FEBRUARY 28, 2020

BUILDING A BETTER IK SOLVER

USING BÉZIER CURVES TO SOLVE INVERSE KINEMATICS PROBLEMS

ARAGORN CROZIER
UNIVERSITY OF WASHINGTON

Contents

Introduction

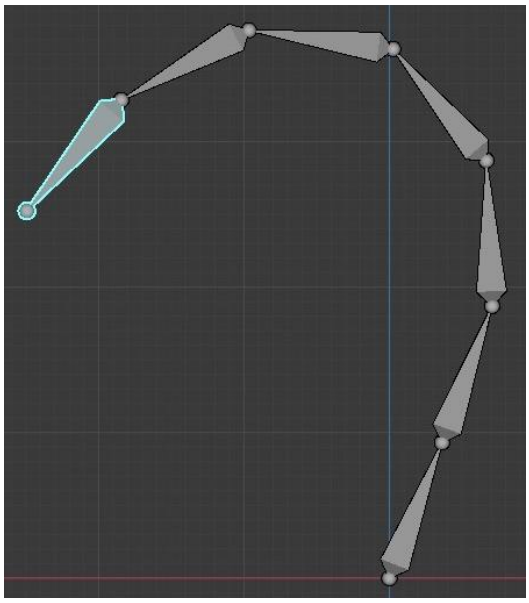

Figure 1 An example of an inverse kinematics (IK) chain (computed using auto IK)

An important part of 3D animation is the process of posing and animating characters. To make animating computer-generated characters easier, the artist creates an armature, or a skeletal structure, that has all the joints the character would need to be able to move around fluidly. Sometimes, this can result in long chains of bones that need to be calculated. This might occur in a leg or a giraffe neck, for example. To simplify the workflow with these long chains, instead of having to pose each bone in a chain, modern animation tools allow the artist to simply translate and rotate the last bone in the chain, and software adjusts all the bones leading up to fit smoothly.

The process of fitting all the bones to match the end one is called inverse kinematics (IK), and as the name implies, is simply the process of taking a final position, in this case the bone at the end of the chain, and computing the kinematics, or the rotations and translations, required to arrive at that point (see Figure 1). Blender, an open-source, popular animation software, includes a variety of IK methods ranging from basic (auto IK) to more advanced (iTaSC). Unfortunately, each of these methods has significant drawbacks. They can make it much more difficult to build armatures and animate characters when they produce undesirable artifacts and do not offer the control that animators need to fit their use cases.

A note on terminology: This paper is based on the IK solvers in Blender, which are only used on armatures. An armature is a linked set of rigid objects, called “bones” because they are often used to represent the motion that an animal’s bones would give them. The term “bone” will be used interchangeably with “rigid segment.”

IK Solvers

An IK solver in the context of this paper is an algorithm which finds the positioning of rigid elements in a chain based on the transformation of the final element. Figure 2 shows what such a chain might look like, where each octahedral prism is a rigid element, and the spheres are joints which allow the elements attached to them to rotate. In the simplest case, like this one, each bone can rotate freely on each joint.

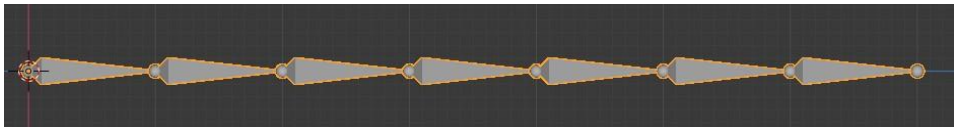

Figure 2 The original armature chain which the IK solver will operate on

The chain starts with the right-most bone, which is the parent of all the bones that come after it. A bone's root is the thicker end, while its tip is the thinner end. Each subsequent bone in the chain is the child of the bones that came before it, so it will inherit transformations down the chain.

Flaws with current IK Solvers

Current IK solvers are used on professional-grade projects but still have some significant flaws. The two broad problems are non-optimal solutions that end up producing sharp angles, and solutions that fail to account for manual rotations of the final bone.

Problem 1:

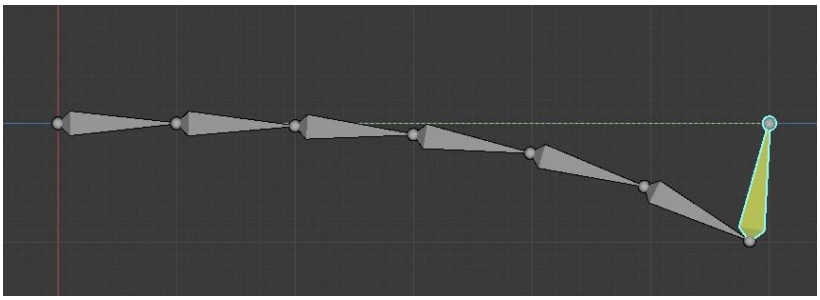

Figure 3 The chain with a standard IK solver applied to it

As seen in Figure 3, this IK solution has produced a very sharp angle between the final bone and its parent, which is unnatural. This IK constraint doesn't allow the tip of the final bone to move, which is an issue in of itself, but can be worked around. The flaws in the solution, however, are a much larger concern.

At first glance, this could be caused by the solver attempting to keep the length of each bone the same as they were originally, which would need to produce some irregularities when the final bone was rotated that far off the central axis. Essentially, the total length of the chain would be longer if it had to keep a smooth angle with the final bone, so each bone in the chain would need to be longer as well. But because the tip of the bone is fixed, it's impossible for the chain to remain at a constant length anyway.

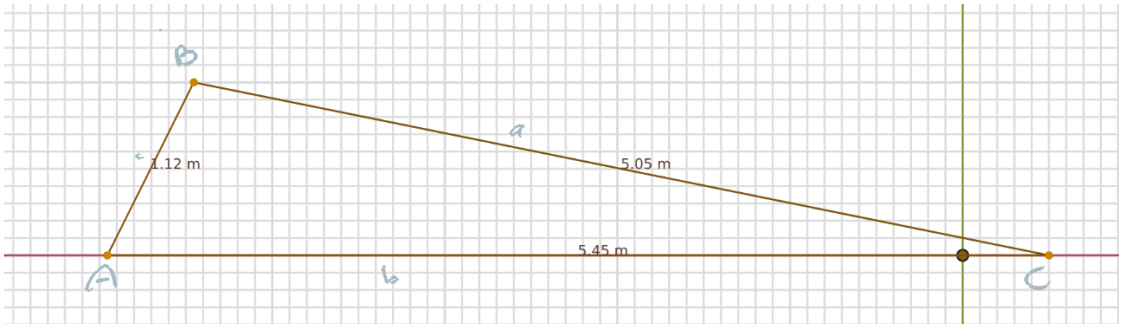

Figure 4 A triangle representing the lengths of chains needed. Side b is the original chain, c is the final bone, and a is the IK chain

Side a is the length of the chain that needs to be computed. Its length can be solved using the law of cosines, $a^2 = c^2 + b^2 - 2cb\cos A$. Since c and b are constant, $\angle A$ is the independent variable and a is the dependent variable. When the final bone is not rotated at all, $\angle A = 0$, in the starting condition, then:

$$a^2 = c^2 + b^2 - 2cb\,cos\,0$$

$$a^2 = c^2 - 2cb + b^2$$

$$a = \pm\sqrt{(c-b)^2}$$

$$a = c - b, b - c$$

Which makes sense because at this point, B lies on b so $a + c = b$ (choosing the second solution to a), as in Figure 5.

