A Practical Implementation of Kempe's Universality Theorem

Yanping Chen
Laura Hallock
Eric Söderström
Xinyi Zhang

Final Project
6.849 Demaine
Fall 2012

Abstract

For this project, we implemented a JavaScript simulation employing Kempe's Universality Theorem to construct linkages constrained to a user-inputted $f(x, y) = 0$ curve. In addition to the literal linkage construction, the implementation includes novel algebraic and physics systems, as well as various optimization techniques to simplify the linkage in size and complexity.

1. Introduction

Kempe's Universality Theorem describes an algorithm to trace any continuous curve described by the equation $f(x, y) = 0$ using a series of connected linkages to precisely restrict the motion of a single node. For this project, we created a JavaScript simulation that implements this algorithm in generating a linkage to trace a user-inputted curve. The user can then edit or simply manipulate the linkage via the user interface. The software can also run the simulation in an automated manner, tracing the curve without direct user manipulation.

2. Previous Work

We encountered two major sources of previous research on automatically generating Kempe linkages from algebraic constraint input: *Automated Generation of Kempe Linkages for Algebraic Curves in a Dynamic Geometry System* by Alexander Kobel [3] and *Automated Generation of Kempe Linkage and Its Complexity* by Xiaoshan Gao and Changcai Zhu [2]. However, both of these papers dealt more with the theoretical aspect of construction and less with the actual display and simulation. Kobel used the CAD software *Cinderella* with *Xalci* to display the linkages and algebraic curves. However, neither program is very web-friendly — the Java applet on his site does not appear to be in working condition, possibly because the *Xalci* backend is no longer functional. Similarly, Gao mentioned the use of *Geometry Expert*, but we were unable to obtain a copy of his program or any screenshots thereof to see what the interface looks like. Regardless, neither implementation allows for interaction with all points of the linkage, which is something we attempted with our physics system. Similarly, we have encountered no universal linkage simulator. We have encountered several physics engines which allow for "rigid links," but most actually use springs to simulate such links. Additionally, there exist several programs that allow for "simulation" of linkages, but the simulated linkages are usually very limited, and the simulation less a physical than a mathematical one that analytically solves the system of constraints.

3. Kempe Linkage Construction

A Kempe linkage is created by first representing the $f(x, y)$ curve as a sum of cosines in polar coordinates, i.e., a sum of terms of the form $dcos(a\theta + b\phi + c)$ for some constants a, b, c, and d and two variable angles θ and ϕ. Each term is represented by a link of length d, which is restricted to lie on the angle $a\theta + b\phi + c$ by a combination of the multiplicator and additor gadgets described in sections 3.2 and 3.3. This means that the x-coordinate of the link's endpoint is restricted to lie along the curve $dcos(a\theta + b\phi + c)$; these links are then combined via the translator gadget to restrict a given point P, the outermost vertex of a parallelogram whose opposite endpoint is fixed to the origin, to lie on the desired curve (see *Figure 1*). Our construction process draws heavily from Anupam Saxena's 2011 paper "Kempe's Linkages and the Universality Theorem," which contains detailed descriptions of Kempe's fundamental linkages and works through an example of linkage generation for a simple piecewise curve [4].

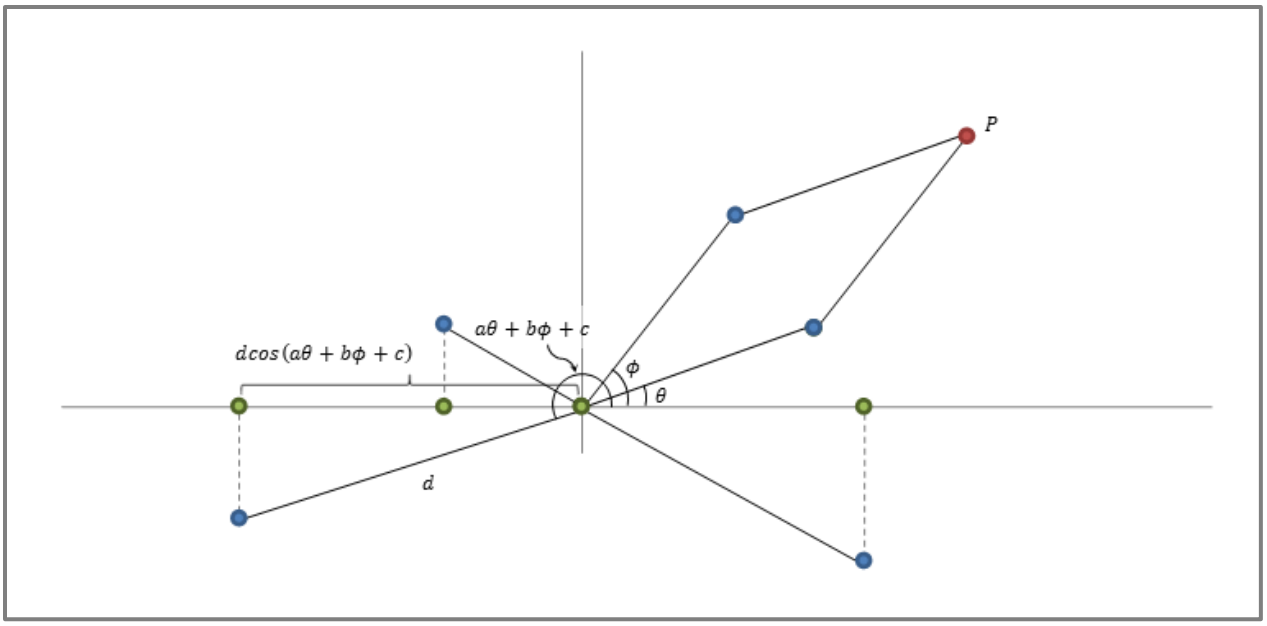


Figure 1: Point P is restricted to lie along a specified curve $f(x, y) = 0$ by a series of linkages dependent on angles θ and ϕ.

3.1 Parallelogram and Contra-parallelogram

The most fundamental building blocks of a Kempe linkage are the parallelogram and contra-parallelogram; they are then combined using the gadgets below, which exploit the linkages' symmetric properties in order to manipulate the relative size and position of various angles within the final linkage (see *Figure 2*). If a parallelogram linkage is manipulated such that all points are made collinear, it can then degenerate into a contra-parallelogram, or vice-versa. This can be fixed by bracing each with additional links. While we implemented such bracing, we elected not to use the code for several reasons. First, our implemented physics system does not cope well with over-constrained systems like those generated by the bracing, and second, our objective is user-interactive animation, not autonomous simulation, so such bracing would simply slow down the graphics while remaining largely unnecessary.

