spring locations

This model proved to be nominally easy to use. Changing parameters was easy, and it was easy to see how the model reacted to the various changes. Furthermore, it was possible through the Vortex visualization interface to manually add a spring to some point on an object by clicking with the mouse. This enabled more intuitive and easy control.

The simulation of the system that had been optimized using the analytical approach in Matlab showed that the system did not roll. Once the amplitude of the vibration was enough to cause the system to move, it hopped in the air and landed on the same face it had been on before. This hopping turned out to be very easy to achieve, and even resulted in forward motion, but the goal of this present research was to create a system that would roll. By decreasing the amplitude of the oscillations, the system stopped hopping, but also stopped doing anything else, and making the original system oscillate in a more horizontal direction just resulted in sliding. By manually changing the parameters, only hopping, sliding, and no motion were possible.

Genetic Optimization

It was the hope that by implementing a genetic algorithm on the simulation that an individual would develop that was capable of rolling. As stated in the previous section on genetic algorithms to optimize the analytical model's parameters, genetic algorithms are able to optimize parameters for a given problem very effectively if given enough time to run. In this case, when the visualizer was turned off, fifteen seconds of visualization time corresponded roughly to one second of real time, and with a small population size of 100 individuals, several generations could be computed per minute.

The implementation of the genetic algorithm was done with a set of open source libraries called Fast Genetic Algorithm that was developed by Alessandro Presta and which is available at fga.sourceforge.net. This implementation requires the user to write a fitness function, a mutation function, and a random gene generator function, and a main function which calls the first three. Code was copied from the example in the documentation. In this approach, the random gene and the mutation functions needed to be written from scratch, while the fitness function was simply the simulation with some extra code at the end to give a numerical value to how well the system rolled.

The input parameters to the fitness function were a set of values between a minimum and a maximum value. These minimum and maximum values were mapped to the range from zero to one such that twelve numbers in that range could specify the system completely. The same twelve parameters were used in this section as in the analytical section, and these are listed in Table 3. Also optimized was the stiffness of the spring, making thirteen parameters total. Nine of these were optimized. In this way, the gene that was given to the fitness function to describe the parameters could be generated by simply generating a set of random numbers between zero and one, and the mutation function could alter this value in a well defined way that was the same for each parameter. The fitness function then mapped each of these values back to the real values of the parameters and computed the fitness.

The fitness was determined by running a simulation of the system and determining how well the system rolled, or more likely how close the system was to rolling. In order for genetic algorithms to work properly, the optimized variable must have a smooth increase over the parameter space toward the optimum value. Many tests that could be performed are Boolean in nature, but this would cause the function to be impossible to optimize using this approach. Instead, three parameters were examined. First, how far did the frame travel? This was investigated in order to maximize the distance it moved. Second, how high did the frame go? If the maximum height of the centroid of the frame got to a height that was greater than it would be at its tipping point, this was penalized. Similarly, if the frame did not get to this height, it was also penalized. Lastly, was the frame rolling? The distance it traveled was compared to the number of revolutions it spun to find if the numbers corresponded. Any excess distance that was not accounted for by rotation was penalized. Equations 12-12c show the equations that were used to calculate the fitness.

$$fitness = fit_1 \cdot 0.2 + fit_2 \cdot 0.2 + fit_3 \cdot 0.6 \tag{12}$$

$$fit_1 = \frac{p_{frame_x}}{2 \cdot \pi \cdot l \cdot n} \tag{12a}$$

$$fit_2 = \frac{-\left|\max\left(p_{frame_y}\right) - r_{outer}\right|}{r_{outer} - r_{inner}} \tag{12b}$$

$$fit_3 = -\left|\frac{p_{frame_x}}{l \cdot n} - \theta_y\right| \tag{12c}$$

In equation 12a, the position of the center of the frame is measured after the simulation is completed, and the x coordinate $p_{frame\text{-}x}$ is normalized to the size of the frame with size length l and number of sides n. This equation is increased as the frame moves farther in the x direction. Equation 12b ensures that the frame does not leave the ground and travel higher than a small bounce. The maximum of the height of the center of the frame $p_{frame\text{-}y}$ is measured and is compared to the radius r_{outer} of the circumscribed circle or sphere. The absolute value of this difference is then compared using division to the expected height it will rise. Equation 12c estimates the change in angle that is expected given the travel of the frame in the x direction. This is compared to the change in angle that occurred and the negative of the

absolute value is taken to ensure any discrepancy has a negative effect. Finally, these pieces are combined together with summation in Equation 12 where weights are added. Notice that by far the most important parameter was the third that checked for rolling.

Optimization was run several times for between a few minutes and overnight. Unfortunately, the maximum fitness that was obtained was -0.0009, while most of the fitness values ranged from -0.1 to -5. Table 4 shows the parameters that made the fittest system, where the relative terms were relative to the maximum possible value before collision. Notice that only nine parameters are listed. It was determined that the ratio of m_m to m_f was important, but the individual values were not. In this vein, m_f was set equal to 1. μ was also set manually to 0.5, while r_m was set at the minimum radius that could be obtained when designing a system in CAD. The number of sides n was set at 8 to form an extruded octagon as appears in Figure 7. Such a system would have a chance to roll but would not be able to be pushed over too easily otherwise. Figure 9 shows the maximum fitness values that were found during each generation. The absolute maximum fitness was found several times and each time it is indicated with a green line.

Parameter	$p_{ox}Rel$	$p_{oy}Rel$	r_1Rel	r_2Rel	φ	ω	l	m_m	*stiffness*
Gene	0.52907	0.93011	0.28386	0.88972	0.07475	0.3932	0.30971	0.45771	0.4859
Real Value	0.0174	0.2582	0.9516	0.3118	-0.2178	13.18	0.2619	1.349	122.9

Table 4: Parameters of the fittest system (SI units or relative to maximum)

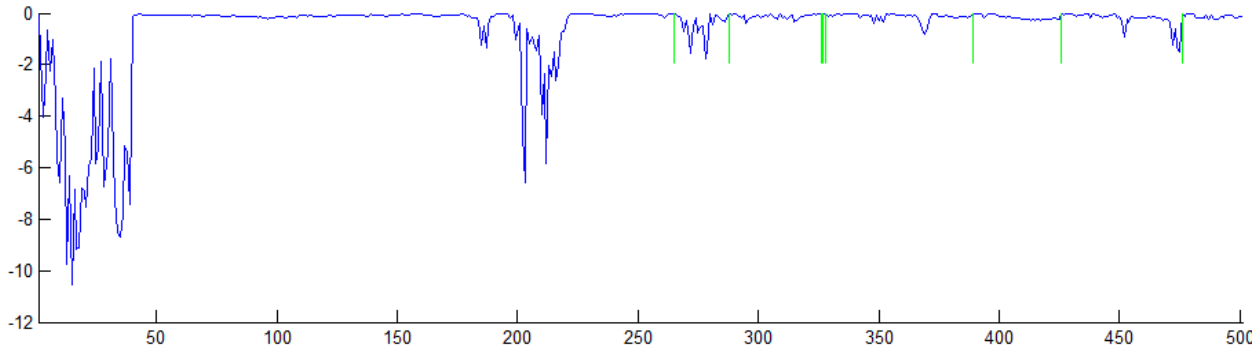


Figure 9: Maximum fitness values found during each generation, with absolute maxima indicated in green

Because the fitness was still below zero, either some of the parameters were not correct, the solution was still improving, or it was impossible to make the system roll. Unfortunately, the maximum fitness of the population did not increase dramatically over time, indicating that there was an issue with the simulation or the system. Upon investigation, the rolling of the frame produced values for the angle θ_y in equation 12c which were remarkably far from the truth. Such estimates of the angle could be twice the actual angle, and this led to extreme values of fit_3. Needless to say, this optimization did not find parameters that were suitable for the system to roll.

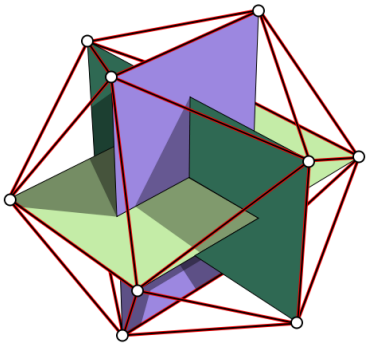
After the failure with the genetic algorithm, two options remained for making a working simulation. First, the fitness function, especially that which appears in Equation 12c, needed improvement. Unfortunately the inaccurate angle θ_y that was input into Equation 12c was determined by a function

from the Vortex library which calculated the Euler angles of an object. Because of this method of determining the angle, it was difficult to update this function to make a more accurate estimate, and therefore it would have been difficult to greatly improve the fitness function.

As an alternative, the parameters which the fitness function found were plugged back into a version which would visualize the rolling behavior, and hand iteration was used to further improve the system. Immediately obvious was that the friction coefficient was too low to allow rolling, and that the value of 0.5 which had been set was too low. Such a low friction coefficient is desirable so that the frame may roll on many types of surfaces, but it is certainly possible in reality to increase this value significantly. When the friction coefficient was increased to 0.7, the system rolled successfully without further ado. A series of figures showing the system rolling appears in the appendix. C++ code for the optimization also appears in the appendix in the computer code section.

The Icosahedron

In this optimization work, an extruded octagon was always used. In reality it would be nice to be able to create a three dimensional object which would be able to roll in more than one direction. For this purpose, the frame was changed to be an icosahedron (the twenty sided platonic solid, roughly like a soccer ball). This icosahedron was constructed using the same convex polygon method as before where each point was defined manually. The points were defined to be at the corners of orthogonal rectangles whose side lengths were the in golden ratio. This is shown in Figure 10, taken from Wikipedia.



$$golden\ \ ratio = \frac{1+\sqrt{5}}{2} \approx 1.618$$

Figure 10: Construction of an icosahedron using rectangles with side lengths in the golden ratio

Because the fitness function in the genetic optimization algorithm was not working correctly, genetically optimizing the parameters using the icosahedron was not done. Instead, hand optimization was used, and produced no working result. Because of time constraints, simulation was abandoned at this point in favor of building a prototype.

Building the Prototype

Finally, and possibly most importantly, the results of the optimization work were tested in reality with a physical prototype. This prototype followed the design of the prior work by Jonas Neubert where a similar structure in the form of a regular tetrahedron was able to hop along the floor. The frame was to be created with a number of rods connected at the corners, and the moving mass was to be connected

to the frame with strings. Springs were placed in the design in series with each string such that resonance was possible. The springs connected directly to the frame while the strings wound around pulleys that were attached to servos that were mounted on the moving mass. In this way the servos could wind in a given direction on one side and in the opposite direction on the other side, and the moving mass would be translated to the side where the strings were wound farther around the pulleys. The springs allowed some compliance in the system such that winding a given string too far would simply stretch the spring and not pull the frame apart. They also provided resonance such that when the frequency of oscillation of the servo position was right the entire moving mass would translate much further than would be possible with the servos and string alone. Figure 11 shows a schematic of this design.

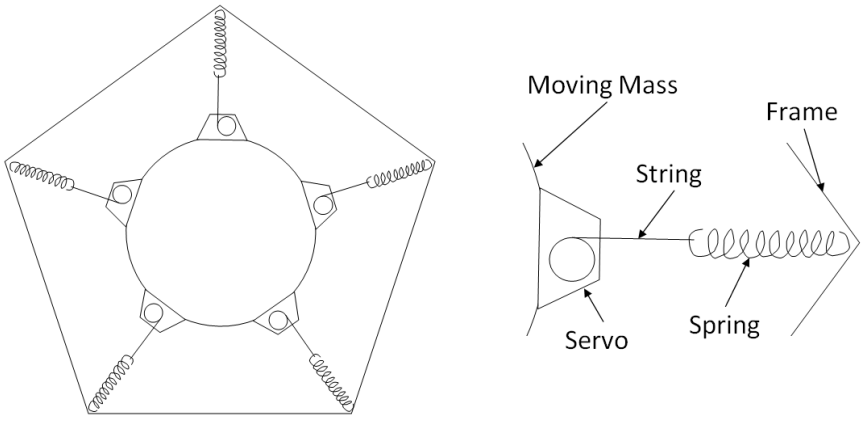


Figure 11: Schematic of the overall design of the system

Designs

In order to allow flexibility and to investigate different options, two designs were created. The first used an extruded octagon as the frame while the second used an icosahedron. The extruded octagon used four servos placed in a square configuration about the moving mass, and these connected to every second edge of the frame. The icosahedron frame used twelve servos placed on the faces of a dodecahedron in such a way that each face of the dodecahedron corresponded to a vertex of the icosahedron

The frame was in each case to be made out of carbon fiber rods with corner brackets to be made using the rapid prototyper. The corners of the frame needed to be able to withstand some bending moment without breaking, and they needed to be able to firmly hold the frame rigid. Also, it was desired to increase the friction of the corners against the ground as much as possible, and small spikes were added around the corners for this purpose. The spring was to attach to the center of one of the edge sections of the octahedral frame or to the vertices of the icosahedral frame. The attachment points for the icosahedral frame are small eyes that were added to the inside of each bracket. Because the corners could be made using the rapid prototyper, they could be in almost any arbitrary shape without added cost. Below, Figures 12a and 12b show the frames as they were designed, each with an inset detail of the corner.