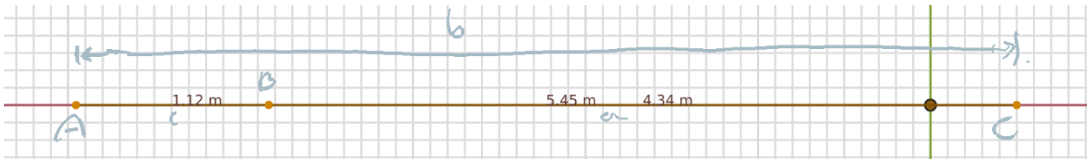

Figure 5 The triangle when A = 0, showing how $a = b - c$

When $A > 0$, $\cos A < 1$ so $-2cb \cos A > -2cb$:

$$a^2 > c^2 + b^2 - 2cb$$

$$\therefore a > \pm(c - b)$$

Using the same solution that was chosen for the equation above:

$$a + c > b$$

From this, we know that no matter how the final bone is rotated, the chain will need to grow longer to accommodate it. The sharp angle then, must be an error with the IK solver, which limits its applicability.

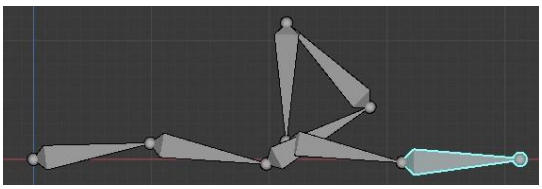

Figure 6 Errors produced by certain transformations by the auto IK solver

A similar issue also appears using the other IK solver, called “auto IK” in Blender, shown in Figure 6. In this case, the solver tries to curve the chain to the left, even when the bone is to the right of the start position. It creates further issues by limiting the range of rotation of the original bone, producing the problems seen above. The reasons this solution is not optimal is specifically because of the disproportionately tight angles it produces in the loop on the chain and because the solution does not create one smooth curve. The direction of the curvature between bones 1 and 2 and bones 2 and 3 are opposite, creating an unnecessarily chaotic chain.

Problem 2:

Auto IK does allow you to move the end of the bone and generally does not induce any sharp angles, but won't take rotation of the final bone (highlighted in blue) into consideration, as seen in Figure 7.

This is obviously not desirable because it fails to account for additional data that would have a major impact on the overall chain. It results in a sharp angle between the chain and the final bone which is clearly not optimal.

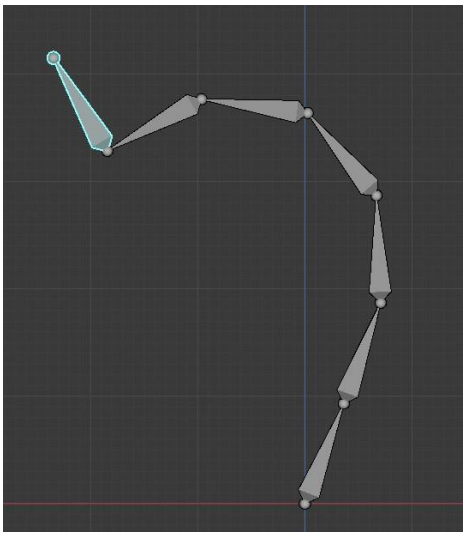

Figure 7 An auto IK-solved chain with the final bone manually rotated

The iTaSC solver does allow for individual bones to be rotated, slightly. Unfortunately, it does this through setting individual constraints on each bone and forces the user to specify a rotation range on the final bone rather than a specific angle. This overly involved process should be faster when performing such a simple task.

By analyzing these requirements, I have a better sense of what my IK solver needs to do:

1. Take account manual rotation of the final bone
2. Create a visually smooth curve (which the current solutions generally do)
3. Be easy for the artist to use

How IK Solvers Work

There are two main types of IK solvers: Those based on splines (or some other kind of mathematical curve) and those using a system called cyclic coordinate descent (CCD).

The curve-based solvers essentially design a mathematical curve to fit the constraints of the chain, then overlay the bones on the curve. This works well for finding an optimal solution for the chain

Meanwhile, CCD is a greedy algorithm that can easily solve for individual joint constraints. A greedy algorithm simply makes the biggest improvement it can at each iteration,

and can also be thought of as an “instant gratification” algorithm, instead of an algorithm that may attempt to make small, gradual improvements or takes into account the overall problem to make strategic modifications that are beneficial in the long run. CCD uses an error function, which is the distance of the tip of the chain from the target, and iterative bone solving to reach a solution. The algorithm begins with the parent bone and finds the individual rotation that minimizes the error function the most, hence making it greedy. When positioning each bone, it is easy for it to restrict the movement to provided constraints but the whole system has some drawbacks.

First, it is called a “blind hill-climbing algorithm” [1] because it is unable to see the overall landscape of the chain. This means it can get stuck if child bones have constraints that prevent them from making the final reach to the target. This can be resolved using a modified path finding algorithm that can trace back and modify earlier bones to accommodate the constraints, or simply try a few random starting positions, assuming that one of them would work (which is how production IK solvers work) [1]. It also might not find an optimal solution as far as smoothness is concerned if it finds a rougher solution first. But more importantly, there is no way for such a system to account for manual rotation of the final bone because by the time it begins positioning the second-to-final bone, its possible solutions are limited by the rest of the chain and it probably won’t be able to make a smooth transition to the final bone. This method is a deal-breaker then, because accounting for manual rotations was one of the original specifications outlined above.

The spline-based solvers appear to be the ones primarily used by Blender. The curves that Blender’s IK systems produce clearly consider the entire chain, indicating that it likely defined an overall curve first before fitting the bones to it. This can be seen in Figure 6 and Figure 7 where several bones make smooth curves, and many don’t point directly towards the tip of the chain, which one would expect if each bone was individually trying to get as close to the target as possible.

New IK Solver

A spline solver is then the best method for considering final bone rotation. For the purposes of making this easier to read, all the numbers have been rounded to 3 decimal places which may result in small discrepancies. However, all calculations have been performed using numbers that often go out to 10 decimal places or more so the inconsistencies will be very minor and do not compound on each other.

To start, I defined the necessary parameters for creating my spline, which can be visualized in Figure 8. Point O is the root of the whole chain and is the location of vector $\vec{u}$, which represents the first bone. Point P is the root of the final bone and is where vector $\vec{v}$ is located. P is set by the user when they move the final bone and is the target of the IK chain. O is an arbitrary location but does not change, so to simplify the problem, $O = (0,0,0)$ and P is moved by an equivalent amount, leaving their relative positions unchanged. So, the IK solver needs to form a spline from O to P with an approximate length of l (the length of the chain).