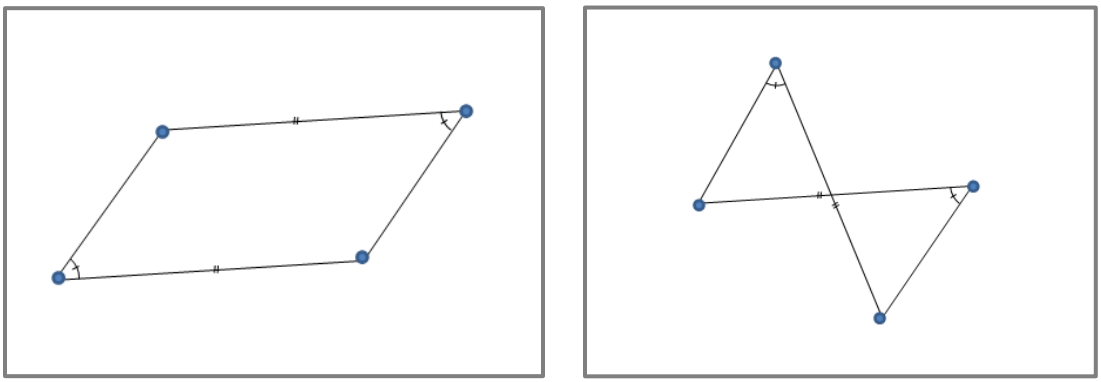
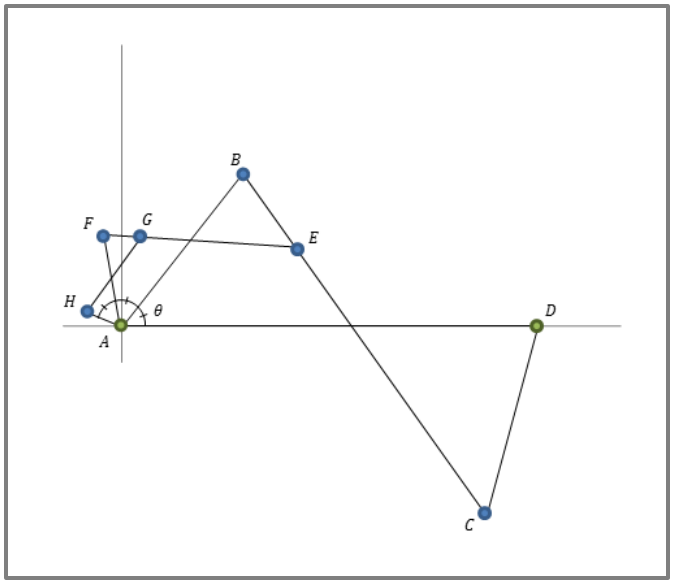


Figure 2: Parallelogram (*left*) and contra-parallelogram (*right*).

3.2 Multiplicator



The multiplicator linkage is used to multiply an angle θ by some integer k; *Figure 3* shows this construction for $k = 3$. Contra-parallelogram $ABCD$ is first constructed such that $\angle BAD = \theta$, point A is fixed to the origin, and point D is fixed to any point on the positive x-axis. Keeping link BC rigid, contra-parallelogram $ABEF$ is then constructed such that $\angle AFE = \angle ABC = \angle ADC$. $ABCD$ is therefore similar to $ABEF$; thus, $\angle FAB = \angle BAD = \theta$, so the two parallelograms have together generated $\angle FAD = 2\theta$. This process is completed k times in order to form the angle $k\theta$. (When $k = 2$, this linkage is also referred to as a *reversor* because the angle θ is reflected, or "reversed," across link AB.)

Figure 3: The multiplicator gadget for $k = 3$, such that $\angle DAH = 3\theta$.

3.3 Additor

The additor linkage is used to add or subtract two angles θ and ϕ; *Figure 4* shows this construction. The linkage is formed by constructing two contra-parallelograms, $ABCD$ and $ABC'D'$, in which link AD is fixed to the x-axis and the two contra-parallelograms share link AB. As in the multiplicator, contra-parallelograms $AFEB$ and $AF'E'B$ are constructed to be similar to $ABCD$ and $ABC'D'$, respectively; thus, by the same contra-parallelogram properties applied in 3.2, link AB bisects both $\angle F'AD'$ and $\angle FAD$. This means that $\angle FAF' = \angle DAD'$. We can then exploit these equalities to create the sum $\theta + \phi$ or the difference $\phi - \theta$ based on our choice of θ and ϕ. An additional constant angle α can then be added or subtracted from either quantity by creating another link and holding it at a fixed angle α to link AF or AF', respectively.

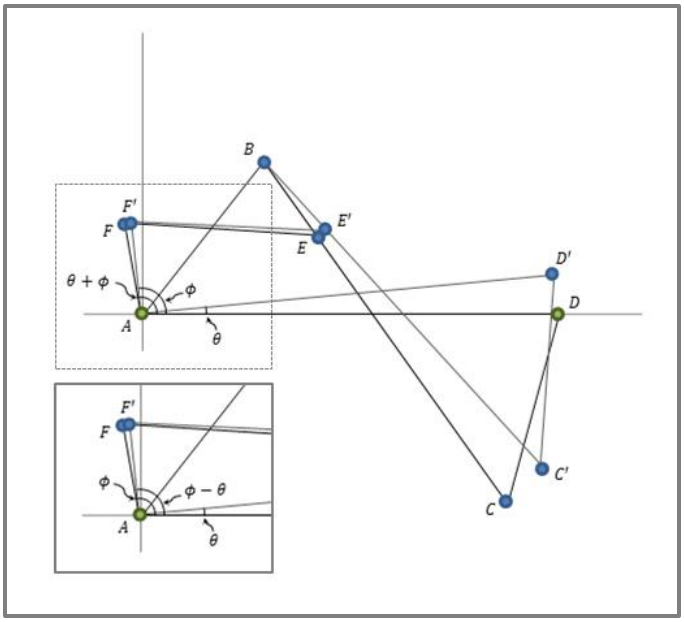


Figure 4: The additor to generate angles $\theta + \phi$ (*top*) and $\phi - \theta$ (*bottom inset*).

3.4 Translator

The translator linkage is used to translate an angle θ laterally, generally to combine it with the rest of the terms in the summation. As shown in *Figure 5*, the linkage simply consists of two parallelograms, which restrict links AB and EF to lie along the same angle θ with the horizontal.

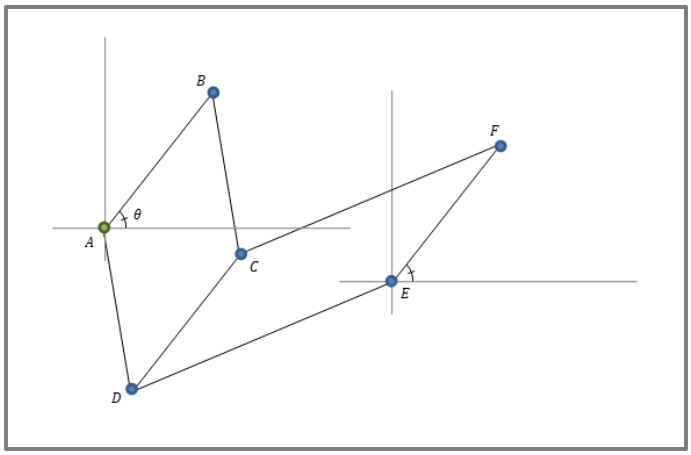


Figure 5: The translator gadget.