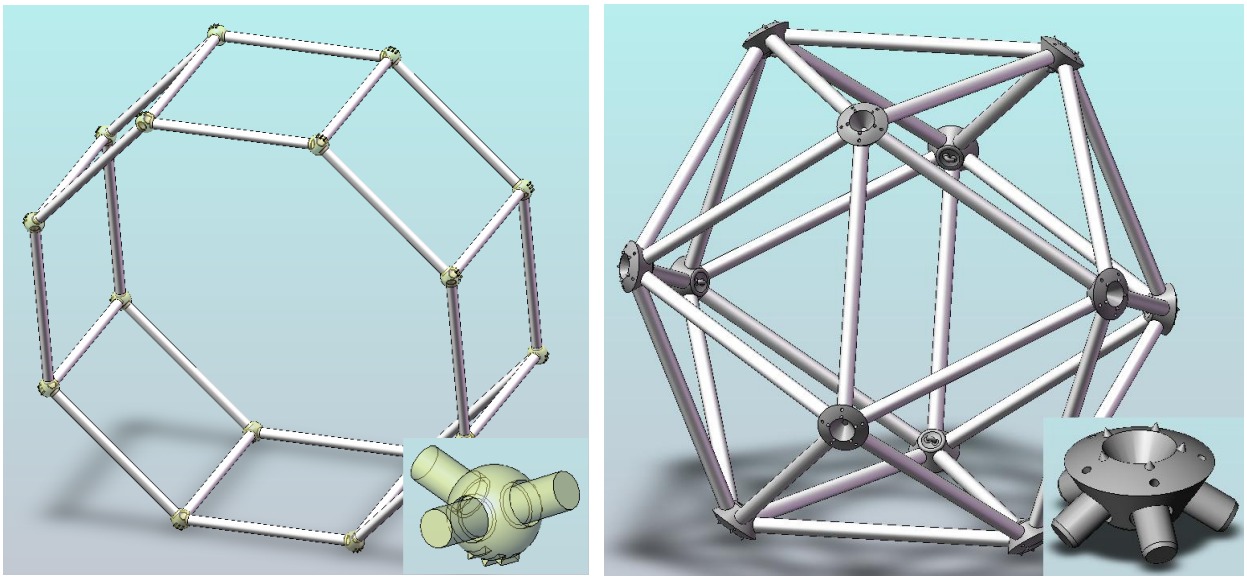


Figure 12: The two frames designed with insets of their corner brackets

The moving mass needed to mount the servos, house the controller for the servos, and house the batteries to power the system. In each case, the moving mass had to mount servos in such a way as to ensure that each string was pointing directly at an attachment point. The servo used was the Dynamixel AX-12+ available from Trossen Robotics (trossenrobotics.com, P/N: FRS-B-AX-12). This servo had been used my many others in the Cornel Computational Synthesis Laboratory, and it was known to work well. The horn included with this servo has four attachment points where a pulley could be mounted. A pulley was designed so that it would just clear whatever surface the servo was mounted on while the servo was mounted on its side. This configuration caused the string to leave contact with the pulley at the shortest possible distance from the center of the moving mass, thus creating the smallest effective radius of the moving mass as described in the analysis section.

The controller for these servos, also available through Trossen Robotics is the Robotis CM-5 microcontroller. It uses software that will be described in more detail later, but is relatively easy to learn and provides low level functionality. Programming in C was also possible, but is not necessary to use this controller. It comes mounted in a case which, when taken apart, is fairly flat except for two large capacitors which protrude from the underside.

Batteries were selected based on the maximum voltage rating of the servos and controller, the amount of current they could provide the system, their weight, and their dimensions which needed to fit inside the moving mass. Because of compatibility issues and the ease of finding appropriate batteries with so many options, batteries were purchased from Trossen Robotics who also provided the controller and servos. Each servo has a maximum current rating of 900mA, and a maximum voltage rating of 12V. The battery selected was the 3 cell 2000mAh 11.1V Li-Po battery from Thunder Power with a maximum current rating of 60A. This battery was also light weight, and it would provide just over three minutes of run time with four servos attached at maximum load. With less load comes significantly improved run time. With this servo and battery in mind, a design was created for the moving mass.

The moving mass for the octagon frame is a simple design where the battery and the controller fit into slots on the inside while four servos mount to the outside. Because of its simplicity, an extruded square was used for the shape of the moving mass for the octagon frame. One servo mounts on each side of the square, each with the location of the tangent to the string in the exact middle of the face, allowing each string to radiate from the centroid of the moving mass. Two of the servos on opposite sides were mounted so that the body of the servo was closer to one end, while the other two had their bodies closer to the opposite end so that the weight was balanced about the centroid. The battery and controller were relatively flat and were to be mounted in the center of the square with access at one end. Since only four motors are used in this design, one battery is enough to provide adequate current and will last for several minutes of use. The battery and controller were held in place by slots into which they fit, and these were sealed by a plate across one end with screws. An on/off switch was provided to be able to quickly provide power to the controller and start the motion of the servos. A picture of this design is shown in Figure 13. In the battery and in the servos, holes were cut out to reduce the weight for mass analysis purposes.

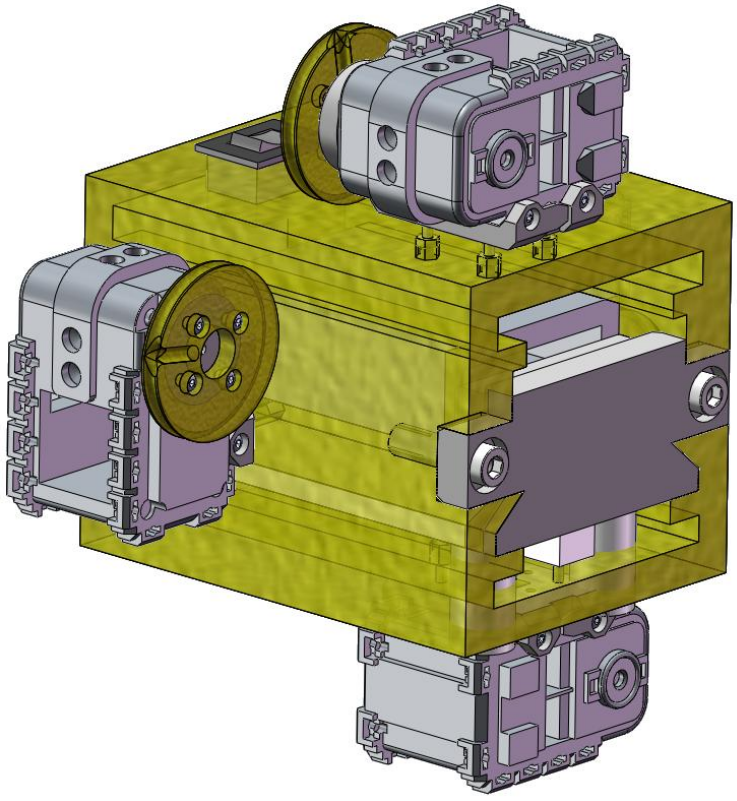


Figure 13: The design for the moving mass for the octagonal frame

Because of the limits the octagonal frame imposed on the direction of rolling, this design was discarded in favor of the more fully three dimensional icosahedron. The icosahedron has three edges which it may roll about at any given time because its faces are triangles. This allows it to travel in a much more versatile way, and it may reach any point on the surface where it lies, and not just along the line in which it is pointed as is the case for the octagonal version. Note also that although this present work

focuses only on rolling, hopping may also be used to further and more accurately position the robot in the plane. If the frame were a sphere, hopping would be much more difficult and may not work (see Condition #1 in the Analysis section).

The icosahedron has twenty faces, twelve vertices, and thirty edges. In order to achieve optimum control, and in order to avoid a strange spatial distribution of spring constants (see Figure 8), one servo was to be attached to each vertex of the frame. These servos may be mounted in a plane that is perpendicular to the string and at a certain distance from the vertex. These planes enclose a volume which turns out to be a regular dodecahedron (the platonic solid having twelve pentagonal faces). The design of a moving mass for this configuration places the servos on the outside of such a shape and the batteries and controller on the inside.

Initially, the servos were mounted on the faces and were connected to the strings in much the same way that they are in the design of the octagonal frame's moving mass. However because each servo must be mounted beside the center of the face in order that the string may leave the pulley at a position above the center of the face, each face has much wasted area. To avoid this problem, a small bracket was designed to fit over the servo and re-route the string so that it would start its path toward the vertex of the frame after leaving a point roughly in the middle of the servo. In this way, the servo could be mounted in the center of the face, and the face only needed to be large enough to hold the servo. This bracket, because it was a strange shape, was printed using the rapid prototyper. The rapid prototyper, the Objet Eden 260V, uses a material which is translucent but not quite clear, thus the clarity of many of the CAD models in the figures here. Because it will have a string constantly rubbing over it, metal pins were inserted to act as pulleys and to eliminate the chance that the string would wear through such features made of plastic. Figure 14 shows the servo and all of its associated components including this bracket used to re-route the string.

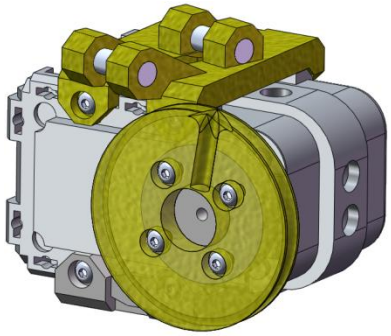


Figure 14: The servo and all connected components

In order to place objects inside the dodecahedron that makes up the center, this dodecahedron was split in half along ten of its edges. The inside was hollowed out and the wall thickness was set at 5mm which would give sufficient support but not be too heavy and provide enough room inside for the various components. The two halves are screwed together, but because of wear issues, threaded inserts were used in the receiving end while flat washers were used under the screw heads. The half that receives the screws, nominally herein called the bottom, is shown below in Figure 15. This part also,

because of its strange shape, is designed to be printed using the rapid prototyper, althuogh this will be made with Stratasys Dimension SST which prints in ABS plastic. The servo itself comes with a small bracket to which it mounts, and this bracket has eight mounting holes in a square to attach to another part. Four holes were positioned in each face of the dodecahedron to correspond to the mounting holes in this bracket. At the inward facing end of those holes, a hexagonal depression was added to catch the nuts that would be added. Finally, because the shape is totally enclosed, holes were cut through which the wires would be fed. These holes were large enough for the 10x4 mm connectors on the ends of the wires.

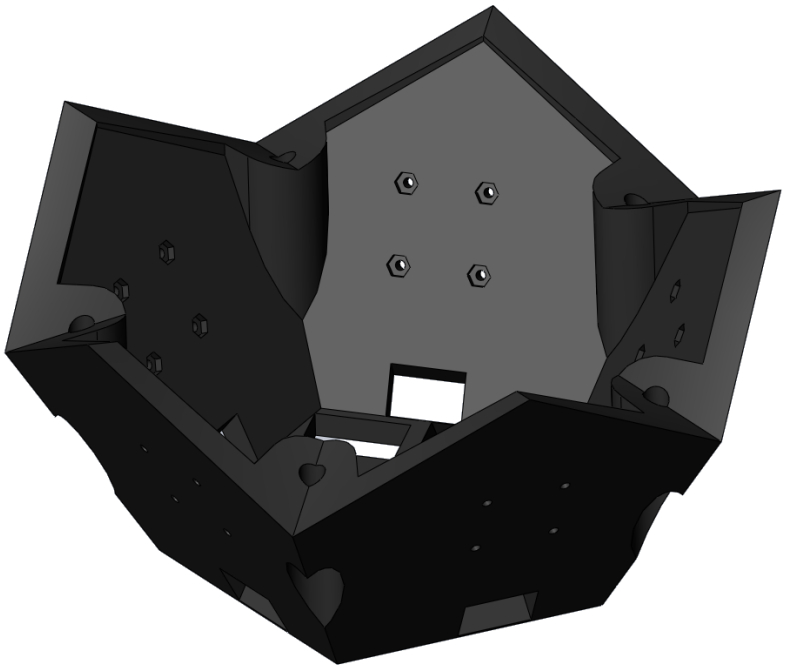


Figure 15: The "bottom" of the dodecahedral shell for the moving mass

The two batteries that the system would use and the controller have no trouble fitting into the open space inside the dodecahedron, and these would be cushioned with bubble-wrap or similar padding. The two batteries are expected to provide several minutes of use, especially considering that the servos on the sides of the dodecahedron see little motion and thus require little power.