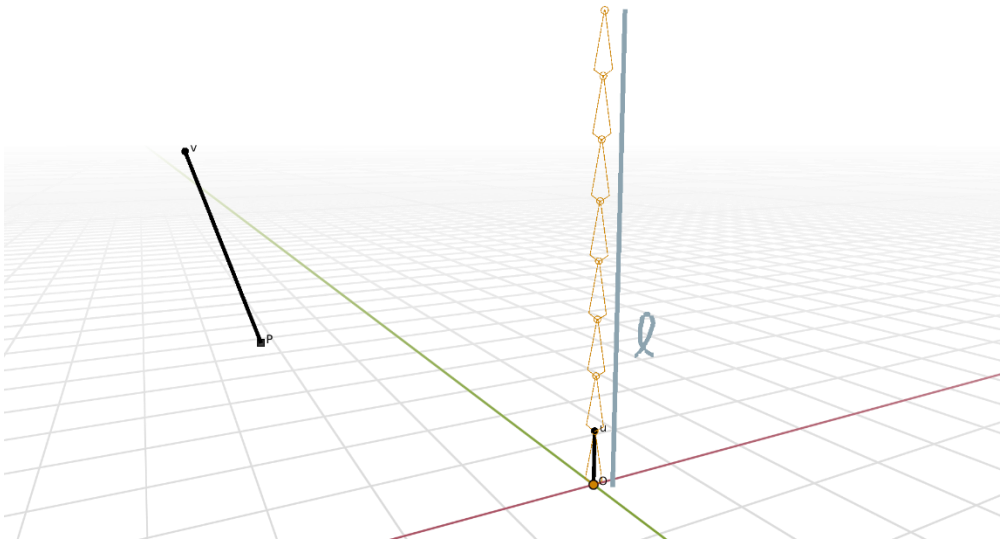

Figure 8 A diagram showing visualizations of the parameters used to define the chain, with the chain shown by the orange outline (where $u = \vec{u}$ and $v = \vec{v}$)

From O, $\vec{u}$ represents the first bone and $\vec{v}$ represents the final bone. In order to make the solver more generally applicable, the influence of the rotation of the final bone should depend on the length of the bone, which is represented by the length of $\vec{v}$, or $|\vec{v}|$. This means

that one could make a final bone with zero length and one would get behavior like current solvers where the rotation of the final bone did not matter.

$\vec{u}$ is necessary to control how much the start joint rotated. Without considering the original rotation of the bone, the solver would freely rotate it, which would likely not be desirable. Most areas where inverse kinematics is used are for animating characters' limbs, and it would be a problem if a character reaching across their body instead had their arm travel through their chest because it ignored that the normal direction of the upper arm is away from the body. This again is dependent on the length of the initial bone so it can adjusted as the animator requires. The solver would need to balance reducing the curvature of the chain, which it could probably do by pointing the first bone more directly, with smoothing the joints between the chain and the first and last bones, which would typically require a longer curve. This can be optimized using numerical root-finding functions, and some fast ones are already included in the SciPy libraries for Python.

Figure 9 shows a potential IK chain with sketched x and y components. It shows that a cubic piecewise spline is not an option because a solution would either require more control points or a higher order polynomial. A piecewise spline also does not provide any simple way to adjust the length of the result to fit the length of the chain.

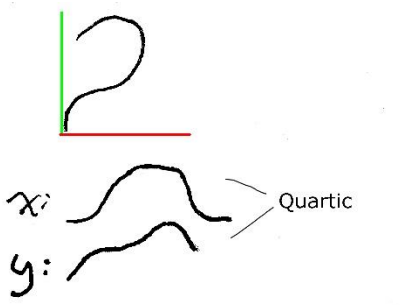

Figure 9 A possible optimal IK chain

Instead of solving based on 2 static control points, a better solution could address both problems at once by approximating the curve instead of interpolating it. An example of how this could look is in Figure 10, where the curve can be

modified by moving the C_1 and C_2 control points. Moving the C_2 control point would be an easy way to adjust the influence of the rotation of the final bone and by approximating there are more degrees of freedom for the solver to adjust based on other constraints.

For this approximation, we use a b-spline curve with C_0, C_1, C_2, and C_3as the control points based on existing approximation methods [3]. With only 4 control points, we will use a first-order b-spline, which happens to be a Bézier curve.

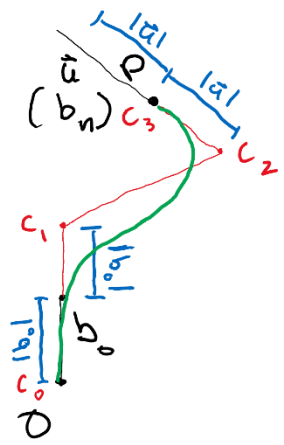

Figure 10 An example of an approximated spline (green) based on 4 control points (red)

Bézier Curves

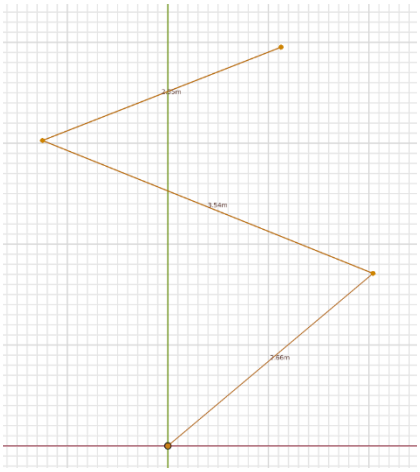

Figure 11 An example of 4 points that would constrain a Bézier curve

A Bézier curve can be thought of as a function that smoothly adjusts the weights of a series of linear functions. As opposed to a piecewise function, like the cubic splines above, Bézier curves have no hard transitions between functions like piecewise functions do at their knots.

Like in Figure 11, as the function advances from its start, at 0, to the end of approximation, the weight of each of the straight-line segments, or polynomials of degree 1, gets smoothly adjusted. This can also be seen in the equation form of a Bézier curve.

$$B(t) = \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i} t^i C_i \;\; t \in [0,1] \tag{1}$$

Where $B(t)$ is the curve function, n is the order of the function, and C_i is the i^{th} control point. The order of the function will always be one less than the number of points that are being approximated [4].

For this case, because there are 4 points, the order of the Bézier curve will be 3. As such, it can be expanded like so:

$$B(t) = \binom{3}{0}(1-t)^3 C_0 + \binom{3}{1}(1-t)^2 t C_1 + \binom{3}{2}(1-t)t^2 C_2 + \binom{3}{3}t^3 C_3$$

$$B(t) = (1-t)^3 C_0 + 3(1-t)^2 t C_1 + 3(1-t)t^2 C_2 + t^3 C_3$$

As t progresses from 0 to 1, the strength of the early terms reduces. $(1-t)^3$ quickly approaches 0 as t approaches 1, and t^3 quickly goes from 0 to 1. Because of this, it can be thought of that each point "pulls" at the curve.

The next challenge is defining the intermediate points C_1 and C_2. For this, the tip of each bone is represented with a three-dimensional position vector, in the form

$$\boldsymbol{r} = \boldsymbol{a} + \lambda \boldsymbol{b}$$

Where a is $P = C_3$ or $O = C_0$, b is a three-dimensional vector representing the corresponding bone, and λ is a scalar used to position the intermediate control points and is equal to 1 to represent the bone.

C_1 will be located at $C_1 = O + 2b$, placing it two bone-lengths away from the origin. C_2 will be placed similarly, at $C_2 = P - 2.2b$. This is because both the origin, O, and the endpoint, P, are the roots of bones, so in order to place C_2 in between C_3 and C_0 it needs a negative coefficient.