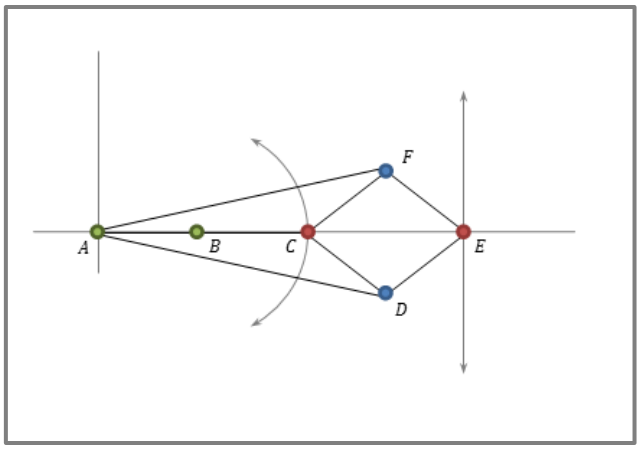


Figure 6: The Peaucellier-Lipkin cell.

3.5 Peaucellier-Lipkin Cell

The Peaucellier-Lipkin linkage predates Kempe's work on linkage instruction and represents the first correct solution to convert rotary motion to a straight line; as in *Figure 6*, fixed link AB restricts point C to move along a curve of radius BC and point E to simultaneously move along a fixed line segment. When the full Kempe linkage is constructed, the user can manipulate point C in order to drive the rest of the construction, which will be attached at point E.

3.6 Putting It All Together

Ultimately, the objective of the Kempe linkage is to restrict a point P to lie along a given curve, which we represent as a sum of cosine terms of θ and ϕ. As previously seen in *Figure 1*, construction begins with a parallelogram anchored at the origin whose links form angles θ and ϕ with the x-axis. These angles are then employed by additor and multiplicator gadgets to create all angles necessary in the eventual construction of the linkage. These angles are then translated atop one another using the translator gadget, which results in a structure culminating in a single point which can then be driven along the x-axis using a Peaucellier-Lipkin cell. *Figure 7* shows an example of a final linkage as implemented in our software; here, point P traces the curve $x^2 - y + 0.3 = 0$, and the Peaucellier-Lipkin cell would be attached at point B, although this functionality was not implemented for reasons discussed in section 3.8.

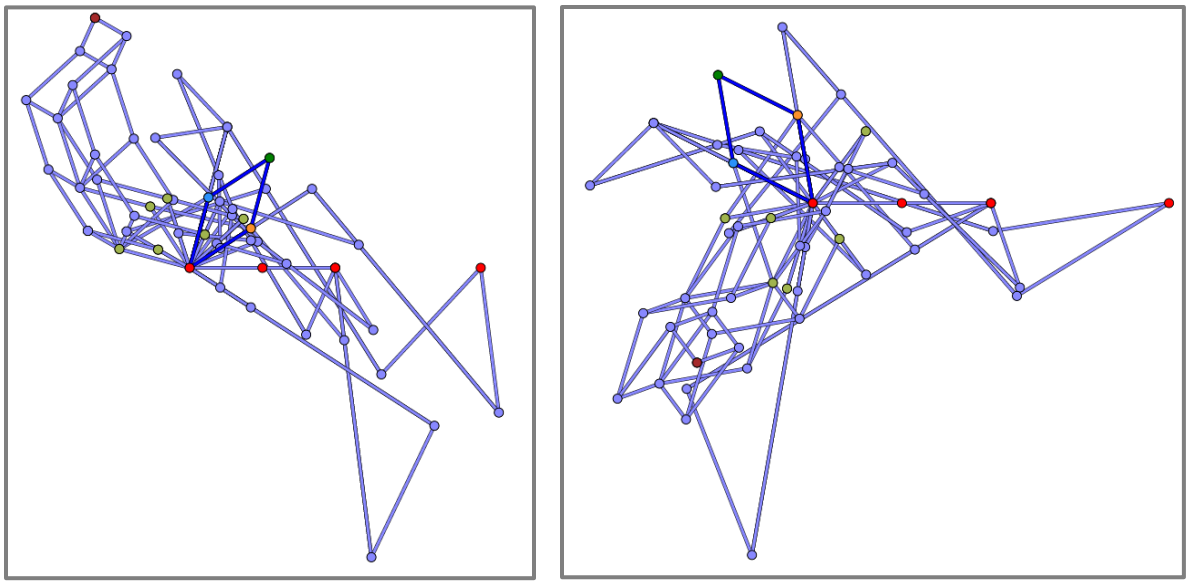


Figure 7: Full Kempe linkage for $x^2 - y + 0.3 = 0$, as implemented in our simulator. Here, the green point traces the indicated curve. Each olive point indicates the construction of a single cosine term and each brown point a sum of cosine terms; the solid dark blue lines and orange and cyan points indicate the drawing parallelogram. Red points are fixed.

3.7 Optimizations

While the constructions mentioned are indeed functional, and from a theoretical point of view they suffice to prove that any algebraic curve can be traced by a linkage with one degree of freedom, they are quite inefficient in several regards. In order to create a displayable and animatible linkage, we developed several optimizations to reduce the number of joints and bars required.

3.7.1 Optimized Multiplicator

One of the most inefficient parts of the Kempe construction is the reality of separate constructions for each term. While this separation makes sense and is quite elegant for a mathematical proof, it is not as practical. One thing to note here is that all the terms can share two multiplicators — one for the the first bar of the parallelogram and one for the second bar. Thus, instead of constructing a multiplicator for each bar for each term, we simply create one multiplicator for each bar with multiplicity equal to the maximum (and minimum) multipliers needed for each bar respectively. We then hook all the additors to the corresponding points on these two multipliers. This greatly reduces the number of bars and joints required, especially on multipart terms such as $\sum_{\{a,b\in[0,5]\}} \cos(a\theta + b\phi)$, where the number of terms is large, but the range of multipliers is not.

Another issue with the multipliers as presented above deals not so much with optimality as with numerical stability. Since the simulation implements a physics system, we need to be careful of disproportionally large or small lengths. The multiplicator construction is actually most likely close to optimal, but it requires a halving of lengths for every additional multiple. This is intrinsically problematic: for instance, starting with an initial parallelogram bar length of 1, after 10 iterations, one ends up with a bar length of less than 0.001. None of the physics systems handle this well: the

ones which solve linear equations end up with determinants either too large or too close to zero, and the ones which use springs treat bar lengths as masses, which means that if the 0.001 length bar was rescaled up to length 1, when the spring-approximated linkage moves, the length-1 bar would take a long time to "catch up" to the proper relative location from the minute force applied by the 0.001-length bar. Another issue with such a shrinking construction is that much of it is not very visible to the user: after approximately five iterations, the bars are already small enough that the user must zoom in in order to see the structure. To deal with this, we devised a version of the gadget in which each multiplicator has smallest bar length of 1. This is done efficiently by "copying and pasting" the original construction for each new multiple, instead of recursively shrinking. In reality, this requires only a few more bars and joints, but because we have already merged all the multiplicators into a single gadget for each parallelogram bar, this is a worthy sacrifice for a more visible and potentially more physics-simulation-stable construction.

Finally, more for completeness than optimization, we implemented negative angle multiples in the multiplicator. This uses the same multiplicator design but shifts the anchor points, such that the linkage creates a "negative" multiple instead of a "positive" one.

The optimized linkage constructions appear in *Figure 8* below.