The string used to connect to the pulleys was non-descript woven nylon string, and the springs were chosen based on the ideal natural frequency from the simulation and on the mass of the moving mass using Equation 4. The string was attached to the pulley by passing it radially through from the middle of the pulley to the outer edge through a small hole, and tying an extra washer to the end if the string at the middle of the pulley so the end would not come through the hole. The string, after going through the re-route bracket, was tied to a small key ring style wire ring. Another similar ring was attached to the eye in each corner bracket of the frame. Springs were attached between these two rings to connect the moving mass to the frame. Figure 16 shows the completed system. Note that the shell of the moving mass in this system was printed with black ABS plastic, while all other printed components use the yellow material from the Objet printer.

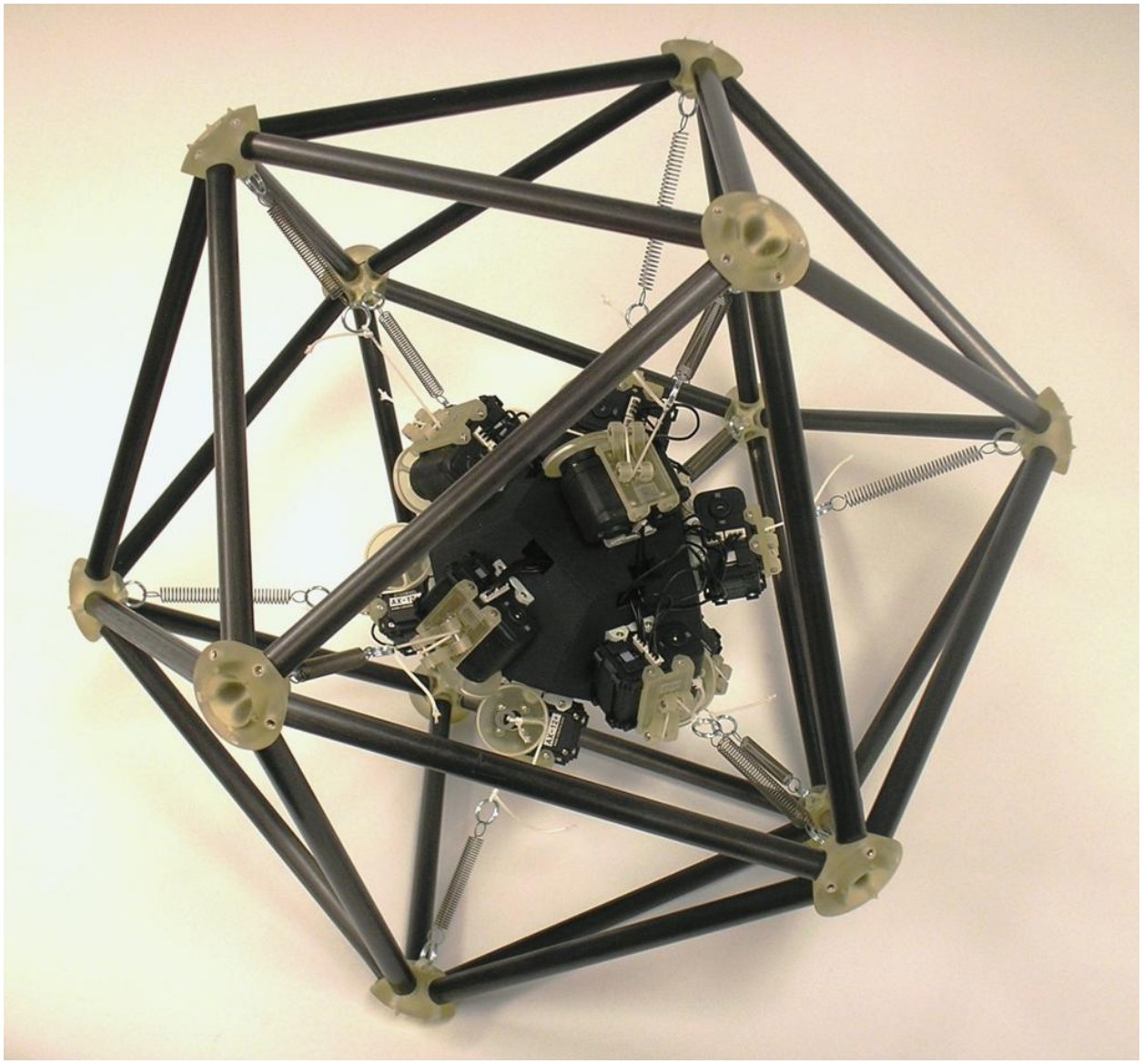


Figure 16: The finished system

Control

Control of the system was implemented through the software that is available from Trossen Robotics. Of the three sets of software available, the Motion Editor was used. The Motion Editor software simply reads and writes to the memory of the CM-5 controller. Setting the position of the servos is done my manually entering numerical positions (0-1023 corresponding to 0-300 degrees), and saving the configuration to a “pose.” Seven of these “poses” may be stored per “page,” and a page may be played multiple times. The page also specifies the speed of the servos, and how long they should wait before executing the next pose. These pages were manually played in testing, but they may be programmed to play in a sequence by using a separate set of included software called the Behavior Control Programmer which allows some very low level logic. Control using C is also possible using the Robot Terminal, but this

requires much more work to program properly than the Motion Editor. For this reason, the Motion Editor was used.

The position was specified at each timestep by a number from 0 to 1023. An algorithm was written in Matlab to transform the motion parameters from the simulation to the numerical control for the servos. This was done by finding the position of each of the twelve locations where the string left contact with the moving mass, and comparing it to the position of the vertices of the frame. The difference between these two positions was computed, and a length was extracted. The minimum length of all of the strings over all times was subtracted from all the strings at every time, so that at least one of the strings always hit zero length but never lower. These lengths were then scaled to coincide with the numerical position of each servo, and were subtracted from 1023. Thus the number 1023 represented the fully contracted string while 0 represented the fully extended string. Equation 13 finds the position of each servo.

$$Position_i(t) = 1023 - \frac{1023}{0.09m}\left[\sqrt{\left(x_{f_i} - x_{m_i}(t)\right)^2 + \left(y_{f_i} - y_{m_i}(t)\right)^2 + \left(z_{f_i} - z_{m_i}(t)\right)^2} - minimum\right] \quad (13)$$

The positions x, y, z of the frame (subscript f) and the moving mass (subscript m) over time were determined by the positions of the vertices with respect to the center of the regular sized frame or a reduced version of the same. This reduced version corresponded to the positions of the exit locations of the strings from the moving mass. In the case of the moving mass, these values were added to the position of the center of the moving mass as determined in Equation 2.

Since these positions were to be entered manually, it was desirable to use as few timesteps as possible for the 12 servos. Two timesteps were chosen, and the motion proceeded to oscillate between these two in a straight line.

Results

Unfortunately, the springs first selected were just barely enough to hold the moving mass off the bottom of the frame when the motors were fully retracted. Although the natural frequency of the system was likely very ideal in this circumstance, there was no way that the system was going to work. One of the springs was even permanently stretched when someone touched it. The motion that was programmed hardly did anything to alter the position of the moving mass, and usually it just sat at the bottom of the frame. A picture in the appendix that shows this behavior.

To remedy this, a range of new springs was ordered. The springs that were used were 0.7 lbs/in while the old ones were 0.12 lbs/in. This increase dramatically increased the natural frequency of the system, and significantly elevated the moving mass off the floor of the frame. In this configuration, oscillation was readily apparent, and when the frequency of oscillation matched the natural frequency of the system, resonance was very apparent as well.

When the program was set to oscillate the moving mass up and down vertically, the frame was able to hop off the ground about one centimeter, and when slightly tipped, this was converted to forward motion at a speed of about 0.8 meters per minute. This result repeats the results of Neubert but with

less optimized travel rate and thus slower motion. The system was able to bounce slightly higher however than in his work.

Rolling was not demonstrated, and although tipping started, it was not completed. One corner of the frame lifted about one centimeter off the ground while the other two stayed fixed, but more was not achieved. It was obvious during this last attempt that the system needed to build up resonance in order to accomplish even this partial tipping. This partial tipping came only every three cycles approximately, and would cause the system to lose all of its built up potential and kinetic energy.

Unfortunately the motion of the servos was restricted by their speed given this higher natural frequency. Because we are operating in most cases at the limit of speed for these motors, more amplitude was not possible, and in some cases it was diminished from the ideal case. Because new servos with higher speed capabilities would be expensive, and because the torque of the present servos is not an issue, increasing the diameter of the pulley would be advantageous. This would however require a re-design of some of the components.

Future Work

Future work on this project will consist of making the current model actually roll. There are a few elements that may be explored in this respect. First, the turning amplitude of the servos is restricted because of their speed. Second the distance the string may be pulled is restricted because of the diameter of the pulleys and also because of the dead zone in the servos. Third, the present work focuses on causing rolling to occur by using resonance. It may be very possible to simply use gravity for the same effect if the moving mass can be translated far enough to one side of the frame.

In order to increase the diameter of the pulleys which seems like the simplest next step, a redesign of the mounting positions of the servos would be necessary. If the servos were mounted on end, the diameter of the pulley could be increased slightly, but if the servo could be placed on its back (so that the axis of the pulley pointed at the corner of the frame) then the diameter of the pulley could be greatly increased. Such a design change would require a redesign of the mounting bracket for the servo (the present version comes packaged in the box with the servo), and also it would require a redesign of the routing bracket.

If such a design were implemented, new springs would also be necessary. Because the position where the string leaves contact with the servo and related components would likely be closer to the corner of the frame, the spring would need to be shorter with the same stiffness and travel, or disappear altogether. Such a spring with shorter length and similar stiffness could take many forms, and a simple one is sketched in Figure 17.

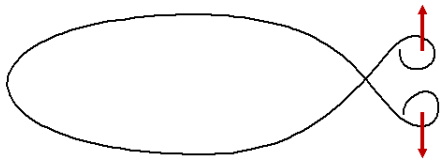


Figure 17: Spring with long extension and short rest length

Once the system is made to roll over one edge, controlling a continuous roll will be the next task. In order that the system may go in a desired direction, some position and orientation sensors will be necessary. In order to determine which face the icosahedron is currently sitting on, pressure sensors on all the faces or accelerometers placed internally may be used. In this way the controller would know how to update the control such that the appropriate action is taken for whichever face the robot is sitting on and whichever direction the robot wants to go, and even to enable it to decide which mode of motion (hopping or rolling) is more advantageous at any time. To determine its location in space, some external means will be necessary, or else the robot will need to have a sensor which can interact with the surroundings (such as a camera, sonar, etc). To determine the orientation in space, a compass may be used for absolute measurements, or some dead reckoning using accelerometers may be used as a starting point for a very rough approximation.

