Simulating Mechanical Curve Drawing using Pymunk

Andrew Davison

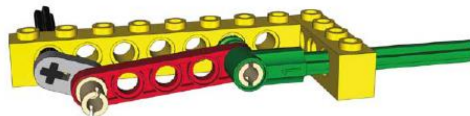
Dept. of Computer Engineering
Prince of Songkla University
Hat Yai, Songkhla 90110, Thailand

E-mail: ad@coe.psu.ac.th

February 2025

For several years I've been teaching geometry to school kids by having them use LEGO Technic pieces to build linkages for drawing straight lines, parabolas, ellipses, and more. This approach has a long history, notably promoted by Zoltán Kovács (https://matek.hu/zoltan/) who uses GeoGebra (https://www.geogebra.org/) to work through the mathematics behind the linkages, and implements them using LEGO [5]. His GeoGebra diagrams are available at https://www.geogebra.org/u/zoltan and instructions for the LEGO builds are at https://github.com/kovzol/lego-linkages. Many videos of people building LEGO linkages can be found by searching for "LEGO linkages" at YouTube; three good ones are: https://www.youtube.com/watch?v=HsEMJOa7WPk, https://www.youtube.com/watch?v=1PX3gEP3ATw, and https://www.youtube.com/watch?v=PD2dtqnTJQA.

These videos tend to restrict themselves to mechanisms which draw straight lines (e.g. Watt's, Chebyshev, and the Peaucellier-Lipkin linkages), which are relatively simple to build with Technic beams pegged together with rotational joints. One exception is the crank which benefits from a sliding (prismatic) joint, as implemented using the green axle below.



Drawing curves mechanically (albeit without the benefits of LEGO) dates back to ancient Greece, often for solving famous problems of antiquity: doubling the cube, squaring the circle, and trisecting an angle. Examples in Rene Descartes' *Geometry* (1637) frequently show how to draw a curve with an apparatus, and this approach was taken up and extended by Frans van Schooten whose *A Treatise on Devices for Drawing Conic Sections* contains numerous mechanical sketches. Incidentally, it's rewarding to use an historical viewpoint like this when teaching [7; 6].

More 'recent' texts (from the 1950s and 1960s) include *Curves and their Properties* by Yates [8] and chapter 5 of *Mathematical Models* by Cundy and Rollett (a wonderful book) [1]; both are available at the Internet Archive.

The main problem I had with GeoGebra and LEGO was quickly pointed out by my students – the physical builds usually fail to draw the clean, smooth curves rendered by the software. Real world issues related to mass, friction, torque, elasticity, and rigidity (kinematics in other words) aren't captured by the purely geometric descriptions in GeoGebra. A (simple) tool which lets students experiment with linkages that have these properties would help to fill the gap between geometry and its implementation. It would also help address the historical question of why there are so many different mechanisms for drawing the same curves.

1. The linkUtils Library

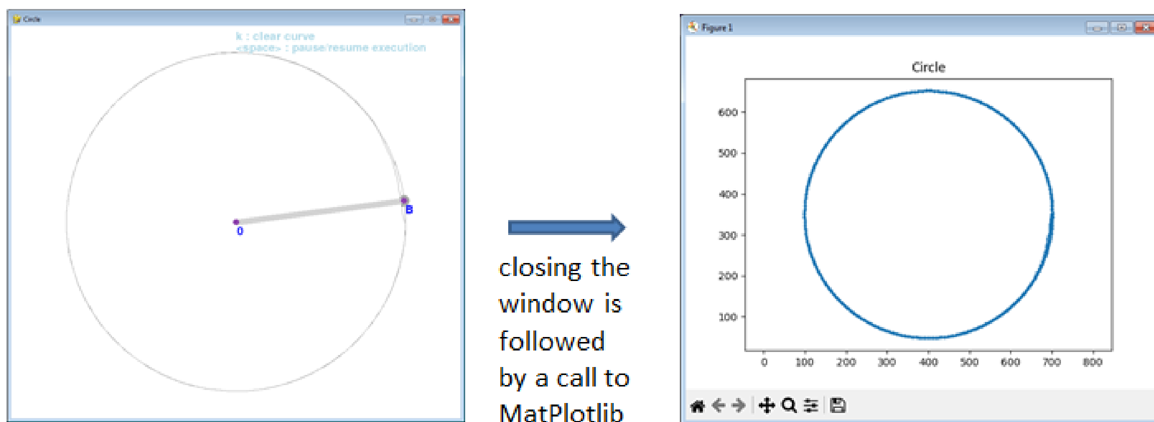
The download at https://coe.psu.ac.th/ad/curveDraw/ includes many examples using my linkUtils Python module, a small collection of functions that utilize Pymunk (https://www.pymunk.org/) and Pygame (https://www.pygame.org/) for creating and animating 2D linkages. The library is designed around the notion of 'beams' pinned together by 'balls' that support rotation. These are 2D shapes, conceptually resting on a sheet of paper, allowing the programmer to ignore the issues of gravity and collision detection. Sliding joints are supported, as well as constraints on how far beams can rotate around a ball. Several functions create composite beams (e.g. t-shapes, frames, and triangles), balls can be enlarged into 'wheels', and there is a limited form of gearing to make wheels rotate at the same speeds.