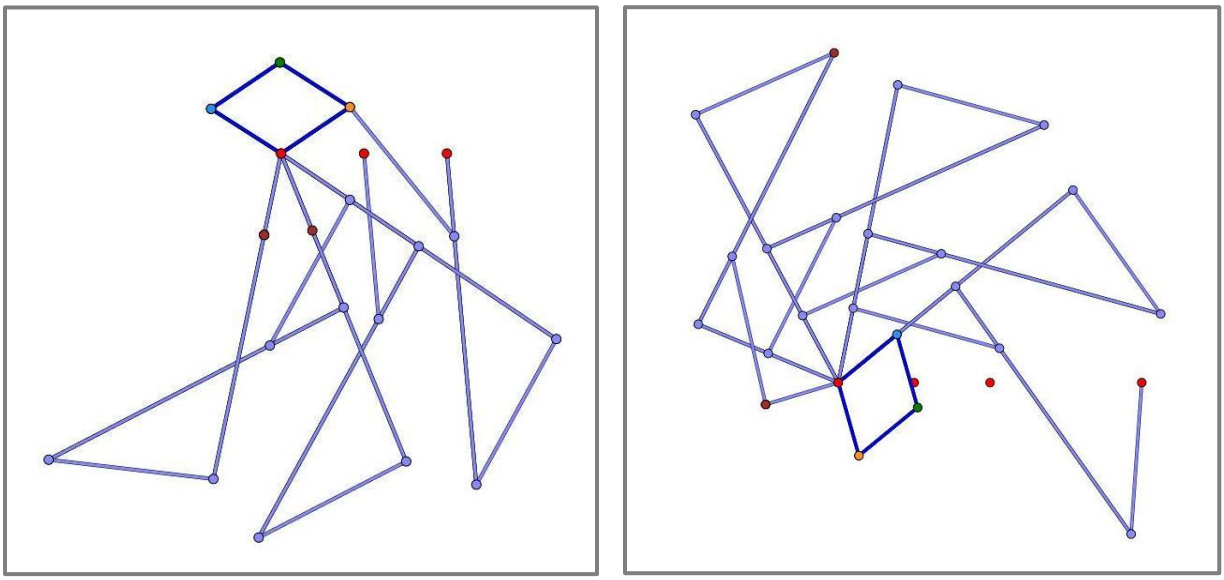


Figure 8: Optimized multiplicator for $k = -3$ *(left)* and $k = 5$ *(right)*.

3.7.2 Side Cases for Additor

The main purpose of optimizing the additor was to deal with the three separate cases: (table with pictures)

both θ and ϕ have nonzero multiples	create an additor (and add on the extra constant angle if it is non-zero)

only θ has a nonzero multiple	only use the joint corresponding to the θ bar (and only add on the extra constant angle if it is nonzero)
only ϕ has a nonzero multiple	only use the joint corresponding to the ϕ bar (and only add on the extra constant angle if it is nonzero)

Aside from optimization, we also noticed that the additor is under-determined. In other words, if only given two points at unit length from the origin, there are actually two possible (and valid) additors that an algorithm could create: one where the angle bisector divides the convex angle and one where the angle bisector divides the reflex angle. An easy way to see this is as follows: for two unit length points A and B, let the constructed additor have angle bisector at C, dividing the convex angle between A and B. Now, fix A and rotate B a full 2π counter-clockwise around the origin. Now, note that A and B are in the same location, but C is now exactly opposite its previous location. Thus, knowing each angle *modulo* 2π is insufficient. None of the previous works we encountered mentioned this: for Kobel, this might be because a slightly different construction was used for adding angles [3], but for Gao, it is unclear if this phenomenon was noticed [2]. Similarly, this was not mentioned in Saxena's article on Kempe's Universality Theorem [4] or in *Geometric Folding Algorithms* [3].

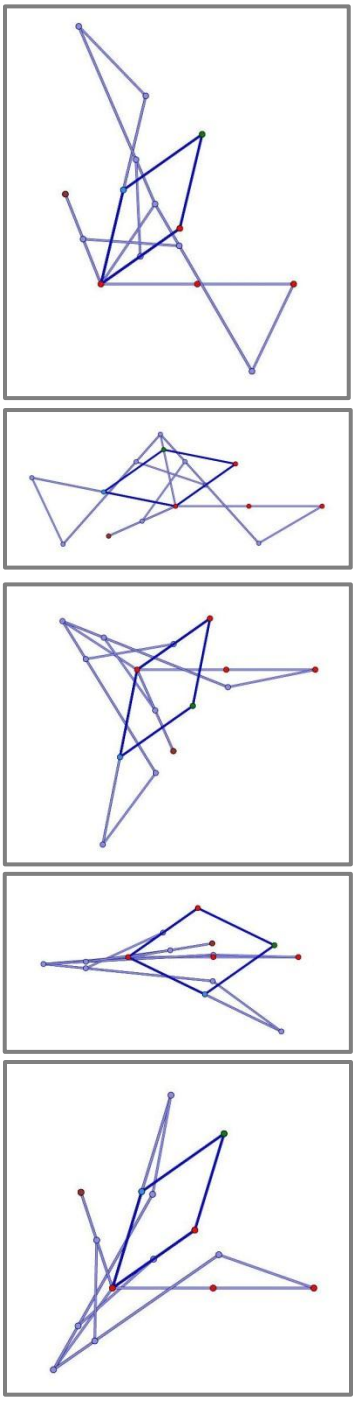


Figure 9: Images depicting the underdetermined nature of the additor. Displaying just the additor, one parallelogram bar is rotated a full 2π, but the linkage ultimately ends up in a different position.

3.8 Coding Remarks

Because our implementation uses JavaScript, we decided that the simplest and most efficient data structure is a simple nested-array data structure. The first array represents points by containing the floating point xy-coordinates of nodes as well as boolean variables indicating whether the point is fixed or free to move. The second array represents edges between pairs of nodes by indexing against the first array, as well as the length of each edge. In code:

```
data = [
           [ // points
                [ pt 1 x coord, pt 1 y coord, true if pt 1 is fixed],
                [ pt 2 x coord, pt 2 y coord, true if pt 2 is fixed],
                ...
           ],
           [ // edges
                [ edge 1 1st endpt index, edge 1 1st endpt index, edge 1 len],
                [ edge 2 1st endpt index, edge 2 1st endpt index, edge 2 len],
                ...
           ]
       ]
```

So, for example, the drawing parallelogram can be represented as:

```
data = [
           [ // points
                [       0,      0,      false],
                [       0,      1,      false],
                [       1,      0,      false],
                [       1,      1,      false]
           ],
           [ // edges
                [       0,      1,      1],
                [       0,      2,      1],
                [       1,      3,      1],
                [       2,      3,      1]
           ]
       ]
```

As shown, we decided to implement a simplified representation in which bars can only be joined to other bars at their endpoints. This requires that we use a three-bar approximation for linkages requiring mid-bar joints. This is, however, worth the simpler representation, as this makes the code much easier to manipulate, implement, and debug, since there is no complicated data structure to track for each individual bar.

For the actual linkage construction, there are five inputs. We have three main inputs: the xy-coordinates of the first parallelogram bar, the xy-coordinates of the second parallelogram bar, and finally, a list of the cosine terms. (There are two additional inputs for correctly constructing the additor which will be discussed later.) We begin by iterating through the cosine terms to find the maximum and minimum multipliers for each angle. Then, for any angle with a non-trivial multiple, we create the multiplicator for that angle. Next, we iterate through the cosine list and construct the appropriate additor for each term. Finally, we piece the full linkage together by creating the translator chain. While we do have code for creating a Peaucellier-Lipkin linkage, there is little

point for display purposes: it will be much larger than the rest of the linkage in most cases, and it is easy to observe by inspection that the final point indeed lies on a vertical line.