References

Lipson, Hod – Local Search Algorithm– CS 4700

Neubert, Jonas – Dynamical Analysis of Tetrabot – MAE 5170

Neubert, Jonas, et all – Exploiting Resonant Dynamics for robot locomotion

Craig, John J., Introduction to robotics, 3rd ed., Prentice Hall, 2004

“Platonic Solid” – Wikipedia. <http://en.wikipedia.org/wiki/Platonic_solid>

“Regular Polygon” – Wikipedia. <http://en.wikipedia.org/wiki/Regular_polygon>

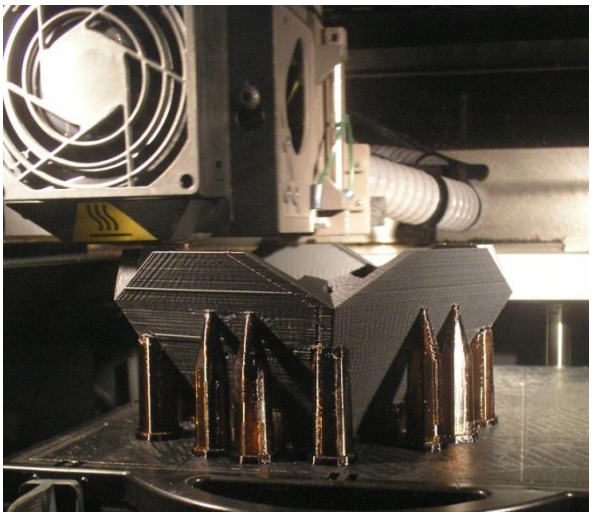
“Icosahedron” – Wikipedia. <http://en.wikipedia.org/wiki/Icosahedron>

Presta, Alessandro – FGA – Fast Genetic Algorithm. <http://fga.sourceforge.net/>

Appendix

Pictures

Figure 18: The sagging moving mass inside the frame with weak springs



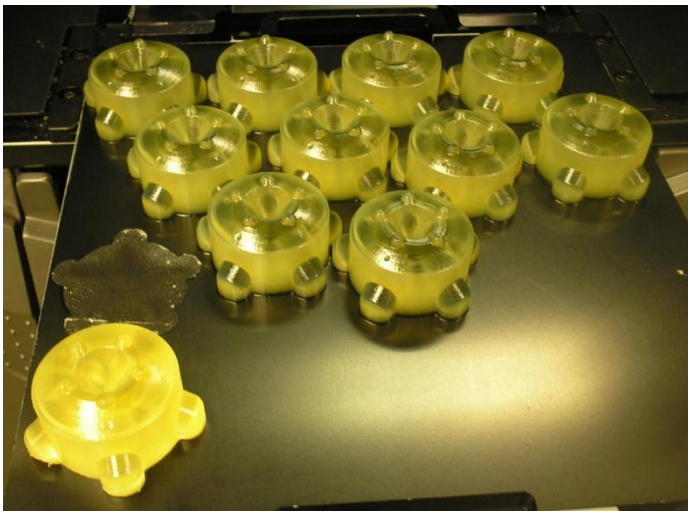


Figure 19: Some of the parts as they were printed, before support material was removed.

Parts List

Center Dodecahedron					$865.39			
PART NUMBER	**QTY.**	**price**	**qty in pack**	**packs needed**	**extended price**	**supplier**	**ordered on**	**Part number**
dodecahedron half, default	1			1	$0.00	printed		
Dynamixel AX-12+ Servo	12	$44.90		12	$538.80	Trossen Robotics	3/16/2010	FRS-B-AX-12
servo horn	12			12	$0.00	--		
Dynamixel AX-12+ Servo mount	12			12	$0.00	--		
string reroute	12			12	$0.00	printed		
pulley	12			12	$0.00	printed		
4 x 15 mm pin - string reroute	24	$7.84	25	1	$7.84	McMaster	3/16/2010	91585A086
screws mount servo to bracket	144			144	$0.00	--		
Threaded insert 10-24 attach halves	5	$8.37	25	1	$8.37	McMaster	3/16/2010	93415A054
nuts to mount servos	48	$4.45	100	1	$4.45	McMaster	3/16/2010	91828A111
screws to mount servos	48	$6.37	10	5	$31.85	McMaster	3/16/2010	91239A707
on/off switch	1	$4.50		1	$4.50	McMaster	3/16/2010	7395K28
dodecahedron half, top	1			1	$0.00	printed		
CM-5 Control Module	1	$69.90		1	$69.90	Trossen Robotics	3/16/2010	FRS-B-CM-5
11.1V 2000mAh LiPo Battery	2	$72.99		2	$145.98	Trossen Robotics	3/16/2010	P-B-TP2000-3SPL
screws to attach dodecahedron halves	5	$6.56	25	1	$6.56	McMaster	3/16/2010	93705A247
washers for above screws	5	$8.88	25	1	$8.88	McMaster	3/16/2010	92217A440
thermo plastic insert	5	$8.33	50	1	$8.33	McMaster	3/30/2010	93365A150
screws for pulley	48	$11.20	25	2	$22.40	McMaster	3/30/2010	91239A704
spring 0.7 lbs/in	12	$7.53	12	1	$7.53	McMaster	4/29/2010	9654K711
Icosahedron Frame					$229.56			
PART NUMBER	**QTY.**	**price**	**qty in pack**		**extended price**	**supplier**		
bar	30	$35.85	5	6	$215.10	McMaster	3/16/2010	2153T41
corner	12			12	$0.00	printed		
Zinc-Plated Steel Split Ring .542" OD, .428" ID, .083" Thick	12	$9.60	100	1	$9.60	McMaster	3/24/2010	90177A213
set screw	60	$4.86	100	1	$4.86	McMaster	3/24/2010	92311A189
Total					**$1,094.95**			

Computer Code

Matlab Code for Single Value test

```
clc, clear, format compact
% function work = rolling_check_general(mf,mm,pmox,pmoy,l,n,r1,r2,mu,phi,omega)
%% inputs
% given by components
    l = .25; % m - edge length
    mf = .69; % kg - mass of frame
    mm = 1; % kg - mass of the mover
    n = 8; % number of sides or faces
    rm = 0.35*l; % radius of the moving mass
    type = 'pgp'; % pgp|ph  polygonal_prism --or-- polyhedron
    mu = .5; % coefficient of friction
% given by dynamics
    po = [0.084,0.03,0]; % m - center of oscillations - 0:1
    r1 = 0.4; % m - size of elipse - 0:1
    r2 = 0.03; % m - size of elipse - 0:1
    phi = 0.17; % rad - angle of elipse (x axis)
    omega = 11.5; % rad/sec - frequency
% other
    g = [0,9.81,0]; % m/sec^2 - earth's gravity
    tsnum = 10; % number of timesteps per radian
    pp = 'full';%625;%'full'; % for plotting - timestep to plot or 'full'
    revs = 4;
    store_as_movie = 0;
%% Calculations
t = (0:1/(tsnum*omega):2*pi/omega)'; % sec - calculation times
if strcmp(type,'pgp')
    ef = l/2*[1,-1/tan(pi/n),0]; % m - distance from centroid to front edge
    eb = ef.*[-1,1,1]; % m = distance from centroid to back edge
    ri = l/(2*tan(pi/n)); nn = 1;
elseif strcmp(type,'ph')
    switch n
        case 4, nn = 1;
        case 6, nn = 2;
        case 8, nn = 3;
        case 12, nn = 4;
        case 20, nn = 5;
        otherwise, error('Number of sides "n" must = 4, 6, 8, 12, or 20.')
    end
    y = 2*cos(pi/5); z = 2*sin(pi/5);
    ri = l*[1/sqrt(24),.5,1/sqrt(6),y^2/(2*z),y^2/(2*sqrt(3))]; % m - radius of inscribed sphere
    rfi = l*[sqrt(3)/6,.5,sqrt(3)/6,1/z,sqrt(3)/6]; % m - radius of inscribed circle on faces
    rfs = l/(2*sin(pi/[3,4,3,5,3]));
    ef = [rfi(nn),-ri(nn),0]; % m - distance from centroid to front edge
    eb = [rfs(nn),-ri(nn),0]; % m = distance from centroid to back edge
    clear y z nn
else
    error('type of solid must be defined properly')
end
ro = sqrt(ri^2+(l/2)^2);
po = po*ri-g/omega^2;
pod = sqrt(po(1)^2+po(2)^2);
r1 = r1*(ri-rm-pod);
r2 = r2*(ri-rm-pod);
pn = [r1*cos(omega*t)*cos(phi)-
r2*sin(omega*t)*sin(phi),r1*cos(omega*t)*sin(phi)+r2*sin(omega*t)*cos(phi),zeros(length(t),1)];
    % m - position of mover vs. po
po = po(ones(length(t),1),:); % makes po a 2D matrix with length(t) rows.
g = g(ones(length(t),1),:); % makes g a 2D matrix with length(t) rows.
ef = ef(ones(length(t),1),:); % makes ef a 2D matrix with length(t) rows.
eb = eb(ones(length(t),1),:); % makes eb a 2D matrix with length(t) rows.
p = po+pn; % m - position of mover
a = -omega^2*pn; % m/sec^2 - acceleration of mover
ff = -g*mf;
fm = -(g+a)*mm;
%% Fundamental calculations
```

```
mef = -cross(-ef,ff)-cross((p-ef),fm); mef = mef(:,3);
meb = cross(-eb,ff)+cross((p-eb),fm); meb = meb(:,3);
F_side = abs(ff(:,1)+fm(:,1));
F_up = ff(:,2)+fm(:,2);
F_up_max = -(mm+mf)*g(1,2)*01;
F_up2 = F_up.*(F_up<F_up_max)+(F_up>F_up_max).*F_up_max.*exp((F_up_max-F_up)/10);
mu_needed = F_side./-F_up; % Checking if the friction is high enough
%% Energy methods and equilibrium point
f = fm+ff; % the overall total force applied to the frame
pf = p(:,2)-fm(:,2)./fm(:,1).*p(:,1); % the position of the point where the lines of force intersect
pt = [-f(:,1)./f(:,2).*(pf-ef(:,2)),ef(:,2)]; % the point where a neutral tipping moment is located
pt_tipping = (pt(:,1)>l/2).*(pt(:,1)-10); % see next line for use
start_tip = find(pt_tipping==min(pt_tipping)); % in order to find when it starts tipping
v = omega*[r1*-sin(omega*t)*cos(phi)-r2*cos(omega*t)*sin(phi),-
r1*sin(omega*t)*sin(phi)+r2*cos(omega*t)*cos(phi),zeros(length(t),1)]; % m/s - velocity of mover
Ek = sum(1/2*mm*v.^2,2); % overall total kinetic energy
k = mm*omega^2; % spring constant in every direction
Ep = sum(1/2*k.*pn.^2,2); % overall total potential energy
deltaEm = 2*(Ek(start_tip)-min(Ek));
deltaEf = mf*9.81*(ro-ri);
if deltaEm>deltaEf
    E_check = 'works';
else
    E_check = 'does not work';
end
%% Results
maxmu_needed = max(mu_needed);
maxf = max(mef); maxb = max(meb); ampf = (maxf-min(mef))/2;
tip_check = (maxf-(maxb+abs(maxb))/2)/ampf;
friction_check = mu-maxmu_needed;
work = abs(tip_check*friction_check).*(and(tip_check>0,friction_check>0)*2-1);
fprintf('forward moment max = %6.4f\n',max(mef))
fprintf('backward moment max = %6.4f\n',max(meb))
fprintf('tip check %6.4f\n',tip_check)
fprintf('max mu needed = %6.4f\n',max(mu_needed))
fprintf('friction check %6.4f\n',friction_check)
fprintf('work = %6.4f\n',work)
fprintf('Energy check: dEm = %fJ, dEf = %fJ, %s\n',deltaEm,deltaEf,E_check)
%% Plots
e = eb(1,1:2);
ae = 0;
for ni = 1:n
    e = [e;e(end,1)+l*cos(ae),e(end,2)+l*sin(ae)]; %#ok<AGROW>
    ae = ae+2*pi/n;
end
prm = po+[(r1+rm)*cos(omega*t)*cos(phi)-(r2+rm)*sin(omega*t)*sin(phi),(r1+rm)*cos(omega*t)*sin(phi)...
    +(r2+rm)*sin(omega*t)*cos(phi),zeros(length(t),1)]; % m - position of mover vs. po
if strcmp(pp,'full'), ppp = 1:length(t); else ppp = pp; end
if store_as_movie, mov = avifile('moving_plot.avi','fps',15,'compression','None'); end
fighandle = figure(1);
for rev = 1:revs
for pp = ppp;
    tprmp = 0:0.1:2*pi; prm_perim = rm*[cos(tprmp'),sin(tprmp')];
    prm_perim(:,1) = prm_perim(:,1)+p(pp,1);
    prm_perim(:,2) = prm_perim(:,2)+p(pp,2);
    subplot(1,2,1)
        hold off
        plot(t,mef,'b',...
            t,meb,'r',...
            t,mu_needed,'g',...
            t,F_side/10,':k',...
            t,F_up/10,'k--'),
        hold on
        xlim([0,t(end)])
        axis(axis)
        plot(t(pp)*[1,1],[-10,10],'m-')
        plot([0,t(end)],[mu,mu],':g')
        plot([0,t(end)],[0,0],':k')
        xlabel('time (sec)'), title('Working system - time varient parameters')
%         legend('M_e_f','M_e_b','\mu_n_e_e_d_e_d','F_s_i_d_e','F_u_p')
    subplot(1,2,2)
```

```
        hold off
        axlim1 = floor(min(min([po(1,1);p(:,1);prm_perim(:,1);e(:,1);p(:,1)+fm(:,1)/20;p(:,1)-
mm*g(:,1)/20;pt(:,1)]))*50);
        axlim2 = ceil(max(max([po(1,1);p(:,1);prm_perim(:,1);e(:,1);p(:,1)+fm(:,1)/20;p(:,1)-
mm*g(:,1)/20;pt(:,1)]))*50);
        axlim3 = floor(min(min([po(1,2);p(:,2);prm_perim(:,2);e(:,2);p(:,2)+fm(:,2)/20;p(:,2)-
mm*g(:,2)/20;pt(:,2)]))*50);
        axlim4 = ceil(max(max([po(1,2);p(:,2);prm_perim(:,2);e(:,2);p(:,2)+fm(:,2)/20;p(:,2)-
mm*g(:,2)/20;pt(:,2)]))*50);
        axlim = [axlim1,axlim2,axlim3,axlim4]/50+[-1,1,-1,1]/50;
        plot(0,0,'+k',... plot centroid
            po(1,1),po(1,2),'+b',... plot the center of ellipse
            p(pp,1),p(pp,2),'mx',... plot actual position
            p(:,1),p(:,2),'-r',...
            ...prm(:,1),prm(:,2),':r',...
            prm_perim(:,1),prm_perim(:,2),'-k',...
            e(:,1),e(:,2),'-k',... [eb(1,1),ef(1,1)],[eb(1,2),ef(1,2)],'-k',...
                % plot the bottom side
            [p(pp,1),p(pp,1)+fm(pp,1)/20],[p(pp,2),p(pp,2)+fm(pp,2)/20],'g',...
                % plot the spring force vector
            [p(pp,1),p(pp,1)-mm*g(pp,1)/20],[p(pp,2),p(pp,2)-mm*g(pp,2)/20],'g:',...
                % plot the spring force vector
            [p(pp,1),p(pp,1)-mm*a(pp,1)/20],[p(pp,2),p(pp,2)-mm*a(pp,2)/20],'g:',...
                % plot the spring force vector
            [0,0],[0,ff(pp,2)/20],'g',... % plot the weight vector of frame
            pt(pp,1),pt(pp,2),'mx') % plot the position of the neutral tipping point
        axis equal
        axis(axlim)
        xlabel('horizontal axis (m)'), ylabel('vertical axis (m)'), title('positions and forces')
    pause(.005)
    if store_as_movie && floor(pp/4)==pp/4
        mov_frame = getframe(fighandle);
        mov = addframe(mov,mov_frame);
    end
end
end
if store_as_movie, mov = close(mov); end
```