Length of a Curve

To calculate the length of an arbitrary curve, we start with the distance formula and the idea of derivatives from first principles:

$$d = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$$

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Combining these two concepts results in a formula for the distance between two arbitrarily close points on a function:

$$\lim_{h\to 0}\sqrt{\big(f(x+h)-f(x)\big)^2+h^2}$$

Summing all the distances in a range from a to b, and expressing the above equation in terms of a, we get the length l from a to b:

$$\mathrm{l(a,b,f(x))}=\lim_{\mathrm{h}\to 0}\sum_{\mathrm{k}=0}^{\frac{\mathrm{b-a}}{\mathrm{h}}-1}\sqrt{\big(\mathrm{f(a+kh+h)-f(a+kh)}\big)^2+\mathrm{h}^2}=\int_{\mathrm{a}}^{\mathrm{b}}\sqrt{1+\left(\frac{\mathrm{dy}}{\mathrm{dx}}\right)^2}\,\mathrm{dx} \qquad (2)$$

The proof of this equality can be found in Appendix I.

Minimizing Error

In order to reconcile the difference between the length of the curve and the length of the bone chain, we can express it as a root-finding function, which can be quickly solved by a computer. In a more mathematical sense, we can write

$$E=l\big(0,1,B(t)\big)-l_{chain}=0$$

We then need a parametric form of the length function in order to calculate the length in three dimensions. In other words, equation 3 needs to be expressed as an integral and derivative in terms of t:

$$\int_0^1\sqrt{\left(\frac{dx}{dt}\right)^2+\left(\frac{dy}{dt}\right)^2+\left(\frac{dz}{dt}\right)^2}\,dt$$

Solving for the derivative of the Bézier curve:

$$\frac{d\big(B(t)\big)}{dt}=-3C_{0s}+6tC_{0s}-3t^2C_{0s}+3(C_{0s}+\lambda b_{0s})-4t(C_{0s}+\lambda b_{0s})+3t^2(C_{0s}+\lambda b_{0s})$$
$$+6t(C_{3s}-1.5\lambda b_{ns})-9t^2(C_{3s}-1.1\lambda b_{ns})+3t^2C_{3s}$$

where $s\in\{x,y,z\}$ is the dimension, b_0 is the first bone, and b_n is the last.

As such, the error function E can be written in terms of λ:

$$E(\lambda)=\int_0^1\sqrt{\left(\frac{dx}{dt}(\lambda)\right)^2+\left(\frac{dy}{dt}(\lambda)\right)^2+\left(\frac{dz}{dt}(\lambda)\right)^2}\,dt-l_{chain}=0 \qquad (3)$$

Modifying λ moves the control points, C_1 and C_2, closer or farther away from their respective endpoints.

Computing the roots of this equation by hand would be nearly impossible, but algorithmic root-finding functions can numerically determine the roots and optimize λ so that the chain length and the function length are the same.

Therefore, the final Bézier curve will be

$$(1-t)^3 C_0 + 3(1-t)^2 t(C_0 + \lambda b_0) + 3(1-t)t^2(C_3 - 1.1\lambda b_n) + t^3 C_3$$

where λ satisfies equation 3. The coefficient on λb_n can be modified as necessary for each use case.

Example

To demonstrate, we can apply this to the example from Figure 8.

In it, we have control points $C_0 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$, $b_0 = \begin{pmatrix} 0 \\ 0 \\ 0.57 \end{pmatrix}$, $C_3 = \begin{pmatrix} -3.448 \\ -2.697 \\ 2.657 \end{pmatrix}$, and $b_n = \begin{pmatrix} -0.227 \\ -0.313 \\ 0.715 \end{pmatrix}$. The chain consists of 11 bones, each with a length of 0.57 units, making a total chain length of 6.27. However, since the final bone is fixed, the total length that is being solved for is just 5.7 units.

We position C_1 and C_2 such that $C_1 = C_0 + \lambda b_0$ and $C_2 = C_3 - 1.1\lambda b_n$, making the weights between the first and last bone fairly even.

Starting with $\lambda = 2$, the Bézier curve produced is:

$$B(t) = \begin{cases} x(t) & = 8.846(1-t)t^2 - 3.448t^3 \\ y(t) & = 6.029(1-t)t^2 - 2.697t^3 \\ z(t) & = 3.42(1-t)^2 t + 3.250(1-t)t^2 + 2.657t^3 \end{cases}$$

Which looks very good:

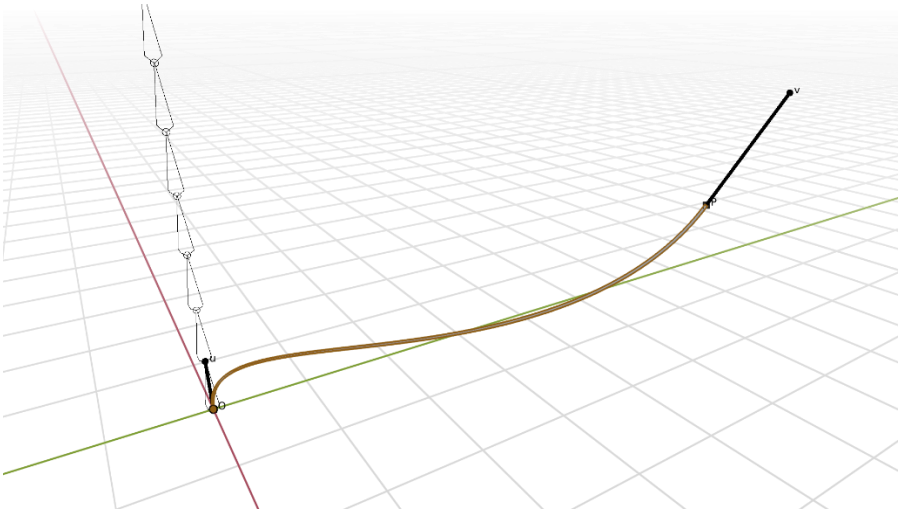

Figure 12 The Bézier curve which fits the original example parameters, in orange and gray

The next step is adjusting the length, which can be done with the equation above and a numerical root finding method. Using SciPy's integrate function, we find that the length of the curve above to be about 6.7161 units. This is slightly longer than the length of the chain, meaning that it needs to be shortened somewhat.

We can use SciPy's scalar root-finding function to determine a value of λ that satisfies the error equation (3). It is good practice to limit λ to values greater than 0 to prevent the chain leaving the bones in the opposite direction. This is unlikely in most cases but could happen if the end bones curve back on each other. Because the difference between C_3 and C_0 is close to the length of the chain, $\lambda = 0.801$. If the distance between the final point and the origin is longer than the chain, the root-finding algorithm will fail to converge, ending the Bézier IK solver at that point.