As mentioned, there are two additional inputs — these are the "aggregate" angles of the two parallelogram bars. Here, aggregate refers to the fact that if a bar swings a full circle, its angle is not 0, but instead 2π. We keep track of this by checking the angle of the bars at each frame of the update loop. Because the bars can only move a minute amount, we can determine if the angle increased — i.e., if the bar rotated counter-clockwise — or decreased — if the bar rotated clockwise. These aggregate angles are passed into the linkage creation for accurate and continuous creation of the additors. Each additor takes the corrsponding aggregate angle and multiplies it by the corresponding multiple, then takes the average of the two as the angle bisector. This creates the "correct" angle bisector, where "correct" means that the location of the angle bisector is continuous. This is very important when we are not simply generating the linkage once at the start for physical simulation, but are instead reconstructing the linkage each frame to animate it, as it removes the ugly "popping" that occurs when the additors are underdetermined.

Finally, we discovered some extremely useful formulas in vector algebra for general purposes, such as dropping the perpendicular of a point to a line and finding intersections of lines specified by two points each. These were quite nice and made the corresponding code much shorter and cleaner than the typical methods involving slope, which must deal with the side cases of vertical lines (and possible horizontal lines as well).

4. Algebra System

The first step in constructing a linkage from an algebraic curve is to turn the algebraic constraint into a constraint involving cosines of integer linear combinations of the two constituent angles. With respect to code, we require the additional step of parsing the user input into a virtual representation of the algebraic formula. Since we were unable to find any good implementations of a computer algebra system in JavaScript, we decided to write our own.

4.1 Computer Algebra System

Our algebra system has two main constructs: *Terms* and *Equations*. A Term is a single algebraic term, consisting of a coefficient and a list of variables and their powers; an Equation is a list of such Terms. To keep parsing simple, variables are constrained to be upper- and lower-case alphabetical characters. In code, an Equation is simply represented as an array of Terms. A Term is an object with a coefficient and a mapping of variables to their respective powers. For example:

```
algebraic string: "-x^2+2xy+y^2"

Term1 = {
                        // coefficient
                        coeff = -1;
                        // mapping of variables to powers
                        powers = {
                                        x : 2
                                };
                }
```

```
Term2 = {
                        // coefficient
                        coeff = 2;
                        // mapping of variables to powers
                        powers = {
                                        x : 1,
                                        y : 2
                                };
                }
Term3 = {
                        // coefficient
                        coeff = 1;
                        // mapping of variables to powers
                        powers = {
                                        y : 2
                                };
                }

Equation = {terms = [Term1, Term2, Term 3];}
```

Our algebra system supports the following operations, whose implementation we will describe below: addition of Terms, multiplication of Terms, simplification of Equations, addition of Equations, multiplication of Equations, substitution of value for variable, and substitution of Equation for variable.

4.1.1 Addition of Terms

This is straightforward — if the variables and powers match, then just add the coefficients and return the new term. Otherwise, add two copies of the Terms being added into an array and return it as an Equation.

```
add(term1, term2)
{
        if term1.powers == term2.powers
        {
                result = new Term();
                result.coeff = term1.coeff + term2.coeff;
                result.powers = term1.powers;
                return result
        }
        else
                return new Equation([term1, term2]);
}
```

4.1.2 Multiplication of Terms

Here, we simply create a new Term whose coefficient is the product of the coefficients of the original Terms, and whose variables' powers are the sums of the powers of the variables of the original Terms, making sure to remove any variables whose powers sum to 0.

```
multiply(term1, term2)
{
```

```
        result = new Term();
        result.coeff = term1.coeff * term2.coeff;
        for variable in term1.powers
                result.powers[variable] += term1.powers[variable];
        for variable in term2.powers
                result.powers[variable] += term2.powers[variable];
        return result;
}
```

4.1.3 Simplification of an Equation

To simplify an Equation, we combine like Terms through the use of an array and remove any Terms with a coefficient of 0.

```
simplify(equ)
{
        for term1 in equ.terms
        {
                for term2 in rest of equ.terms
                {
                        if term1.powers == term2.powers
                        {
                                insert add(term1,term2) into equ.terms
                                remove term1 and term2 from equ.terms
                        }
                }
        }
}
```

4.1.4 Addition of Equations

Here, we simply append the array of terms of one Equation to the other, then simplify the resultant Equation.

```
add(equ1, equ1)
{
        result = new Equation();
        result.terms.concat(equ1.terms);
        result.terms.concat(equ2.terms);
        simplify(result);
        return result;
}
```

4.1.5 Multiplication of Equations

Here, we loop over every pair of Terms of the two Equations being multiplied, multiplying them and appending them to an array. We then make an Equation out of the array and simplify it.

```
multiply(equ1, equ2)
{
        result = new Equation();
```

```
        for (term1 in equ1.terms)
        {
                for (term2 in equ2.terms)
                {
                        result.terms.push( multiply(term1, term2) );
                }
        }
        simplify(result);
        return result;
}
```

4.1.6 Substitution of Value for Variable

For a Term, we check if the variable has a nonzero power. If so, we substitute in the value, multiplying by the coefficient, and remove that variable from the powers mapping. For an Equation, we substitute the value in for each Term, then simplify.

```
sub(term, variable, value)
{
        if term.powers[variable] != 0
        {
                term.coeff *= pow(value, term.powers[variable]);
                term.powers[varaible] = 0;
        }
        return term;
}
sub(equ, variable, value)
{
        for term in equ.terms
        {
                sub(term, variable, value);
        }
        simplify(equ);
        return equ;
}
```

4.1.7 Substitution of Equation for Variable

This operation is the most complicated of all the operations we implemented but is necessary for making the crucial substitutions of $x = \cos\theta + \sin\phi$ and $y = \sin\theta + \sin\phi$. We follow the same steps we made in substituting a value for a variable, but instead of simply multiplying each Term's coefficient by the appropriate value, we need to multiply it by an Equation instead. Additionally, for calculating powers, we simply use repeated multiplication of Terms — this is indeed a little inefficient, but because the Kempe construction would get very complicated for any large powers anyway, speed is hardly an issue.

```
sub(term, variable, newequ)
{
        if term.powers[variable] != 0
        {
                calculate newequ to power of term.powers[variable]
                term.powers[variable] = 0;
```

```
                return multiply(newequ, term);
        }
        return term;
}
sub(equ, variable, newequ)
{
        for term in equ.terms
        {
                sub(term, variable, newequ);
        }
        simplify(equ);
        return equ;
}
```