Matlab Code for Genetic Algorithm

```
clc, clear, clf, format compact
%% Inputs
start_pop_size = 200;
child_pop_size = 200;
breeder_population = 40;
iterations = 50;
mutation_factor = 0.1;
experiments = 3;
%% Calculations
for experiment = 1:experiments
    best_iostuff = [];
    inputs = rand(start_pop_size,12);
    fprintf('experiment %2.0f\n   ',experiment)
%     comment this for speed
    plot1 = subplot(2,1,1);
    plot2 = subplot(2,1,2);
    for iteration = 1:iterations
        fprintf('.')
        if iteration == floor(iteration/10)*10, fprintf('%4.0f\n   ',iteration), end
        pop_size = length(inputs(:,1));
        iostuff = zeros(pop_size,15);
        for pi = 1:pop_size
            ri = inputs(pi,:); % inputs values 0:1
            % This is it!!! --------------
            [iostuff(pi,2),iostuff(pi,3),iostuff(pi,1)] =
rolling_check_general(ri(1),ri(2),ri(3),ri(4),ri(5),ri(6),ri(7),ri(8),ri(9),ri(10),ri(11),ri(12));
        end
        iostuff(:,4:15) = inputs; %#ok<AGROW> % valued 0:1
        iostuff = flipdim(sortrows(iostuff,1),1); % sorts the rows in decending order based on fitness
        iostuff = iostuff(1:breeder_population,:); % eliminates the worse results
```

```
%         comment next line to make best population irrelevant
        iostuff = [iostuff;best_iostuff]; %#ok<AGROW> % adds on the best results from ever
        iostuff = flipdim(sortrows(iostuff,1),1); % sorts the rows in decending order based on fitness
        best_iostuff = iostuff(1:breeder_population,:); % eliminates the worse results
        best(iteration,:) = [iostuff(1,:),mean(best_iostuff(:,1))]; %#ok<AGROW>
        inputsnew = zeros(child_pop_size,12);
        for pi = 1:child_pop_size
            r = ceil(rand(1,12)*breeder_population);
            for c = 1:12
                inputsnew(pi,c) = inputs(r(c),c);
            end
        end
        inputs = inputsnew+mutation_factor*(rand(size(inputsnew))-.5);
        one = find(inputs>1); inputs(one) = 1;
        zero = find(inputs<0); inputs(zero) = 0;
%         comment this for speed
        subplot(plot1), axis normal,
plot(1:iteration,best(:,1),'b',1:iteration,best(:,2),'r',1:iteration,best(:,3),'g'), hold on,
axis(axis), plot(1:iteration,best(:,16),'b:'), hold off % fitness,work,idealness, average fitness
        subplot(plot2), plot(best_iostuff(:,1))
        pause(0.005)
    end
    fprintf('\n')
%     subplot(plot1), legend ('fitness','work','idealness'), hold off
    ri = iostuff(1,4:end);
    [mf,mm,po,l,n,r,rm,mu,phi,omega,g,ri2,idealness] =
find_vars(ri(1),ri(2),ri(3),ri(4),ri(5),ri(6),ri(7),ri(8),ri(9),ri(10),ri(11),ri(12)); % find the actual
value of the variables instead of the 0:1 "inputs"
    experiment_result(experiment,:) = [iostuff(1,1:3),mf,mm,po,l,n,r,rm,mu,phi,omega,ri2]; %#ok<AGROW>
    dlmwrite('all_results.csv', experiment_result(experiment,:), '-append')
    clear best bes_iostuff inputs inputsnew iostuff
end
best_fitness = max(experiment_result(:,1));
f = find(experiment_result(:,1) == best_fitness);
r = experiment_result(f,:);
fprintf('fitness = %5.3f\n',r(1))
fprintf('work = %5.3f\n',r(2))
fprintf('idealness = %5.3f\n',r(3))
fprintf('mf = %5.3f kg\n',r(4))
fprintf('mm = %5.3f kg\n',r(5))
fprintf('po = %5.3f  %5.3f  %5.3f m\n',r(6),r(7),r(8))
fprintf('l = %5.3f m\n',r(9))
fprintf('n = %2.0f\n',r(10))
fprintf('r = %5.4f  %5.4f m\n',r(11),r(12))
fprintf('rm = %5.3f m\n',r(13))
fprintf('mu = %4.2f\n',r(14))
fprintf('phi = %4.2f radians\n',r(15))
fprintf('omega = %5.2f rad/sec\n',r(16))
fprintf('(ri = %5.3f m)\n',r(17))



-----------------------------------------------------------------
function [work,idealness,fitness] =
rolling_check_general(mfin,mmin,poxin,poyin,lin,nin,r1in,r2in,rmin,muin,phiin,omegain)
% mfin=0;mmin=0;poxin=0;poyin=0;lin=0;nin=0;r1in=0;r2in=0;rmin=0;muin=0;phiin=0;omegain=0;
[mf,mm,po,l,n,r,rm,mu,phi,omega,g,ri,idealness] =
find_vars(mfin,mmin,poxin,poyin,lin,nin,r1in,r2in,rmin,muin,phiin,omegain);
%%
% q = [0,0,0,mf,mm,po,l,n,r,rm,mu,phi,omega,ri];
% fprintf('mf = %5.3f kg\n',q(4))
% fprintf('mm = %5.3f kg\n',q(5))
% fprintf('po = %5.3f  %5.3f  %5.3f m\n',q(6),q(7),q(8))
% fprintf('l = %5.3f m\n',q(9))
% fprintf('n = %2.0f\n',q(10))
% fprintf('r = %5.4f  %5.4f m\n',q(11),q(12))
% fprintf('rm = %5.3f m\n',q(13))
% fprintf('mu = %4.2f\n',q(14))
% fprintf('phi = %4.2f radians\n',q(15))
% fprintf('omega = %5.2f rad/sec\n',q(16))
% fprintf('(ri = %5.3f m)\n\n\n',q(17))
```

```
%% Calculations
tsnum = 100; % number of calculation timesteps in each radian
ef = l/2*[1,-1/tan(pi/n),0]; % m - distance from centroid to edge
eb = ef.*[-1,1,1]; % m - distance from centroid to edge
t = (0:1/(tsnum*omega):2*pi/omega)'; % sec - calculation times
pn = [r(1)*cos(omega*t)*cos(phi)-
r(2)*sin(omega*t)*sin(phi),r(1)*cos(omega*t)*sin(phi)+r(2)*sin(omega*t)*cos(phi),zeros(length(t),1)]; %
m - position of mover vs. po
po = po(ones(length(t),1),:); % makes po a 2D matrix with length(t) rows.
g = g(ones(length(t),1),:); % makes g a 2D matrix with length(t) rows.
ef = ef(ones(length(t),1),:); % makes e a 2D matrix with length(t) rows.
eb = eb(ones(length(t),1),:); % makes e a 2D matrix with length(t) rows.
p = po+pn; % m - position of mover
a = -omega^2*pn; % m/sec^2 - acceleration of mover
ff = -g*mf;
fm = -(g+a)*mm;
%% Fundamental calculations
mef = -cross(-ef,ff)-cross((p-ef),fm); mef = mef(:,3);
meb = cross(-eb,ff)+cross((p-eb),fm); meb = meb(:,3);
F_side = abs(ff(:,1)+fm(:,1));
F_up = ff(:,2)+fm(:,2);
F_up_max = -(mm+mf)*g(1,2)*0.01;
F_up = F_up.*(F_up<F_up_max)+(F_up>F_up_max).*F_up_max.*exp((F_up_max-F_up)/10);
mu_needed = F_side./-F_up; % Checking if the friction is high enough
%% Results
mu_needed = max(mu_needed);
maxf = max(mef); maxb = max(meb); ampf = (maxf-min(mef))/2;
tip_check = (maxf-(maxb+abs(maxb))/2)/ampf;
friction_check = mu-mu_needed;
work = abs(tip_check*friction_check).*(and(tip_check>0,friction_check>0)*2-1);
idealness = idealness/160;
fitness = work+idealness;
%% comment out this section for function use
% clc
% fprintf('Fitness = %6.4f\n',fitness)
% fprintf('Work = %6.4f\n',work)
% fprintf('Idealness = %6.4f\n',idealness)
%%


----------------------------------------------------------------------
function [mf,mm,po,l,n,r,rm,mu,phi,omega,g,ri,idealness] =
find_vars(mfin,mmin,poxin,poyin,lin,nin,r1in,r2in,rmin,muin,phiin,omegain)
%% inputs
% given by components
    l = 0.25;%0.25;%0.25;%0.25; % m - edge length
    mf = [0.7,0.74];%[0.65,0.73];%[0.53,0.73];%[0.3,0.8]; % kg - mass of frame
    mm = 1;%1;%1;%1; % kg - mass of the mover
    n = 8;%8;%8;%8; % number of sides or faces
    rm = [0.36,0.38].*l;%[0.33,0.37].*l;%[0.2,0.35].*l;%[0.2,0.35].*l; % radius of the moving mass
    mu = 0.7;%0.7;%0.7;%[0.7,0.8]; % coefficient of friction
% given by dynamics
    pox = [0.08,0.085];%[0.08,0.088];%[0.08,0.12];%[0.1,.135]; % unitless - center of oscillations /
inscribed radius
    poy = [0.026,0.037];%[0.023,0.037];%[0.02,0.05];%[-0.02,0.08];
    r1 = [0.33,0.43];%[0.3,0.5];%[0.25,0.55];%[0.2,0.6]; % m - size of elipse vs. maximum allowed
    r2 = [0,0.04];%[0,0.07];%[0,0.07];%[0,0.15]; % m - size of elipse vs. maximum allowed
    phi = [0.12,0.16];%[0.10,0.18];%[0.09,0.2];%[0.02,0.12]; % rad - angle of elipse (x axis)
    omega = [11.5,13];%[10.5,12.5];%[10.5,12.5];%[10.5,12.5]; % rad/sec - frequency
% other
g = [0,9.81,0]; % m/sec^2 - earth's gravity
%% Initial calculations
mm = mm(1)+(mm(end)-mm(1))*mmin;
mf = mf(1)+(mf(end)-mf(1))*mfin;
pox = pox(1)+(pox(end)-pox(1))*poxin;
poy = poy(1)+(poy(end)-poy(1))*poyin;
l = l(1)+(l(end)-l(1))*lin;
n = ceil(n(1)+(n(end)-n(1))*nin);
r1 = r1(1)+(r1(end)-r1(1))*r1in;
r2 = r2(1)+(r2(end)-r2(1))*r2in;
rm = rm(1)+(rm(end)-rm(1))*rmin;
```

```
mu = mu(1)+(mu(end)-mu(1))*muin;
phi = phi(1)+(phi(end)-phi(1))*phiin;
omega = omega(1)+(omega(end)-omega(1))*omegain;
%% Comment out for genetic algorithm function
% mfin = rand;
% mmin = rand;
% poxin = rand;
% poyin = rand;
% lin = rand;
% nin = rand;
% r1in = rand;
% r2in = rand;
% rmin = rand;
% muin = rand;
% phiin = rand;
% omegain = rand;
%% Comment out for genetic algorithm function
% mf = 0.353;
% mm = 0.879;
% pox = 0.047;
% poy = -0.087;
% l = 0.244;
% n =  8;
% r1 = 0.0204;
% r2 = .0441;
% rm = 0.141;
% mu = 0.78;
% phi = 1.61;
% omega = 12.65;
%% Ideal values, IMportance inputs, general Size inputs, idealness calculations
mfid    = 1.0 ; mfim    = 0.4; mfs    = 0.3;
mmid    = 1.3 ; mmim    = 0.2; mms    = 0.4;
poxid   = 0   ; poxim   = 0.2; poxs   = 0.05;
poyid   = 0   ; poyim   = 0.4; poys   = 0.05; % m and k are not yet defined
lid     = 0.4 ; lim     = 0.2; ls     = 0.2;
nid     = 4   ; nim     = 0.4; ns     = 3;
r1id    = 0   ; r1im    = 0.2; r1s    = 1;
r2id    = 0   ; r2im    = 0.2; r2s    = 1;
rmid    = 0.16; rmim    = 0.4; rms    = 0.05;
muid    = 0.2 ; muim    = 0.6; mus    = 0.4;
phiid   = 0   ; phiim   = 0.0; phis   = pi;
omegaid = 4*pi; omegaim = 1.0; omegas = 20;
mfid =    ((mf-mfid)/mfs).^2*mfim;
mmid =    ((mm-mmid)/mms).^2*mmim;
poxid =   abs((pox-poxid)/poxs).^3*poxim;
poyid =   abs((poy-poyid)./poys).^3*poyim;
lid =     ((l-lid)/ls).^2*lim;
nid =     abs((abs(n)-nid)/ns).^3*nim;
r1id =    ((r1-r1id)/r1s).^2*r1im;
r2id =    ((r2-r2id)/r2s).^2*r2im;
rmid =    ((rm-rmid)/rms).^2*rmim;
muid =    ((mu-muid)/mus).^4*muim;
phiid =   ((phi-phiid)/phis).^2*phiim;
omegaid = ((omega-omegaid)/omegas).^2*omegaim;
idealness = mfid+mmid+poxid+poyid+lid+nid+r1id+r2id+rmid+muid+phiid+omegaid;
idealness = 0-idealness(:,:,:,:,:,:,:,:,:,:,:,:,1);
%% more calcs
ri = l/(2*tan(pi/n));
po = [pox,poy,0].*(ri-rm-g./omega^2)-g./omega^2;
pod = sqrt(po(1)^2+po(2)^2);
r = [r1,r2]*(ri-rm-pod);
if r<0, r = [0,0]; end
```