Pymunk is more powerful than what's offered by linkUtils, but the functional restrictions are deliberate. I wanted to make the mechanisms as simple as possible, while retaining realistic curve drawing.

The examples in the download () are divided into four categories: simple examples that illustrate the main elements of linkUtils, examples that draw straight lines, code for drawing conic section curves, and a collection of other curve mechanisms. The following four sections describes each of these.

2. Simple Examples

circle.py draws a circle using a single ball and beam (see below on the left).



The complete code:

```
import os
os.environ['PYGAME_HIDE_SUPPORT_PROMPT'] = "hide"
from pymunk.vec2d import Vec2d
from linkUtils import *

title = "Circle"
screen = scrInit(title)
scrWidth, scrHeight = screen.get_size()
space = pymunk.Space()

coord0 = Vec2d(scrWidth/2, scrHeight/2)
```

```
coordB = coord0 + Vec2d(300, 0)

pt0 = staticBall(space, "0", coord0)
ballB = ball(space, "B", coordB)
beam0B = beam(space, pt0, ballB)

# ballB.velocity = (0,-100) # move up
pts = animate(screen, space, ballB)
plotPoints(pts, title)
```

The details of initializing Pymunk and Pygame are hidden by linkUtils.scrInit() and linkUtils.animate(). The "0" static ball is the immovable object at the center of the window, while the "B" ball can move during the animation. The balls are linked by a beam.

When the window appears, it is up to the user to 'grab' a ball or beam with the cursor and drag it to start things moving. The ballB reference passed to animate() means that the "B" ball's path is drawn as a black line during the animation. The animation can be paused/resumed by pressing the space bar, and the currently drawn path can be cleared and reset by pressing "k". Clicking on the window's close box causes the animation to terminate.

In this example, the coordinates making up the path are passed to linkUtils.plotPoints() which uses matplotlib (https://matplotlib.org/) to draw the points on a graph (see the right hand window above).

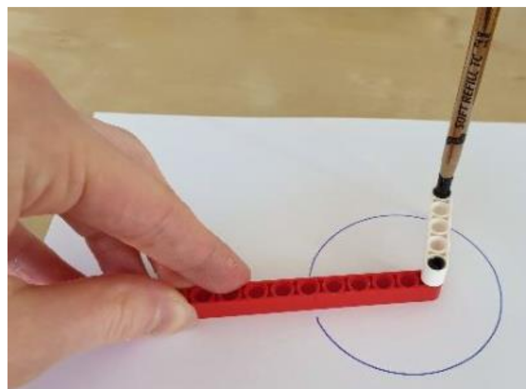
All linkUtils programs have the same structure as circle.py, with ball and beam creation sandwiched between scrInit() and animate() calls. Linkage creation typically takes four steps, with the first three being: definition of the coordinates, ball creation using those coordinates, and beam creation using the balls. The fourth stage, which doesn't appear in circle.py, is the addition of joint constraints to limit how the beams move.

circle.py includes a commented-out line:

```
# ballB.velocity = (0,-100) # move up
```

This specifies that the "B" ball should start with an upwards velocity, which means that the animation will begin with the beam rotating counter-clockwise. The coordinate system used by linkUtils is borrowed from Pygame, where the y-axis runs down the window.