4.2 Cosine Conversion and Trigonometric Identities

Now, after parsing the user input into an Equation, we let the following variables represent the sine and cosine of angles:

$$a = \cos\theta$$
$$b = \cos\phi$$
$$c = \cos\theta$$
$$d = \cos\phi$$

We then make the substitutions $x = a + b$ and $y = c + d$ to obtain an algebraic equation in terms of the sines and cosines. However, we still need to remove all the products of trigonometric functions. This is achieved using the product-to-sum trigonometric identity:

$$\cos a \cos b = \frac{1}{2}(\cos(a+b) + \cos(a-b))$$

Instead of implementing a fully functional trigonometric system, we simply programmed in cases for each of the possible terms:

$$\cos(a\theta + b\phi + c)\cos\theta = \frac{1}{2}(\cos((a+1)\theta + b\phi + c) + \cos((a-1)\theta + b\phi + c))$$

$$\cos(a\theta + b\phi + c)\cos\phi = \frac{1}{2}(\cos(a\theta + (b+1)\phi + c) + \cos(a\theta + (b-1)\phi + c))$$

$$\cos(a\theta + b\phi + c)\sin\theta = \cos(a\theta + b\phi + c)\cos(\theta - \frac{\pi}{2})$$
$$= \frac{1}{2}(\cos((a+1)\theta + b\phi + c - \frac{\pi}{2}) + \cos((a-1)\theta + b\phi + c + \frac{\pi}{2}))$$

$$\cos(a\theta + b\phi + c)\sin\phi = \cos(a\theta + b\phi + c)\cos(\phi - \frac{\pi}{2})$$
$$= \frac{1}{2}(\cos(a\theta + (b+1)\phi + c - \frac{\pi}{2}) + \cos(a\theta + (b-1)\phi + c + \frac{\pi}{2}))$$

However, this is not quite enough. After implementing the previous step, our output is the following:

$$\cos\theta\cos\phi + \sin\phi\sin\theta = \frac{1}{2}\cos(\theta+\phi) + \frac{1}{2}\cos(\theta-\phi) + \frac{1}{2}\cos\left(\theta+\phi+\frac{\pi}{2}\right) + \frac{1}{2}\cos(-\theta+\phi)$$

This is clearly not simplified, and will thus result in a needlessly complex linkage. In order to make the resulting linkages simpler, we implemented a few more trigonometric identities:

$$\cos(-x) = \cos x$$
$$\cos(x+\pi) = -\cos x$$

Combining all of the above finally gives us the cosine angle subtraction identity:

$$\cos\theta\cos\phi + \sin\phi\sin\theta = \cos(\theta-\phi)$$

5. Physics

The basic premise for animating the motion of the linkage is to model the linkage as a particle system. Forces can be applied to this particle system, either by the user or by the rigid constraints, and a numerical integrator is used to evaluate the new position of the nodes and edges after one time step. There are then two main functions involved in simulating the physics of the linkage, *evalForces* and *timeStep*:

`evalForces(state)`	Given the current positions and velocities of all nodes in the linkage, computes the forces experienced by those nodes.
`timeStep(state, forces, h)`	Given a state and forces acting on that state, uses numerical integration to update the positions and velocities of all nodes in the linkage after one time step of duration *h*.

Physical simulation is of course not the only method for animating a linkage construction. Cinderella is a piece of software that allows the user to animate the motion of a linkage by cleverly setting up constraints, such as defining a node to lie on the intersection of two circles. This allows for the motion of a specific linkage, but only in a very limited sense, and it requires that the user be able to provide these mathematical constraints for the system. By simulating a linkage as particle system, we remove the burden of providing constraints from the user, and create a more general-purpose linkage simulator.

5.1 Springs

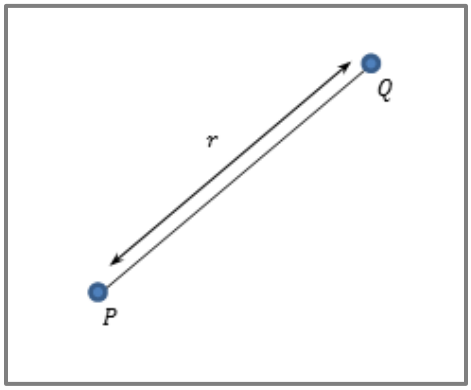


Figure 10: Points P and Q connected by edge of length r.

Most JavaScript physics libraries use the basic particle system and numerical integration abstraction described above. Rigid bars are frequently approximated as very stiff springs. To simulate two points P and Q connected by a bar of length r (see *Figure 10*), the spring force would be calculated using Hooke's Law:

$$F_{q\ on\ p} = -k_{spring}(||d|| - r)\frac{d}{||d||}$$

where $d = P - Q,$ and k_{spring} is the spring constant.

The fundamental problem with this approach is that it requires particles to communicate forces with each other indirectly through displacement. This means that to simulate a rigid bar requires extremely stiff springs. The differential equations for very stiff springs require numerical integration with a time step inversely proportional to k_{spring}, which rapidly becomes mathematically intractable for large values of k_{spring}. This doesn't pose too much of a problem for applications in which complete accuracy isn't too important. For example, cloth is commonly modeled as a mesh of points connected by springs. The fact that each pair of points in a cloth mesh model do not perfectly obey the rigid bar constraint doesn't matter, as the overall look and feel of cloth is preserved.

For modeling linkages, however, especially large linkages such as those produced by the Kempe algorithm, small numerical error will inevitably build up, and springs are simply insufficient for this task. Instead, we need a way to directly calculate the constraint force that will guarantee that the distance between a pair of nodes connected by a rigid bar will remain fixed.

5.2 Rigid Constraints

Spring restoring forces are not good enough for large linkages, so we need to use a different technique for determining the constraint forces in our particle system. The overarching idea for rigid bar simulation is to calculate a constraint force on each of the nodes in our linkage as part of the *evalForces* function. This constraint force must obey the following two properties.

1. The rigid constraint forces, when applied to each node in the system, will cause those nodes to move in exactly the path that preserves the distance constraints created by the edges in the linkage.
2. The constraint forces must not change the overall kinetic energy of the system. Otherwise the system could rapidly explode numerically.

The technique involved in implementing these properties is clear when applied to a very simple linkage, like that shown in *Figure 11*, and can be extended to more complex linkages.

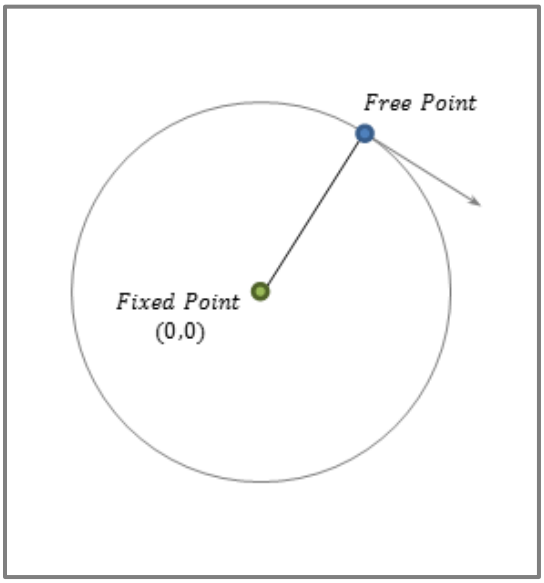


Figure 11: Free point constrained to lie within unit length of the origin.