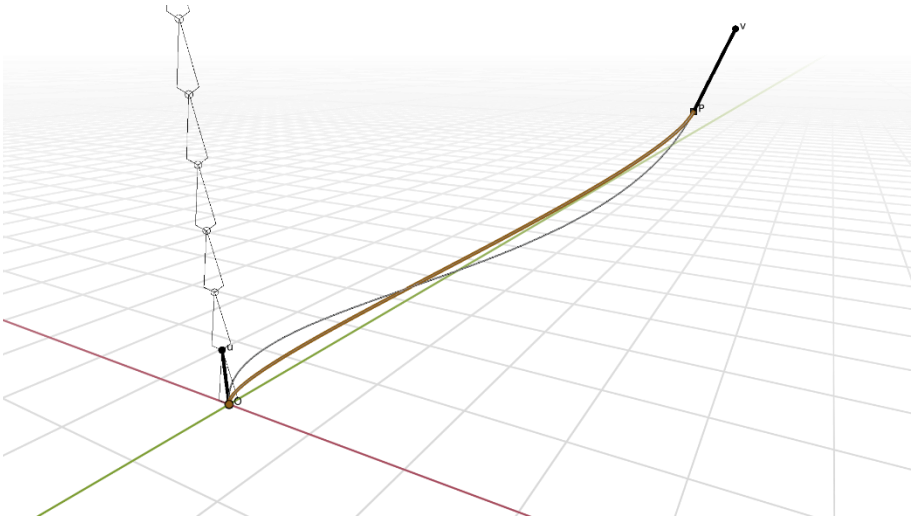

Figure 13 The shortened Bézier curve (orange and gray) alongside the original curve (gray)

Figure 13 shows that the method of curve solving correctly balances between more even curves and fitting the length of the chain. For reference, the new curve equations are shown below:

$$B(t) = \begin{cases} x(t) = & -9.745(1-t)t^2 - 3.448t^3 \\ y(t) = & -7.266(1-t)t^2 - 2.697t^3 \\ z(t) = & 1.369(1-t)^2t + 6.081(1-t)t^2 + 2.657t^3 \end{cases} \quad (4)$$

Placing Bones

The process of placing the bones on the curve is tedious but not mathematically challenging. The simplest method is to start by representing each bone as a sphere with a center at the tip of the previous bone and calculating the next tip by finding the intersection between the curve and the sphere. A bone can be represented like so:

$$b_i = \left(x - b_{i-1,x}\right)^2 + \left(y - b_{i-1,y}\right)^2 + \left(z - b_{i-1,z}\right)^2 = |b_i|^2$$

where b_i is the tip of the i^{th} bone and $|b_i|$ is the length of the i^{th} bone. This defines a sphere in $\mathbb{R}^3$ with a center at the tip of the $(i-1)^{\text{th}}$ bone and a radius of the length of the i^{th} bone.

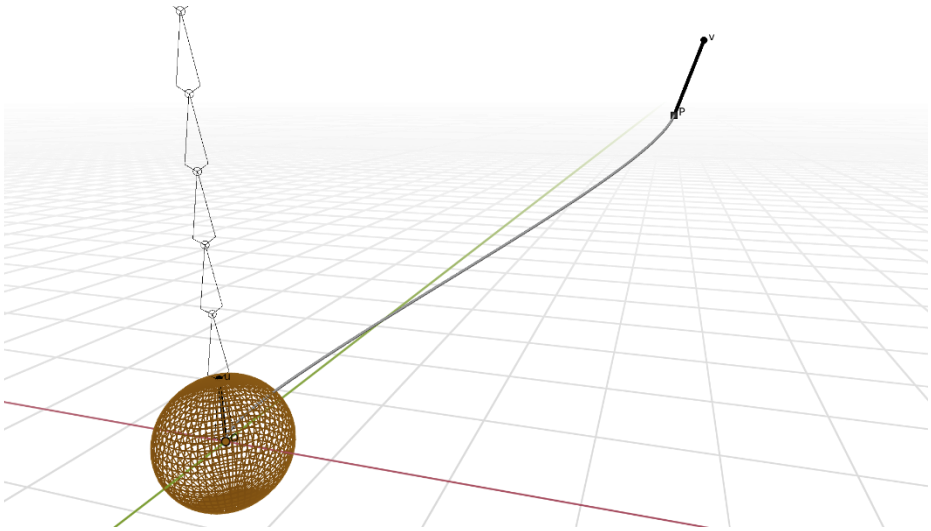

Figure 14 A sphere with a radius of the first bone (orange) showing the potential locations of the tip of the bone

We can substitute the original Bézier curves into the above equation, providing a recursive equation for each tip:

$$\big(x(t) - b_{i-1,x}\big)^2 + \big(y(t) - b_{i-1,y}\big)^2 + \big(z(t) - b_{i-1,z}\big)^2 - |b_1|^2 = 0$$

Which makes it another simple root-finding problem. For the first bone, the center of the sphere will simply be at the origin, so the equation becomes:

$$x(t)^2 + y(t)^2 + z(t)^2 - 0.57^2 = 0$$

And $x(t)$, $y(t)$, $z(t)$ come from equation 5 above. Using the same numerical root-finding method as before, the value of t that fits the equation is 0.198 so the coordinates of the tip of the bone as a position vector are approximately $\begin{pmatrix} -0.335 \\ -0.251 \\ 0.387 \end{pmatrix}$. We can verify that this is the correct location by using the distance formula between that vector and the origin to check that it is the same length as the bone:

$$\sqrt{(-0.335)^2 + (-0.251)^2 + 0.387^2} = 0.57$$

This can then be used as the center of the next bone's sphere. In order to make the optimization faster, the domain of $B(t)$ can begin at the value of t that the first bone lies on, so $t \in [0.198, 1]$. This also prevents the root-finding method from finding the wrong intersection

of the sphere with the curve (since the sphere will intersect the curve in the direction of the next bone and the direction of the previous bone). The only remaining issue is that the length of the bones on the curve will not be the same as the length of the curve itself. This can be resolved using the chain length adjustment from earlier, but shouldn't be used as the primary method on low-power systems or when computation is at a premium since the bone placement requires finding a root for every bone each time the overall length is adjusted, which is more computationally expensive. In these cases, the tolerance can be increased, requiring fewer iterations of the root-finding system. The difference between the bones on the curve and the curve itself is very small except in conditions where the bones are very long and the curve has a large amount of curvature (which can be visualized in Figure 15).

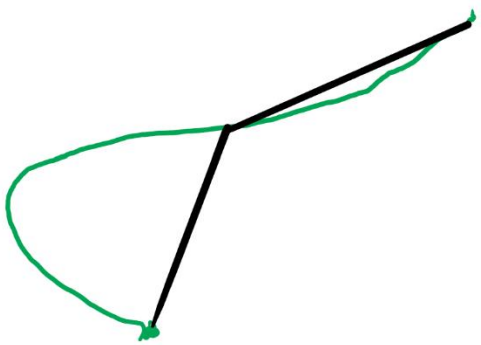

Figure 15 An example of a Bézier curve with large curvature and long bones compared to the length of the curve

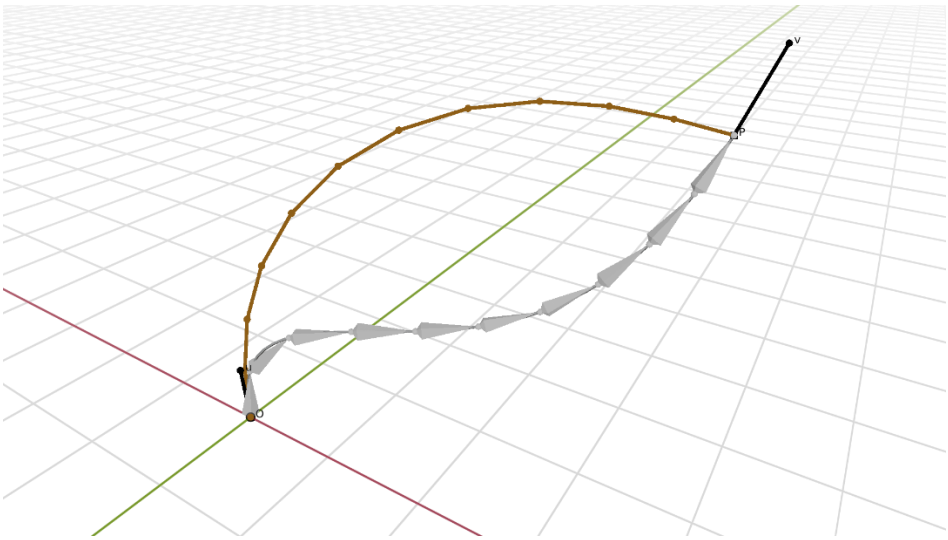

Figure 16 My final IK chain (gray) alongside the auto-IK chain (orange)

Finally, in Figure 16, one can see the final product that the new IK method produced. While it does produce tighter angles (such as near the root of the chain), it is able to consider the rotation of the final bone which the built-in method does not do. The built-in method creates a nearly 90-degree angle between the user-positioned bone, v, and the computed

chain. The Bézier chain, on the other hand, never has any angles anywhere near 90 degrees, making it superior. The smooth arc that auto-IK makes may be desirable in some situations but is useless if the artist wants to position the last bone themselves.