C++ Code for the Genetic Algorithm with Octagonal Frame

```
#include "Vx/VxBox.h"
#include "Vx/VxConvexMesh.h"
#include "Vx/VxCollisionGeometry.h"
#include "Vx/VxConstraintController.h"
#include "Vx/VxContactProperties.h"
#include "Vx/VxEventSubscriber.h"
#include "Vx/VxFrame.h"
#include "Vx/VxPart.h"
#include "Vx/VxPlane.h"
#include "Vx/VxSphere.h"
#include "Vx/VxSpring.h"
#include "Vx/VxUniverse.h"
#include "Vx/VxVector3.h"
#include "Vx/VxVisualizer.h"
#include "fga\include\fga.hpp"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <fstream>
//#include "fga\include\test_include.hpp"

#define pi 3.1415926 // cause the software doesnt know what pi is
#define false 0
#define true 1

using namespace Vx; // allows use of the stuff that begins with Vx...
VxReal3 g = {0,0,-9.81}; // gravity constant. For some reason it fails if this is set as type 'const'

#define visualizeron false
#define frictionboxon false
#define ruleron false

VxPart* shape; // define a pointer to the frame and give it a mass. we cant call it frame cause thats a
class of something
VxPart* ball; // define a pointer to the ball and give it a mass

VxUniverse* universe; // defines a global variable for the universe

/*template <class GeneType>
class Population;

Population<char> *p;*/

double fitness_function(double *chromosome) // outputs the fitness with inputs of an array pointer
called chromop
{
        // Start things up
        using namespace Vx;
        universe = new VxUniverse;
        int running = 1; // set some variable to say that running is true
        VxFrame* const frame = VxFrame::instance(); // makes a frame called 'frame'
        const float time_step = 1.0f/60.0f; // set time step
        frame->setTimeStep(time_step); // sets an iteration timestep
        VxFrameRegisterAllInteractions(frame); // ?
        const double theta = 3*pi/2; // Starting angle around the elipse. zero is at right at
{r1*cos(phi),0,r1*sin(phi)}
        double ang_shape_total; // initial angle.
        double fitness; // function output
        float cond1 = 0.2; // the centroid translates in the x direction
        float cond2 = 0.2; // the centroid only slightly translates in the y direction
        float cond3 = 0.6; // (x displacement) == l*n*(theta displacement)


        frame->addUniverse(universe); // updates the variables at the pointer frame to include universe

        // set up visualizer

        universe = frame->getUniverse(0);
        universe->setAutoSleep(false);
```

```
	// find values of each parameter
		const int n = 8; // number of sides of the frame
		const double rm = 0.12; // radius of the moving mass
		const double mu = 0.5; // defines the coefficient of friction. This representation is
only approximate, but in the X and y directions its not bad.
	// Given by components
		double offsetRelMin=-0.3, offsetRelMax=0.3;
		double phiMin=-0.3, phiMax=0.8;
		double omegaMin=12, omegaMax=15;
		double r1RelMin=0.1, r1RelMax=0.4;
		double r2RelMin=-0.4, r2RelMax=0.4;
		double lMin=0.2, lMax=0.4;
		double mmMin=0.8, mmMax=2.0;
		double kMin=50, kMax=200;

	// derived values
		// poxRel,poyRel,r1Rel,r2rel,phi,omega,l,mm,stiffness
		double l = (lMax-lMin)*chromosome[6]+lMin; // the length of one of the sides
		double mf = n*(0.037+0.368*l); // the mass of the frame "shape".
		double offsetRel[3]={(offsetRelMax-
offsetRelMin)*chromosome[0]+offsetRelMin,(offsetRelMax-offsetRelMin)*chromosome[1]+offsetRelMin,0};
		double r1Rel = (r1RelMax-r1RelMin)*chromosome[2]+r1RelMin;
		double r2Rel = (r2RelMax-r2RelMin)*chromosome[3]+r2RelMin;
		double phi = (phiMax-phiMin)*chromosome[4]+phiMin;
		double omega = (omegaMax-omegaMin)*chromosome[5]+omegaMin;
		double mm = (mmMax-mmMin)*chromosome[7]+mmMin; // the mass of the moving mass "ball"
		double stiffness = (kMax-kMin)*chromosome[8]+kMin; // the stiffness of the spring used to
retain the ball.

		double ri = l/(2*tan(pi/n)); // the inscribed radius of the solid
		double ro = l/(2*sin(pi/n)); // the circumscribed radius of the solid
		double offset[3];
		for (int i=0;i<3;i++) offset[i] = offsetRel[i]*ri; // the actual offset of the neutral
position (not relative as above)
		double linearOffset =
pow(offset[0]*offset[0]+offset[1]*offset[1]+offset[2]*offset[2],0.5); // the linear distance of the
actual offset
		double reduced_ri = ri-rm-linearOffset-mm*g[2]/stiffness;
		if(reduced_ri<0) reduced_ri=0;
		double r1 = r1Rel*reduced_ri; // the actual principle radius of the elipse. The first
number is the relative length w/ range (0:1).
		double r2 = r2Rel*reduced_ri; // secondary radius. see above
		double p_shape_0[3] = {0, 0, ri}; // define the position of the frame

	// Set up parts
	// SHAPE
		shape = new VxPart(mf);
		double shapeCenter[2] = {-4,4}; // defined the Y coordinate of the center of the ends of
the shape.
		double shape_points[2*n][3]; // The set of points created relative to the center of the
part.
		for (int i = 0; i<2; i++){ // for each end...
			for (int ii = 0; ii<n; ii++){ // ...for each point on that end...
				shape_points[ii+i*n][0] = ro*(cos(-pi/2+2*pi/n*(ii+0.5))); // ...define
the coordinates of that point
				shape_points[ii+i*n][1] = shapeCenter[i];
				shape_points[ii+i*n][2] = ro*(sin(-pi/2+2*pi/n*(ii+0.5)));
			}
		}
		float shapecolor[4] = {1, 0, 0, 0.5}; // red, semi-transparent
		shape->setName("shape");
		shape->addGeometry(new VxConvexMesh(shape_points, 2*n), 0, 0);
		shape->setControl(VxEntity::kControlDynamic);
		shape->setPosition(p_shape_0);
		universe->addEntity(shape);
		shape->freeze(false);
		shape->updateNode();

	// BALL
		ball = new VxPart(mm);
```

```
		VxReal3 p_ball_0 =
{p_shape_0[0]+offset[0],p_shape_0[1]+offset[1],p_shape_0[2]+offset[2]+mm*g[2]/stiffness};
		ball->setName("ball");
		float ballcolor[4] = {0, 1, 0, 1}; // green, opaque
		//ball->addGeometry(new VxSphere(rm), 0, 0, 0.5); // This must all be commented so that
the ball does not interfere with the shape
		ball->setPosition(p_ball_0);
		universe->addEntity(ball);
		ball->updateNode();

	// PLANE
		VxReal3 p_plane = {0, 0, 0};
		VxPart* plane = new VxPart();
		plane->setName("plane");
		plane->addGeometry(new VxPlane(), 0); // Note: This actuall makes an entire plane.
		universe->addEntity(plane);
		plane->freeze(true);
		plane->setPosition(p_plane);

	// declare the variables for use with the spring
	VxReal3 p_ball = {0,0,0};
	VxReal3 p_shape = {0,0,0};
	VxReal3 ang_shape = {0,0,0};
	VxReal3 p_ball_old, p_shape_old, springVector, springForce, springTorque, ang_shape_old;
	float ballLineColor[4] = {.2, 1, .2, 1};
	float shapeLineColor[4] = {1, .2, .2, 1};
	double ponew[3];
	double activeOffset[3] = {0,0,0};
	int i;

	// Set the friction definitions
	VxContactProperties* cp = universe->getContactProperties(0, 0);
	cp->setFrictionType(VxContactProperties::kFrictionTypeTwoDirection);
	cp->setFrictionModel(VxContactProperties::kFrictionModelScaledBox);
	universe->setGravity(g);
	cp->setFrictionCoefficientPrimary(mu);
	cp->setFrictionCoefficientSecondary(mu);

	VxReal totalTime = 0;
	ang_shape_total = 0;
	double maxy = 0;
	const double maxTime = 30;

	do // Roll. Loops for each timestep
	{
		frame->step(); // Actually runs the simulation
		totalTime += frame->getTimeStep();
		//ball->getLinearVelocity(vel); // you need the linear velocity if you have a cutoff for
when the ball starts going slowly. see the while condition below for this loop.
		for (i=0; i<3; i++){
			p_ball_old[i]=p_ball[i];
			p_shape_old[i]=p_shape[i];
			ang_shape_old[i]=ang_shape[i];
		};
		ball->getPosition(p_ball); // gets the position of the ball
		shape->getPosition(p_shape); // gets position of the shape
		shape->getOrientationEulerXYZ(ang_shape); // initial angle.
		ang_shape_total+=(ang_shape[0]-ang_shape_old[0]);
		for (i=0; i<3; i++) ponew[i] = offset[i]+p_shape[i]; // redefines the position of the
center of the shape
		if (p_shape[2]>maxy) maxy=p_shape[2];
		if (totalTime>1){
			activeOffset[0] = r1*cos(omega*totalTime+theta)*cos(phi)-
r2*sin(omega*totalTime+theta)*sin(phi);
			activeOffset[2] =
r1*cos(omega*totalTime+theta)*sin(phi)+r2*sin(omega*totalTime+theta)*cos(phi);
		};
		for (i=0; i<3; i++) {
			springVector[i] = p_ball[i]-ponew[i]-activeOffset[i]; // defibnes the difference
in location of the ball from where it should be.
```

```
                        springForce[i] = -springVector[i]*stiffness;}; // defines the force that the
spring exerts on the BALL
                ball->addForce(springForce); // applies said force to the ball
                for (i=0; i<3; i++) springForce[i] *= -1; // redefines the force as that which is applied
to the FRAME
                springTorque[0] = offset[1]*springForce[2]-offset[2]*springForce[1]; // defines the
torque that acts on the shape due to the location of the ball within it not being the center of mass
                springTorque[1] = offset[2]*springForce[0]-offset[0]*springForce[2];
                springTorque[2] = offset[0]*springForce[1]-offset[1]*springForce[0];
                shape->addForce(springForce); // adds a force to the frame
                shape->addTorque(springTorque); // adds the torque to the frame
        }       while ((running) && totalTime < maxTime);// && vel.norm() > err_vel); // keep rolling
WHILE ...

        double fit1 = p_shape[0]/(2*pi*l*n); // the centroid translates in the x direction
        double fit2 = -abs(maxy-ro)/(ro-ri); // the centroid only slightly translates in the y direction
        double fit3 = -abs(p_shape[0]/l/n-ang_shape_total)/1; // (x displacement) == l*n*(theta
displacement)
        fitness = (fit1*cond1)+(fit2*cond2)+(fit3*cond3);
        printf("fitness = %f   ",fitness);
        printf("%4.2f, %4.2f, %4.2f, %4.2f, %4.2f, %4.2f, %4.2f, %4.2f,
%4.2f\n",chromosome[0],chromosome[1],chromosome[2],chromosome[3],chromosome[4],chromosome[5],chromosome[
6],chromosome[7],chromosome[8]);
        // write a file
                FILE * output_text_file;
                output_text_file = fopen("output_text_file.csv","a");
                if (output_text_file!=NULL)
                {

        fprintf(output_text_file,"%7.5f,%7.5f,%7.5f,%7.5f,%7.5f,%7.5f,%7.5f,%7.5f,%7.5f,%7.5f\n",fitness,
chromosome[0],chromosome[1],chromosome[2],chromosome[3],chromosome[4],chromosome[5],chromosome[6],chromo
some[7],chromosome[8]);
                        fclose(output_text_file);
                };

        // Shut down procedures
        frame->release();
        return fitness;
}
double random_gene() // returns a gene between 0 and 1
{
        double random_num = rand(); // This line can get deleted later once it all works.
        double random_gene = random_num/RAND_MAX; // makes it between 0 - 1
        //printf("%4.2f, ",random_gene);
        return random_gene;
}
double mutate_gene(double gene)
{
        float mutate_limit = 0.05;
        double random_num = rand();
        random_num = random_num/RAND_MAX*mutate_limit*2-mutate_limit; // makes it between -mutate_limit :
mutate_limit
        double mutated_gene;
        if (gene+random_num>1 || gene+random_num<0)
                mutated_gene = gene-random_num;
        else
                mutated_gene = gene+random_num;
        //printf(".");
        return mutated_gene;
}
int main()//int argc, const char *argv[])
{
        int population_size = 100;
        double min_tolerated_fitness = 0.2;
        int max_generations = 500;
        int length_of_chromosomes = 9; // poxRel,poyRel,r1Rel,r2rel,phi,omega,l,mm,stiffness
        float my_mutation_rate = 0.3;
        using namespace fga;
        srand((unsigned)time(NULL));
        //print time
                time_t rawtime;
```

```
		struct tm * timeinfo;
		time ( &rawtime );
		timeinfo = localtime ( &rawtime );
	// write a file
		FILE * output_text_file;
		output_text_file = fopen("output_text_file.csv","a");
		if (output_text_file!=NULL)
		{
			fprintf (output_text_file,"\n%s",asctime(timeinfo));

	fprintf(output_text_file,"fitness,poxRel,poyRel,r1Rel,r2rel,phi,omega,l,mm,stiffness\n");
			fclose(output_text_file);
		};
	// write a file
		FILE * result_text_file;
		result_text_file = fopen("result_text_file.csv","a");
		if (result_text_file!=NULL)
		{
			fprintf (result_text_file,"\n%s",asctime(timeinfo));
			fprintf(result_text_file,"best,average,stdev\n");
			fclose(result_text_file);
		};
	Population<double> my_population(population_size, length_of_chromosomes, (float (__cdecl
*)(double *))fitness_function, mutate_gene, random_gene, NULL, NULL, NULL, NULL);
	using namespace fga;
	my_population.set_mutation_rate(my_mutation_rate);

	my_population.run(min_tolerated_fitness, max_generations);
	float best_score = my_population.get_best_score();
	double *best_chromosome = my_population.get_best();
	float fitness = my_population.get_best_score();
	printf("\nthe fitness is %f\n",fitness);
	bool terminate = my_population.has_converged();
	my_population.save("mypopulation.txt");
//	int returnval = fitness*1000;
	return 1; //returnval;
}
```