Figure 11 shows a free node, in blue, connected by an edge of unit length to a fixed point, in green, located at the origin. The large grey circle shows the valid positions that the blue node can occupy while maintaining the distance constraint imposed by the rigid bar connecting it to the green node. We can formalize these position, velocity, and acceleration constraints as shown below. Let p represent a tuple (x, y) of the spacial coordinates for the blue node. We here use the convention that $x \cdot y$ represents the scalar dot product between two vectors x and y, $\dot{x}$ represents the time derivative of x, and $\ddot{x}$ represents the second time derivative of x.

$$C = \frac{1}{2}(p \cdot p - 1) = 0 \qquad \text{(legal positions)}$$
$$\dot{C} = p \cdot \dot{p} = 0 \qquad \text{(legal velocities)}$$
$$\ddot{C} = \ddot{p} \cdot p + \dot{p} \cdot \dot{p} = 0 \qquad \text{(legal accelerations)}$$

If the system is initialized with a legal position and velocity, then it will maintain legal position and velocity as long as it only undergoes legal acceleration. Using Newton's condition $F = ma$, we have:

$$\ddot{p} = \frac{f+\hat{f}}{m} \tag{1}$$

where f is the applied force on point p, and $\hat{f}$ is the rigid constraint force that must be *added* to the system in order to preserve legal accelerations. Simply stated, the acceleration of node p is the net force divided by the mass. This substitution yields the following formula for $\ddot{C}$:

$$\ddot{C} = \frac{f+\hat{f}}{m} \cdot p + \dot{p} \cdot \dot{p} = 0 \tag{2}$$

which can be rewritten as,

$$\hat{f} \cdot p = -f \cdot p - m\dot{p} \cdot \dot{p} \tag{3}$$

This gives us a formula for computing the constraint force $\hat{f}$ in terms of the velocity $\dot{p}$, the position p, and the applied force f. This alone is not quite enough, because $\hat{f}$ has two components (one in the x-direction and one in the y-direction). To obtain a specific value for $\hat{f}$ we must take advantage of the second requirement for a rigid constraint force — that it must preserve the kinetic energy of the system. From 8.01, we recall that the kinetic energy of a moving mass is equal to $\frac{1}{2}mv^2$. Applying this to our simple linkage, we can express the kinetic energy, KE as follows:

$$KE = \frac{m}{2}\dot{p} \cdot \dot{p} \tag{4}$$

And its derivative with respect to time,

$$\dot{KE} = m\ddot{p} \cdot \dot{p} = mf \cdot \dot{p} + m\hat{f} \cdot \dot{p} \tag{5}$$

The time derivative of kinetic energy of the system can also be conceptualized as the amount of work done on that system by the total force $f + \hat{f}$. In order to guarantee that the constraint force $\hat{f}$ does no work on the system, we need to ensure that $\hat{f} \cdot \dot{p} = 0$ for all legal velocities. We determined already that legal velocities are those that satisfy $\dot{C} = p \cdot \dot{p} = 0$. In other words,

$$\hat{f} \cdot \dot{p} = 0, \forall \dot{p} \mid p \cdot \dot{p} = 0 \tag{6}$$

This simply means that $\hat{f}$ *must* be a scalar multiple of the position vector p. Intuitively, this should make sense. If the constraint force is to do no work, it must lie in a direction normal to the position constraint curve shown in *Figure 11*. Any such vector acting on point p and lying normal to the constraint curve must be pointing in the same direction as the position vector p:

$$\hat{f} = \lambda p \tag{7}$$

Substituting back into (3), and solving for λ, we get

$$\lambda = -\frac{f \cdot p - m\dot{p} \cdot \dot{p}}{\dot{p} \cdot \dot{p}} \tag{8}$$

λ is called the Lagrange multiplier for this particular constraint, and once computed allows us to solve for the rigid constraint force $\hat{f}$ explicitly using $\hat{f} = \lambda p$, from (7).

5.3 Our Implementation

In order to simulate a full linkage, we will of course need many constraints. Fortunately, the method described in section 5.2 works for a system of constraints as well as for single constraints. Instead of having a constraint formula for legal position, legal velocity, and legal acceleration, we now have constraint vectors. These vectors are essentially lists of each of the distance constraints imposed by each edge in the linkage.

$C = [C_1, C_2, \dots C_m] = [0, 0, \dots 0]$ (legal positions)
$\dot{C} = \left[\dot{C_1}, \dot{C_2}, \dots, \dot{C_m} \right] = [0,0, \dots 0]$ (legal velocities)
$\ddot{C} = \left[\ddot{C_1}, \ddot{C_2}, \dots, \ddot{C_m} \right] = [0,0, \dots 0]$ (legal accelerations)

Similarly, instead of a single applied force f, we now have a vector of applied forces of length $2n$, where n is the number of nodes:

$$Q = [f_1, f_2, \dots, f_n]$$

and instead of a single position vector p, we now have a vector q of length $2n$ containing the x and y position coordinates of each of n nodes:

$$q = [x_1, y_1, x_2, y_2, \dots x_n, y_n]$$

$\dot{C}$ can be computed directly from C by differentiating using the chain rule. Symbolically, we can represent $\dot{C}$ as

$$\dot{C} = \frac{\partial C}{\partial q}\dot{q}$$

where $\frac{\partial C}{\partial q}$ is the Jacobian Matrix of C, and $\dot{q}$ is the element-wise time derivative of q. A second application of the chain rule gives us

$$\ddot{C} = \dot{J}\dot{q} + J\ddot{q}$$

The matrix $\dot{J}$ is the element-wise time derivative of the Jacobian Matrix.

Just as before, when we used $F = ma$ to substitute for acceleration in terms of force, we can now use

$$Q_{net} = M\ddot{q} \rightarrow \ddot{q} = M^{-1}(Q + \hat{Q})$$

to express the acceleration vector in terms of the velocity vector. M is a matrix containing the values of the masses for each node. However, because each node in a linkage can be described as having unit mass without loss of generality, I will omit M^{-1} for the remainder of this derivation. This leaves us with

$$\ddot{C} = \dot{J}\dot{q} + J(Q + \hat{Q}) = 0$$

Rearranging to isolate $\hat{Q}$,

$$J\hat{Q} = -\dot{J}\dot{q} - JQ \tag{9}$$

So far, we have done nothing more complex than using matrix notation to represent a list of constraints, positions, and forces, and using Newton's second law to express acceleration in terms of applied forces. We now use the fact that the rigid constraint forces must not change the kinetic energy of the system, symbolically represented as

$$\hat{Q} \cdot \dot{q} = 0, \forall \dot{q} \,|\, J\dot{q} = 0$$

Thus, we can express the constraining force vector as

$$\hat{Q} = J^T\lambda$$

λ is a vector with one element for each constraint equation. These are the Lagrange multipliers of the constraints, and essentially how much each constraint should be weighted in determining the final rigid constraint force vector $\hat{Q}$. Substituting back in to equation (9), we finally arrive at

$$JJ^T\lambda = -\dot{J}\dot{q} - JQ \tag{10}$$

This is analogous to equation (8) from the one-constraint example. From this equation, we numerically solve for the Lagrange multiplier vector λ, and determine the rigid constraint force vector $\hat{Q}$.

In order to account for numerical error that can accumulate from our integration step, we introduce restoring force terms to equation (10).

$$JJ^T\lambda = -\dot{J}\dot{q} - JQ - k_s C - k_d \dot{C} \tag{11}$$

We see that when the linkage is in a valid state, $C = \dot{C} = \mathbf{0}$, so these terms contribute nothing. But when the linkage has drifting slightly off from a valid configuration, the terms provide a damped restoring force that quickly brings the system back to a valid configuration.