Conclusion

Overall, this IK solver meets the criteria for improving on the existing solvers that are available in Blender. This solver accounts for manual rotation of the final bone and tends to produce visually pleasing curves. In the creation of the example, it was found that computation times were quick, even when being performed on a laptop using line-interpreted code (as opposed to compiled code, like Fortran or C, which runs significantly faster).

The primary drawback is that introducing individual joint constraints with this method would be difficult. However, once that is an issue, it is not significantly more difficult for the artist to also constrain the final bone and use the iTaSC solver. While it is possible that with further research and tweaking of the model one could force a Bézier curve to obey those joint constraints, it will probably be easier, if there are many highly constrained bones, to simply use a CCD or iTaSC algorithm, as described earlier.

References

1. Blow, Jonathan, *Inverse Kinematics with Joint Limits*, Game Developer Magazine (2002) pp. 16-18.

2. House, Donald H, *Spline Curves*, Clemson University (2015) pp. 87-103.

3. Ezhov, Nikolaj and Neitzel, Frank, *Spline Approximation, Part 1: Basic Methodology*, Journal of Applied Geodesy (2018).

4. Editors of Wikipedia, *Bézier Curve*, Wikipedia, accessed 22 February 2020.

Appendices

Appendix I: Length of a Curve Proof

$$l = \lim_{h\to 0} \sum_{k=0}^{\frac{x_n - x_0}{h} - 1} \sqrt{\left(f(x_0 + kh + h) - f(x_0 + kh)\right)^2 + h^2}$$

$$l = \lim_{h\to 0} \sum_{k=0}^{\frac{x_n - x_0}{h} - 1} \sqrt{\left(f(x_0 + kh + h) - f(x_0 + kh)\right)^2 \cdot \frac{h^2}{h^2} + h^2}$$

Using $x = x_0 + kh$, $a = x_0$, and $b = x_n$

$$l = \lim_{h\to 0} \sum_{k=0}^{\frac{b-a}{h} - 1} \sqrt{\left(\frac{f(x+h) - f(x)}{h}\right)^2 h^2 + h^2}$$

$$l = \lim_{h\to 0} \sum_{k=0}^{\frac{b-a}{h} - 1} \sqrt{\left(\left(\frac{f(x+h) - f(x)}{h}\right)^2 + 1\right) h^2}$$

$$l = \lim_{h\to 0} \sum_{k=0}^{\frac{b-a}{h} - 1} \sqrt{1 + \left(\frac{dy}{dx}\right)^2}\, h$$

$$\because \int_a^b f(x)\, dx = \lim_{w\to 0} \sum_{i=0}^{\frac{b-a}{w} - 1} f(a + wi)\, w$$

Where $x = a + iw$ and $f(x) = \sqrt{1 + \left(\frac{dy}{dx}\right)^2}$, then:

$$l = \int_a^b \sqrt{1 + \left(\frac{dy}{dx}\right)^2}\, dx$$

To parameterize the function:

$$l = \int_a^b \sqrt{1 + \frac{(dy)^2}{(dx)^2} \cdot \frac{\left(\frac{1}{dt}\right)^2}{\left(\frac{1}{dt}\right)^2}}\, dx$$

$$l = \int_a^b \sqrt{1 + \frac{\left(\frac{dy}{dt}\right)^2}{\left(\frac{dx}{dt}\right)^2}}\, dx$$

$$l = \int_a^b \sqrt{1 + \frac{\left(\frac{dy}{dt}\right)^2}{\left(\frac{dx}{dt}\right)^2}}\, dx \cdot \frac{\frac{dx}{dt}}{\frac{dx}{dt}}$$

$$l = \int_a^b \sqrt{\left(\frac{dx}{dt}\right)^2 + \frac{\left(\frac{dy}{dt}\right)^2 \cdot \left(\frac{dx}{dt}\right)^2}{\left(\frac{dx}{dt}\right)^2}}\, dx \cdot \frac{1}{\frac{dx}{dt}}$$

$$l = \int_a^b \sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2}\, dx \cdot \frac{dt}{dx}$$

Generalizing to three dimensions results in:

$$l = \int_a^b \sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2 + \left(\frac{dz}{dt}\right)^2}\, dt$$

Appendix II: 3D Curve Graphing Code

add_curve_3d_function_curve.py:[1]

```
#Configuration info for Blender
bl_info = {
    "name" : "Math Curve",
    "author" : "Aracro Products",
```