C++ Code for a Manual Test with Visualization for the Icosahedron

```
#include "Vx/VxBox.h"
#include "Vx/VxConvexMesh.h"
#include "Vx/VxCollisionGeometry.h"
#include "Vx/VxConstraintController.h"
#include "Vx/VxContactProperties.h"
#include "Vx/VxEventSubscriber.h"
#include "Vx/VxFrame.h"
#include "Vx/VxPart.h"
#include "Vx/VxPlane.h"
#include "Vx/VxSphere.h"
#include "Vx/VxSpring.h"
#include "Vx/VxUniverse.h"
#include "Vx/VxVector3.h"
#include "Vx/VxVisualizer.h"
// #include "fga\include\fga.hpp"
#include <stdio.h>

#define pi 3.1415926 // cause the software doesnt know what pi is

using namespace Vx; // allows use of the stuff that begins with Vx...

// Given by components
const double golden_ratio = (1+pow(5,.5))/2; // used to find the vertices of the solid.
const double rm = 0.12; // radius of the moving mass
const double mu = 0.7; // defines the coefficient of friction. This representation is only approximate,
but in the X and y directions its not bad.
```

```
VxReal3 g = {0,0,-9.81}; // gravity constant. For some reason it fails if this is set as type 'const'
const double theta = 3*pi/2; // Starting angle around the elipse. zero is at right at
{r1*cos(phi),0,r1*sin(phi)}

/*double chromosome[9] = {0.52907,    0.93011,       0.40000,       0.00000,       0.07475,
       0.39320,       0.30971,       0.45771,       0.7};
              //                          poxRel,        poyRel,        r1Rel,         r2rel,
       phi,           omega,         l,                    mm,                    stiffness
              //                          -0.3:0.8       -0.3:0.8       0.1:0.4        -0.4:0.4
       -0.3:0.8       6:15           0.2:0.4        0.8:2.0        50:200
// Given by components
       double offsetRelMin=-0.3, offsetRelMax=0.3;
       double phiMin=-0.3, phiMax=0.8;
       double omegaMin=6, omegaMax=15;
       double r1RelMin=0.1, r1RelMax=0.4;
       double r2RelMin=-0.4, r2RelMax=0.4;
       double lMin=0.2, lMax=0.4;
       double mmMin=0.8, mmMax=2.0;
       double kMin=50, kMax=200;
// derived values
       double l = (lMax-lMin)*chromosome[6]+lMin; // the length of one of the sides
       double mf = n*(0.037+0.368*l); // the mass of the frame "shape".
       double offsetRel[3]={(offsetRelMax-offsetRelMin)*chromosome[0]+offsetRelMin,(offsetRelMax-
offsetRelMin)*chromosome[1]+offsetRelMin,0};
       double r1Rel = (r1RelMax-r1RelMin)*chromosome[2]+r1RelMin;
       double r2Rel = (r2RelMax-r2RelMin)*chromosome[3]+r2RelMin;
       double phi = (phiMax-phiMin)*chromosome[4]+phiMin;
       double omega = (omegaMax-omegaMin)*chromosome[5]+omegaMin;
       double mm = (mmMax-mmMin)*chromosome[7]+mmMin; // the mass of the moving mass "ball"
       double stiffness = (kMax-kMin)*chromosome[8]+kMin; // the stiffness of the spring used to retain
the ball.
*/
// Manual values
       double l = 0.3;
       //double offsetRel[3] = {0.4,0.1,0.0};
       //double r1Rel = 0.5;
       //double r2Rel = 0.08;
       double r1 = 0.02;
       double r2 = 0;
       double offset[3] = {0.02,0.005,0};
       double phi = 0.2;
       double mm = 1.4+0.122*2;
       double stiffness = 4*0.7*(39.37*4.448); // (converts in/lb to N/m) //500;//omega*omega*mm;
       double omega = 2.25*2*pi;//pow(stiffness/mm,.5);
       double mf = 0.70;//0.6693; // the mass of the frame "shape".

double ri, ro, linearOffset;//, r1, r2, offset[3];
float time_step = 1.0f/60.0f; // set time step
const double cond1 = 0.2; // the centroid translates in the x direction
const double cond2 = 0.2; // the centroid only slightly translates in the y direction
const double cond3 = 0.6; // (x displacement) == l*n*(theta displacement)

VxPart* shape; // define a pointer to the frame and give it a mass. we cant call it frame cause thats a
class of something
VxPart* ball; // define a pointer to the ball and give it a mass
VxPart* frictionBox; // define a pointer to a box that we use to test the friction. It may be hidden.
VxReal3 frictionBoxForce = {.5085,0,-1}; // The x coordinate here, when the object just starts to drift
away, is the coefficient of friction.
double p_shape_0[3] = {0, 0, l*golden_ratio/2}; // define the position of the frame
double max_spring_force = 0;
VxReal3 camTarget = { 0, 0, .3}, camPos = {0, 0, 5}; // Target and position of the camera for viewing.
double maxTime = 50;
#define false 0
#define true 1
#define visualizeron true
#define frictionboxon false
#define ruleron false

VxUniverse* universe; // defines a global variable for the universe
```

```
double mainnew(int argc, const char *argv[])
{
	// Start things up
	universe = new VxUniverse;
	int running = 1; // set some variable to say that running is true
	VxFrame* const frame = VxFrame::instance(); // makes a frame called 'frame'
	frame->setTimeStep(time_step); // sets an iteration timestep
	VxFrameRegisterAllInteractions(frame); // ?
	double ang_shape_total; // initial angle.
	double current_spring_force;
	double fitness;

	frame->addUniverse(universe); // updates the variables at the pointer frame to include universe

	// set up visualizer
	#if visualizeron
		VxVisualizer* visualizer;
		visualizer = VxVisualizer::create();
		visualizer->createDefaultWindow(argc, argv, "Example of friction force"); // This is the
only line that uses argv and argc
		visualizer->setFrameRateLock(true);
		visualizer->setUniverse(universe); // updates the variables at the pointer visualizer to
include universe
	#endif


	universe = frame->getUniverse(0);
	universe->setAutoSleep(false);
	#if visualizeron
		visualizer->setUniverse(universe);
		visualizer->startViewer();
		//visualizer->addHelpString("Press F10 to reset"); // top help string
		//visualizer->addHelpString("This example demonstrates the scale box friction effect
along different velocity directions.");
		visualizer->createHelp();
		visualizer->toggleHelp();
		visualizer->setFrameRateLock(false);
		//VxReal3 camTarget = { 0, 0, 0}, camPos = {-2, -4, 1};
		visualizer->setCameraLookAtAndPosition(camTarget, camPos);
		visualizer->setPaused(false);
	#endif

	// derived values
		int i=0;
		int ii=0;
		//ri = l/(2*tan(pi/n)); // the inscribed radius of the solid
		//ro = l/(2*sin(pi/n)); // the circumscribed radius of the solid
		//for (i=0;i<3;i++) offset[i] = offsetRel[i]*ri; // the actual offset of the neutral
position (not relative as above)
		linearOffset = pow(offset[0]*offset[0]+offset[1]*offset[1]+offset[2]*offset[2],0.5); //
the linear distance of the actual offset
		//r1 = r1Rel*(ri-rm-linearOffset); // the actual principle radius of the elipse. The
first number is the relative length w/ range (0:1).
		//r2 = r2Rel*(ri-rm-linearOffset); // secondary radius. see above

	// Set up parts
	// SHAPE
		shape = new VxPart(mf);
		double shapeCenter[2] = {-4,4}; // defined the Y coordinate of the center of the ends of
the shape.
		double gl = golden_ratio*l;
		double shape_points[12][3] = {{l,gl,0},{-l,gl,0},{l,-gl,0},{-l,-gl,0},{0,l,gl},{0,-
l,gl},{0,l,-gl},{0,-l,-gl},{gl,0,l},{-gl,0,l},{gl,0,-l},{-gl,0,-l}}; // The set of points created
relative to the center of the part.
		for (i=0;i<12;i++)
			for (ii=0;ii<3;ii++)
				shape_points[i][ii] /= 2;
		//shape_points[row][column] = stuff
		// for rectangle 1
		// for rectangle 2
		// for rectangle 3
```

```
		float shapecolor[4] = {1, 0, 0, 0.5}; // red, semi-transparent
		shape->setName("shape");
		shape->addGeometry(new VxConvexMesh(shape_points, 12), 0, 0);
		#if visualizeron
			shape->setNode(visualizer->createNodeFromGeometry(shape, shapecolor)); //Note that
Vx Visualizer.h needs to be modified at lines 217 and 218 to not call the create pastel color function.
		#endif
		shape->setControl(VxEntity::kControlDynamic);
		shape->setPosition(p_shape_0);
		universe->addEntity(shape);
		shape->freeze(false);
		shape->updateNode();

	// BALL
		ball = new VxPart(mm);
		VxReal3 p_ball_0 = {p_shape_0[0],p_shape_0[1],p_shape_0[2]+mm*g[2]/stiffness};
		ball->setName("ball");