5.4 Limitations

Inspection of equation (11) shows that there is no solution for λ when the matrix JJ^T is non-invertible. Fortunately, this doesn't happen too often. In practice, we have observed two main situations in which JJ^T will be singular:

1. In an infinitely taut situation, such as that shown in *Figure 11*, we can see there will be no linear combination of constraint forces that could counteract an applied downward force. The constraint force supplied by the fixed green point on the left would have to point straight toward the movable blue node, as proven in the derivation in section 5.2. Similarly, the constraint force from the rightmost fixed green node would have to point straight toward the movable blue node. When this linkage becomes infinitely taut, these two constraint forces will span only a single dimension, and it becomes impossible to nullify any applied forces with a non-zero component in the vertical direction.
2. In the case of over-bracing, the constraint matrix C is no longer linearly independent, which implies that the Jacobian of C is also not linearly independent. Thus, JJ^T is non-invertible and there is no solution for λ.

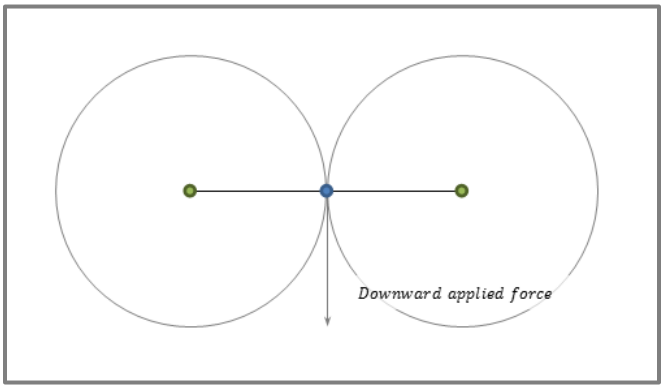


Figure 12: Two fixed points (green) connected to a free point (blue). The free point is experiencing a downward applied force. Large grey circles represent constraint curves imposed by each of the two fixed points.

Constraining a node to lie on a curve is fundamentally no different from constraining a node to lie within a fixed distance of another node. The algebraic curve is simply one additional term in our vector of constraints C. By taking the appropriate partial derivatives of the algebraic curve, we can also still compute J and $\dot{J}$ just as we did before in section 5.3. This allows us to animate the parallelogram portion of the Kempe construction such that the tracing node, shown in red in *Figure 12*, is guaranteed to trace along the given algebraic curve. From the geometry of this constrained parallelogram, we can derive the configuration of the rest of the linkage. This moderately "hacky" solution is only used for the universal Kempe portion of our project. The general Kempe linkage simulator uses the full constraint physics, because it is unlikely for a user to create a large enough

linkage with enough degeneracy to cause the matrix JJ^T to become non-invertible. Even if this matrix is non-invertible momentarily, our approximation for λ using the conjugate gradient method is sufficient to return the linkage to a valid configuration for a general linkage construction.

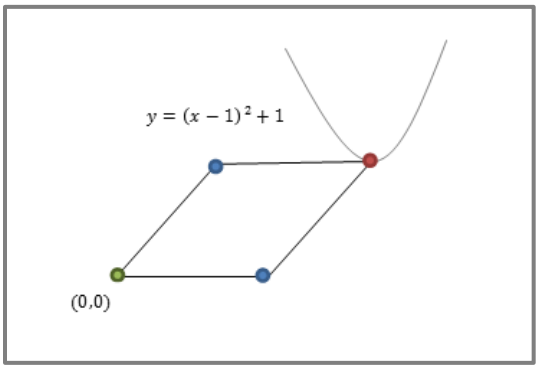


Figure 13: Parallelogram of Kempe construction, with the red node constrained to lie on the given parabola.

6. User Interface

Overall, we decided to code our implementation in JavaScript primarily from a user interface and accessibility perspective. JavaScript makes it easy to draw shapes with the new HTML5 canvas, but more importantly, it requires nothing more than a modern browser, e.g., Chrome, to run the application — there is no need to installing any plugins or download any software.

We attempted to keep the user interface as user-friendly and quick-to-respond as possible. In the physics simulation mode, the user can interact via the mouse:

> **clicking and dragging** on a free point applies a force in the direction of the mouse on that point
> **dragging** anywhere else moves the view
> **scrolling** zooms in and out

In edit mode, users can quickly create and edit linkages:

> **dragging** on any point moves it
> **clicking + CTRL** creates new free points
> **clicking + CTRL + SHIFT** creates new fixed points
> **clicking + ALT** deletes points and all associated edges
> **dragging + SHIFT** between two points creates a new edge (or removes the existing one) between those two points
> **dragging / scrolling** still moves and zooms the view as before

We decided on this series of controls because we felt it was much more intuitive and much faster to use than controls involving clicking on buttons to make new points or edges. These controls are also much easier to implement than having to devise a method of selecting points and edges, since it takes a nontrivial amount of computation to determine which edges are under the cursor.

7. Conclusion and Future Work

Ultimately, we have created a working JavaScript implementation of Kempe's construction algorithm for generating linkages. This includes physically simulating the rigid constraints that constitute linkages with a linkage editor, as well as constructing and animating a Kempe linkage for a given algebraic curve. In order to do so, we have not only implemented the Kempe construction, but also an algebraic system for parsing algebraic user input into a sum of cosines and a physics system that accurately simulates rigid constraints.

While everything we constructed works appreciably well, there are still a number of places to both improve and expand upon our work on this project. One area of future improvement would be a more flexible algebraic input parser which allows parameterized input for multiplying equations; here we have left that conversion responsibility to the user. Additionally, the physics system can be much improved by plugging in a better conjugate gradient solver. The current system is functional, but only for relatively simple systems — a complex or over-constrained system would result in a degenerate matrix in the final solving of constraints, and the current conjugate gradient solver does not compute a suitably fast and accurate solution. A more ambitious improvement might be to implement more simplifications for complicated linkages — perhaps multiple additors could be replaced with a single, larger additor, saving nodes and edges in the process. Finally, the system is noticeably slower for very large Kempe linkages. Perhaps this can be fixed by moving to a more performance-oriented language such as C++, though this would be far less portable than our current implementation.

8. References

[1] Demaine, Erik D., and Joseph O'Rourke. *Geometric Folding Algorithms: Linkages, Origami, Polyhedra*. Cambridge University Press, 2008.

[2] Gao, Xiaoshan, and Changcai Zhu. *Automated Generation of Kempe Linkage and Its Complexity*. Journal of Computer Science and Technology 14.5 (1999): 460-467.

[3] Kobel, Alexander, and Frank-Olaf Schreyer. *Automated Generation of Kempe Linkages for Algebraic Curves in a Dynamic Geometry System*. Diss. Bachelor's Thesis in Computer Science, Saarland University, 2008.

[4] Saxena, Anupam. *Kempe's Linkages and the Universality Theorem*. Resonance 16.3 (2011): 220-237.

[5] Witkin, Andrew, David Baraff, and Michael Kass. "Physically Based Modeling." Siggraph 2001 Course Notes. N.p., 2001. Web. <http://www.computer-graphics.se/TSBK03-files/pdf/Witkin_SGnotes2001_Physically%20based%20modeling.pdf>.