[1] I published this code to be included in the official Blender release and it is protected under the GNU General Public License

```
    "version": (1, 0, 0),
    "blender" : (2, 80, 75),
    "location" : "View3D > Add > Curve",
    "description" : "Generate a curve according to three functions",
    "warning" : "",
    "category" : "Add Curve",
    }

import bpy
from math import *
from bpy.types import Operator
from bpy.props import (
        StringProperty,
        IntProperty,
        FloatProperty,
        BoolProperty,
        )

#Functions that can be used by eval() when generating the curve
safe_list = ['math', 'acos', 'asin', 'atan', 'atan2', 'ceil', 'cos', 'cosh',
    'degrees', 'e', 'exp', 'fabs', 'floor', 'fmod', 'frexp', 'hypot',
    'ldexp', 'log', 'log10', 'modf', 'pi', 'pow', 'radians',
    'sin', 'sinh', 'sqrt', 'tan', 'tanh']

# Use the list to filter the local namespace
safe_dict = dict((k, globals().get(k, None)) for k in safe_list)
#puts the value of the global variable into safe_dict value corresponding to
the
#name from the safe_list

def create_curve_object(context, verts, name, bevel_depth=0.1, resolution=2):

    curve = bpy.data.curves.new(name, type='CURVE')

    poly_line = curve.splines.new('POLY')
    poly_line.points.add(len(verts) - 1)
    for i, coord in enumerate(verts):
        x,y,z = coord
        poly_line.points[i].co = (x, y, z, 1)

    curve.dimensions = '3D'
    curve.bevel_depth = bevel_depth
    curve.fill_mode = 'FULL'
    curve.resolution_u = resolution

    #newCurve = bpy.data.objects.new("Math_Curve", curve)
    #bpy.context.collection.objects.link(newCurve)

    from bpy_extras import object_utils
    return object_utils.object_data_add(context, curve, operator=None)

def xyz_function_curve_verts(self, x_eq, y_eq, z_eq,
            range_t_min, range_t_max, range_t_step):

    verts = []

    # Distance of each step in Blender Units
```

```
    tStep = (range_t_max - range_t_min) / range_t_step

    # Number of steps in the vertex creation loops.
    # Number of steps is the number of faces
    #   => Number of points is +1 unless wrapped.
    tRange = range_t_step + 1



    try:
        expr_args_x = (
            compile(x_eq, __file__.replace(".py", "_x.py"), 'eval'),
            {"__builtins__": None},
            safe_dict)
        expr_args_y = (
            compile(y_eq, __file__.replace(".py", "_y.py"), 'eval'),
            {"__builtins__": None},
            safe_dict)
        expr_args_z = (
            compile(z_eq, __file__.replace(".py", "_z.py"), 'eval'),
            {"__builtins__": None},
            safe_dict)

    except:
        import traceback
        self.report({'WARNING'}, "Error parsing expression(s) - "
                    "Check the console for more info")
        print("\n[Add X, Y, Z Function Surface]:\n\n",
traceback.format_exc(limit=1))
        return []

    for tN in range(tRange):
        t = range_t_min + (tN * tStep)


        safe_dict['t'] = t

        # Try to evaluate the equations.
        try:
            verts.append((
                float(eval(*expr_args_x)),
                float(eval(*expr_args_y)),
                float(eval(*expr_args_z))))
        except:
            import traceback
            self.report({'WARNING'}, "Error evaluating expression(s) - "
                                                    "Check the console for more
info")
            print("\n[Add X, Y, Z Function Surface]:\n\n",
traceback.format_exc(limit=1))
            #return []

    return verts


#Operator class that shows up in Blender interface
class AddXYZFunctionCurve(Operator):
```

```
bl_idname = "curve.primitive_xyz_function_curve"
bl_label = "Add X, Y, Z Function Curve"
bl_description = ("Add a curve defined by 3 functions:\n"
                  "x=F1(t), y=F2(t), z=F2(t)")
bl_options = {'REGISTER', 'UNDO', 'PRESET'}

x_eq: StringProperty(
    name="X equation",
    description="Equation for x=F(t).",
    default="cos(t)"
    )
y_eq: StringProperty(
    name="Y equation",
    description="Equation for y=F(t).",
    default="sin(t)"
    )
z_eq: StringProperty(
    name="Z equation",
    description="Equation for z=F(t).",
    default="t"
    )
range_t_min: FloatProperty(
    name="T min",
    description="Minimum T value, lower value of range",
    min=-100.00,
    max=0.00,
    default=0.00
    )
range_t_max: FloatProperty(
    name="T max",
    description="Maximum T value, upper value of range",
    min=0.00,
    max=100.00,
    default=2*pi
    )
range_t_step: IntProperty(
    name="T step",
    description="T subdivisions",
    min=1,
    max=1024,
    default=32
    )
range_depth: FloatProperty(
    name="Bevel depth",
    description="Bevel depth of the generated curve",
    default=0.1
    )
range_res: IntProperty(
    name="Resolution",
    description="Bevel resolution of curve",
    default=2,
    min=1
    )

#code that runs after changes have been made to parameters
def execute(self, context):
    verts = xyz_function_curve_verts(
```

```
            self,
            self.x_eq,
            self.y_eq,
            self.z_eq,
            self.range_t_min,
            self.range_t_max,
            self.range_t_step
            )
        if not verts:
            return {'CANCELLED'}

        obj = create_curve_object(context, verts, "XYZ Function", bevel_depth
= self.range_depth, resolution = self.range_res)

        return {'FINISHED'}

#Describes where the operator should appear in the Blender interface
def menu_func(self, context):
    bl_label = 'Change'

    obj = context.object
    layout = self.layout

    layout.operator("curve.primitive_xyz_function_curve",
                    text="Math Curve", icon="CURVE_DATA")

classes = [
    AddXYZFunctionCurve
    ]

#Code that runs when add-on is enabled
def register():
    for cls in classes:
        bpy.utils.register_class(cls)
    #adds the function to the add_curve menu
    bpy.types.VIEW3D_MT_curve_add.append(menu_func)

#Code that runs when add-on is disabled
def unregister():
    bpy.types.VIEW3D_MT_curve_add.remove(menu_func)

    from bpy.utils import unregister_class
    for cls in reversed(classes):
        bpy.utils.unregister_class(cls)

if __name__ == "__main__":
    register()
```