		float ballcolor[4] = {0, 1, 0, 1}; // green, opaque
		//ball->addGeometry(new VxSphere(rm), 0, 0, 0.5); // This must all be commented so that
the ball does not interfere with the shape
		#if	visualizeron
			ball->setNode(visualizer->createSphere(rm, ballcolor, 0, true, true));
		#endif
		//p_ball_0[0] = 0;
		//p_ball_0[1] = 0;
		//p_ball_0[2] = 1;
		ball->setPosition(p_ball_0);
		universe->addEntity(ball);
		ball->updateNode();

	// RULER
		#if ruleron
			VxPart* ruler = new VxPart(0.001);
			double rulerSize[3] = {1,0.03,0.005}; // about meter stick size.
			float rulercolor[4] = {1, 1, 0, 1}; // yellow, opaque
			VxReal3 p_ruler = {0.5,0,1};
			ruler->setName("ruler");	// give a name to the part to be able to retrieve it
later.
			//ruler->addGeometry(new VxBox(rulerSize), 0, 0);
			#if visualizeron
				ruler->setNode(visualizer->createBox(rulerSize,rulercolor,0,true,true));
			#endif
			ruler->setControl(VxEntity::kControlDynamic);
			ruler->setPosition(p_ruler);
			universe->addEntity(ruler);
			ruler->freeze(true);
			ruler->updateNode();
		#endif

	// FRICTION BOX
		#if	frictionboxon
			frictionBox = new VxPart(0.001);
			double frictionBoxSize[3] = {.5,.5,.1};
			float frictionBoxcolor[4] = {1, 1, 0, 1};
			VxReal3 p_frictionBox = {0,2,.05};
			frictionBox->setName("frictionBox");  // give a name to the part to be able to
retrieve it later.
			frictionBox->addGeometry(new VxBox(frictionBoxSize), 0, 0);
			frictionBox->setNode(visualizer-
>createBox(frictionBoxSize,frictionBoxcolor,0,true,true)); // Comment this to hide it.
			frictionBox->setControl(VxEntity::kControlDynamic);
			frictionBox->setPosition(p_frictionBox);
			universe->addEntity(frictionBox);
			frictionBox->updateNode();
		#endif

	// PLANE
		VxReal3 p_plane = {0, 0, 0};
		VxPart* plane = new VxPart();
		plane->setName("plane");
```

```
		plane->addGeometry(new VxPlane(), 0); // Note: This actuall makes an entire plane.
		universe->addEntity(plane);
		plane->freeze(true);
		plane->setPosition(p_plane);
		#if visualizeron
			float planecolor[4] = {0, 0, 1, 0.9};
			VxReal3 planeSize = {10, 10, 0.001}; // For some reason this tries to find the
license when the Z thickness is 0.0001
			plane->setNode(visualizer->createBox(planeSize, planecolor, 0, false)); // Note:
although entire plane exists, only a box is shown.
		#endif

	// LINES
		#if visualizeron
			float linecolor[4] = {1, 1, .5, 1};
			VxVector3 from(-150, 0, 0.01); // X
			VxVector3 to(150, 0, 0.01); // X
			visualizer->createLine(from,  to , linecolor); // Y lines - These are not parts.
Only lines.
			//float linecolor[4] = {.2, .2, 1, 1};
			from.set(0, -1, 0.01); // Y
			to.set(0, 1, 0.01); // Y
			visualizer->createLine(from,  to , linecolor); // X lines
		#endif

	// declare the variables for use with the spring
	VxReal3 p_ball = {0,0,0};
	VxReal3 p_shape = {0,0,0};
	VxReal3 ang_shape = {0,0,0};
	VxReal3 p_ball_old, p_shape_old, springVector, springForce, springTorque, damping_force,
ang_shape_old;
	float ballLineColor[4] = {.2, 1, .2, 1};
	float shapeLineColor[4] = {1, .2, .2, 1};
	double ponew[3];
	double activeOffset[3] = {0,0,0};

	// Set the friction definitions
	VxContactProperties* cp = universe->getContactProperties(0, 0);
	cp->setFrictionType(VxContactProperties::kFrictionTypeTwoDirection);
	cp->setFrictionModel(VxContactProperties::kFrictionModelScaledBox);
	universe->setGravity(g);
	cp->setFrictionCoefficientPrimary(mu);
	cp->setFrictionCoefficientSecondary(mu);

	VxReal totalTime = 0;
	ang_shape_total = 0;
	double ang_rolled;
	int iteration = 0;
	double sumy = 0;

	do // Roll. Loops for each timestep
	{
		iteration++;
		#if visualizeron
			if (!visualizer->isPaused()) // if the visualizer is NOT paused...
			{
		#endif
				frame->step(); // Actually runs the simulation
				totalTime += frame->getTimeStep();
		#if visualizeron
			}
		#endif
		#if visualizeron
			running = visualizer->update();
		#endif
		//ball->getLinearVelocity(vel); // you need the linear velocity if you have a cutoff for
when the ball starts going slowly. see the while condition below for this loop.
		for (i=0; i<3; i++){
			p_ball_old[i]=p_ball[i];
			p_shape_old[i]=p_shape[i];
			ang_shape_old[i]=ang_shape[i];
```

```
		};
		ball->getPosition(p_ball); // gets the position of the ball
		shape->getPosition(p_shape); // gets position of the shape
		shape->getOrientationEulerXYZ(ang_shape); // initial angle.
		ang_rolled = abs(ang_shape[0]-ang_shape_old[0]);
		ang_shape_total=ang_shape_total+ang_rolled;
		for (i=0; i<3; i++) ponew[i] = offset[i]+p_shape[i]; // redefines the position of the
center of the shape
		#if visualizeron
				if (visualizeron) visualizer->createLine(p_ball_old, p_ball,
ballLineColor); // creates a line from the center of the frame to the center of the ball.
				if (visualizeron) visualizer->createLine(p_shape_old, p_shape,
shapeLineColor); // creates a line from the center of the frame to the center of the ball.
		#endif
		sumy+=p_shape[2];
		if (totalTime<1){
			activeOffset[0] = 0.03;
		};
		if (totalTime>1){
			activeOffset[0] = (r1*cos(omega*totalTime+theta)*cos(phi)-
r2*sin(omega*totalTime+theta)*sin(phi))*cos((double)pi/6);
			activeOffset[1] = activeOffset[0]*tan((double)pi/6);
			activeOffset[2] =
r1*cos(omega*totalTime+theta)*sin(phi)+r2*sin(omega*totalTime+theta)*cos(phi);
		};
		for (i=0; i<3; i++) {
			springVector[i] = p_ball[i]-ponew[i]-activeOffset[i]; // defibnes the difference
in location of the ball from where it should be.
			damping_force[i] = (p_ball_old[i]-p_ball[i])*0.1; // makes the springs slightly
hysteretic w/ damping
			springForce[i] = -springVector[i]*stiffness+damping_force[i]; // defines the force
that the spring exerts on the BALL
		};
		ball->addForce(springForce); // applies said force to the ball
		current_spring_force =
pow(springForce[0]*springForce[0]+springForce[1]*springForce[1]+springForce[2]*springForce[2],0.5);
		if (max_spring_force < current_spring_force)
			max_spring_force = current_spring_force;
		for (i=0; i<3; i++) springForce[i] *= -1; // redefines the force as that which is applied
to the FRAME
		springTorque[0] = offset[1]*springForce[2]-offset[2]*springForce[1]; // defines the
torque that acts on the shape due to the location of the ball within it not being the center of mass
		springTorque[1] = offset[2]*springForce[0]-offset[0]*springForce[2];
		springTorque[2] = offset[0]*springForce[1]-offset[1]*springForce[0];
		shape->addForce(springForce); // adds a force to the frame
		shape->addTorque(springTorque); // adds the torque to the frame
		#if frictionboxon
				frictionBox->addForce(frictionBoxForce); // adds the force to the
frictionBox to see if it moves - to determine the friction coeffiecient
		#endif
		#if visualizeron
				//visualizer->setCameraLookAtAndPosition(p_shape, camPos); // changes the
camera orientation to track the shape position.
		#endif
	}	while ((running) && totalTime < maxTime);// && vel.norm() > err_vel); // keep rolling
WHILE ...

	#if visualizeron
			visualizer->stopViewer();
			visualizer->startViewer();
	#endif
			double approx_perim = 6*l*pow(1+golden_ratio*golden_ratio,.5);
	double fit1 = p_shape[0]/(approx_perim); // the centroid translates in the x direction
	double fit2 = -abs(sumy/iteration-ro)/(ro-ri); // the centroid only slightly translates in the y
direction
	double fit3 = -abs(p_shape[0]-ang_shape_total*approx_perim)/1; // rolling check: (x displacement)
== perimeter*(theta displacement)
	fitness = fit1*cond1+fit2*cond2+fit3*cond3;

	// Shut down procedures
	#if visualizeron
```

```
                visualizer->stopViewer();
        #endif
        frame->release();
        #if visualizeron
                if (visualizeron) delete visualizer;
        #endif
        return fitness;
}
int main(int argc, const char *argv[])
{
        double fitness;
        fitness = mainnew(argc, argv);
        printf("the fitness is %f\n",fitness);
        int returnval = fitness*1000;
        return returnval;
}
```