Appendix III: Code for Constructing a Bézier Curve

IK.py:

```
import three_vector
import bezier as b
```

```
import scipy.integrate as integrate
import scipy.optimize as optimize

#The initial conditions for the solver to work on
c0 = [0, 0, 0]
b0 = [0, 0, 0.57]
c3 = [-3.44839, -2.69727, 2.65739]
bn = [-0.227075, -0.312513, 0.715447]
chain_length = 5.7

#   Returns the value of the bezier curve defined by the control points,
                    first and last bone, and mue at position t
#   PARAMETERS:
#   t   - (float) the parameter for the bezier curve 0 <= t <= 1
#   c0  - (float list) a 3 element array describing the location of the first
                    control point [x, y, z]
#   b0  - (float list) a 3 element array representing the vector of the first
                    bone [x, y, z]
#   c3  - (float list) a 3 element array describing the location of the last
                    control point [x, y, z]
#   bn  - (float list) a 3 element array representing the vector of the last
                    bone [x, y, z]
#   mue - (float) the scalar that adjusts the location of the middle control
                    points
#   RETURNS:
#   a 3 element list with the x, y, and z values of the bezier curve at t
                    respectively
def bt(t, c0, b0, c3, bn, mue):
    c1 = b.get_control(c0, b0, mue)
    c2 = b.get_control(c3, bn, mue * -1.1)
    curve = []
    for x in range(0, 3):
        curve.append((1-t)**3 * c0[x] + 3 * (1-t)**2 * t * c1[x] + 3 * (1-t)
                    * t**2 * c2[x] + t**3 * c3[x])
    return curve

#   Returns the value of the first-order partial derivative of the bezier
                    curve for dimension s with control points
#   defined by mue at the given value of t
#   Note: requires that c0, b0, c3, and bn are already defined in the scope
                    of execution
#   PARAMETERS:
#   s   - (int) the dimension of the equation (0 is x, 1 is y, and 2 is z)
#   t   - (float) the parameter for the bezier curve 0 <= t <= 1
#   mue - (float) the scalar that adjusts the location of the middle control
                    points
#   RETURNS:
#   float representing the value of the deriverative for the given dimension
                    at the given value of t
def derivbs(s, t, mue):
    c1 = b.get_control(c0, b0, mue)
    c2 = b.get_control(c3, bn, mue * -1.1)
    return (-3 + 6*t - 3*t**2)*c0[s] + (3 - 4*t + 3*t**2)*c1[s] + (6*t -
                    9*t**2)*c2[s] + 3*t**2*c3[s]

#   Prints the equations for the bezier curve defined by the parameters in a
                    Python-interpretable format
```

```
#   PARAMETERS:
#   c0  - (float list) a 3 element array describing the location of the first
                    control point [x, y, z]
#   b0  - (float list) a 3 element array representing the vector of the first
                    bone [x, y, z]
#   c3  - (float list) a 3 element array describing the location of the last
                    control point [x, y, z]
#   bn  - (float list) a 3 element array representing the vector of the last
                    bone [x, y, z]
#   mue - (float) the scalar that adjusts the location of the middle control
                    points
def bezier(c0, b0, c3, bn, mue):
    c1 = b.get_control(c0, b0, mue)
    c2 = b.get_control(c3, bn, mue * -1.1)
    print(b.make_third_degree_bezier(c0, c1, c2, c3))

#   Sums the squares of each partial derivative
#   PARAMETERS:
#   t   - (float) the parameter for the bezier curve 0 <= t <= 1
#   mue - (float) the scalar that adjusts the location of the middle control
                    points
#   RETURNS:
#   (float) The sum of each partial deriverative of the bezier function based
                    on mue squared
def integrand_function(t, mue):
    sum_deriv = 0
    for s in range(0, 3):
        sum_deriv += derivbs(s, t, mue)**2
    return sum_deriv**0.5

#   Integrates the integrand_function from 0 to 1 based on mue
#   PARAMETERS:
#   mue - (float) the scalar that adjusts the location of the middle control
                    points
#   RETURNS:
#   (float) the length of the bezier curve
def length(mue):
    return integrate.quad(integrand_function, 0, 1, args=mue)

#   Finds the difference between the length of the bezier curve and the chain
#   PARAMETERS:
#   mue - (float) the scalar that adjusts the location of the middle control
                    points
#   RETURNS:
#   (float) the difference between the length of the bezier curve and the
                    chain
def error(mue):
    return length(mue)[0] - chain_length

#   Positions the given bone such it lies on the bezier curve
#   PARAMETERS:
#   bi1     - (float list) A 3 element array containing the location of the
                    tip of the previous bone
#   bone    - (float list) A 3 element array representing the vector of the
                    bone
#   mue     - (float) the scalar that adjusts the location of the middle
                    control points
```

```
#   tmin    - (float) the minimum value of t that the tip of the bone could
                  have (usually the value of t that previous tip has)
#   RETURNS:
#   (float) the value of t that the tip ended up at
#   (boolean) whether or not the root-finding function converged
def position_bone(bi1, bone, mue, tmin):

    #   The distance between the bezier curve at t and the location of the
                  tip of the previous bone, minus the length of the bone
    def func(t):
        value = bt(t, c0, b0, c3, bn, mue)
        error = (value[0]-bi1[0])**2 + (value[1]-bi1[1])**2 + (value[2]-
                  bi1[2])**2 - three_vector.modulus(bone)**2
        return error
    try:
        return_value = optimize.root_scalar(func, bracket=(tmin, 1))
    except ValueError:
        return 0, False
    return return_value.root, return_value.converged

#   Positions all the bones in a chain along a bezier curve
#   mue         - (float) the scalar that adjusts the location of the middle
                  control points
#   bones       - (float list list) An array of 3 element arrays which
                  represent each bone as a vector
#   optimize    - (boolean) whether or the method should return values to be
                  used in root finding
#   RETURNS:
#   (float) distance between the length of the bones on the chain and the
                  chain itself or
#   (float list list) array of 3 element arrays representing the tip of each
                  bone
def make_chain(mue, bones, optimize):
    bi1 = [0, 0, 0]
    tmin = 0
    tips = []
    length = 0
    for i, bone in enumerate(bones):
        length += three_vector.modulus(bone)
        t, converged = position_bone(bi1, bone, mue, tmin)
        print(i)
        if converged == False or (i==len(bones)-1 and t!=1):
            return_val = 100
            return_val *= three_vector.modulus(bone) * (len(bones)-i) -
                  three_vector.modulus(three_vector.sum(three_vector.scale(b
                  t(tmin, c0, b0, c3, bn, mue), -1), c3))
            print("returning: ", return_val)
            return return_val
        tmin = t
        bi1 = bt(tmin, c0, b0, c3, bn, mue)
        tips.append(bi1)
    if optimize==True:
        return 0
    return tips
#   Optimizes mue so that the chain fits the bezier curve
#   PARAMETERS:
```

```
#   bones   - (float list list) An array of 3 element arrays which represent
                  each bone as a vector
#   new_mue - (float) An initial guess for what mue should be
#   RETURNS:
#   (float) the value of mue for which the chain correctly lies on the curve
def finalize_chain(bones, new_mue):
    final_mue = optimize.root_scalar(make_chain, x0=new_mue, args=(bones,
                  True), bracket=(0, 10))
    return final_mue

#Creates a chain based on the control points and given bones
bones = [[0, 0, .57]]*10
chain_length = 0
for bone in bones:
    chain_length += three_vector.modulus(bone)
print(bones)

new_mue = optimize.root_scalar(error, bracket=(0,10)).root
print("optimized mue " + str(new_mue))
final_mue = finalize_chain(bones, new_mue).root
print(final_mue)
print(make_chain(final_mue, bones, False))
```

bezier.py:

```
import three_vector

#   Defines the control points for the bezier curve
#   PARAMETERS:
#   a       - (float list) the position of an adjacent control point
#   b       - (float list) the vector of the direction in which the control
point should lie
#   scalar  - (float) a multiplier of b, controlling how far from the
adjacent point this point should be
#   RETURNS:
#   (float list) the location of the control point
def get_control(a, b, scalar):
    return three_vector.sum(a, three_vector.scale(b, scalar))

#   Returns the equations for a bezier curve defined by 4 control points
#   PARAMETERS:
#   ci  - (float list) a 3 element array [x, y, z] with coordinates for each
control point
def make_third_degree_bezier(c0, c1, c2, c3):
    x = "(1-t)**3 * " + str(c0[0]) + " + 3 * (1-t)**2 * t * " + str(c1[0]) +
" + 3 * (1-t) * t**2 * " + str(c2[0]) + " + t**3 * " + str(c3[0])
    y = "(1-t)**3 * " + str(c0[1]) + " + 3 * (1-t)**2 * t * " + str(c1[1]) +
" + 3 * (1-t) * t**2 * " + str(c2[1]) + " + t**3 * " + str(c3[1])
    z = "(1-t)**3 * " + str(c0[2]) + " + 3 * (1-t)**2 * t * " + str(c1[2]) +
" + 3 * (1-t) * t**2 * " + str(c2[2]) + " + t**3 * " + str(c3[2])
    return x, y, z
```

three_vector.py:

```
import math
```

```
def getVector():
    v1 = input("v1 ")
    v2 = input("v2 ")
    v3 = input("v3 ")
    a = [float(v1), float(v2), float(v3)]
    return a

#gets the magnitude of a vector x
def modulus(x):
    return (x[0] ** 2 + x[1] ** 2 + x[2] ** 2)**0.5

#returns the cross product of vectors a and b
def crossProduct(a, b):
    return [(a[1] * b[2] - b[1] * a[2]), (-1 * (a[0] * b[2] - b[0] * a[2])),
(a[0] * b[1] - b[0] * a[1])]

#returns the dot product of vectors a and b
def dotProduct(a, b):
    return a[0] * b[0] + a[1] * b[1] + a[2] * b[2]

#returns the angle between vectors a and b
def vectorAngle(a, b):
    return math.acos(dotProduct(a, b) / (modulus(a) * modulus(b)))

#returns vector a scaled by double (or integer) s
def scale(a, s):
31

    return [a[0] * s, a[1] * s, a[2] * s]

#adds vectors a and b together
def sum(a, b):
    return [a[0] + b[0], a[1] + b[1], a[2] + b[2]]
```