A LEGO Technic implementation:

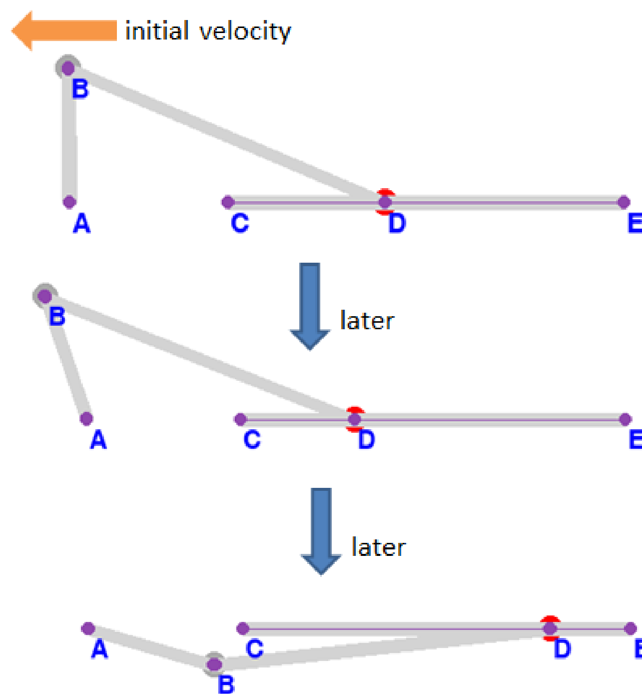


The "0" static ball is replaced by a beam that the student holds . Alternatively the ball could be held static by a blob of putty or a tack, which is arguably better since the red beam stops the circle from closing.

Even something as simple as a circle can illustrate the mismatch between the geometric description and reality. If the velocity applied to "B" is too large or directed too far from the beam's tangent, then the curve will 'wobble'.

2.1. Cranks

crank.py and **crankRocker.py** illustrate a simple use of a groove (a sliding joint). Three screenshots of crank.py are shown below:



Most of the code:

```
coord0 = Vec2d(scrWidth/2, scrHeight/2)
coordA = coord0 + Vec2d(-120,0)
coordB = coordA + Vec2d(0,-100)
coordD = coord0 + Vec2d(120,0) # end of crank shaft
coordE = coord0 + Vec2d(300,0)

ptA = staticBall(space, "A", coordA)
ballB = ball(space, "B", coordB)
ptC = staticBall(space, "C", coord0)
ballD = ball(space, "D", coordD, color=RED)
ptE = staticBall(space, "E", coordE)

beamAB = beam(space, ptA, ballB) # rotating arm
beamBD = beam(space, ballB, ballD) # crank shaft
beamCE = beam(space, ptC, ptE) # groove

groove(space, beamCE, ballD)

ballB.velocity = (-1000,0) # move left
 # -5000 or more is too fast
```

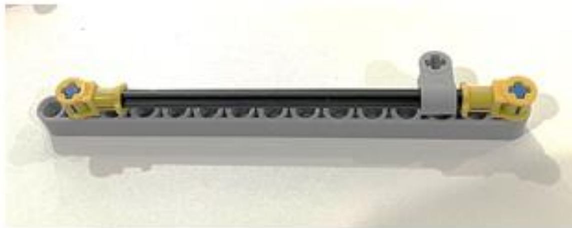
```
animate(screen, space, ballD)
```

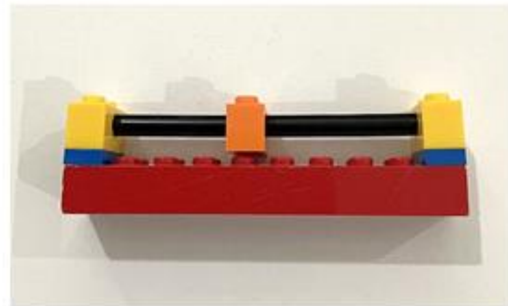
The determination of the coordinates are a little more tricky, and I encourage my students to use graph paper, defining the points relative to the center of the window (i.e. to employ coord0 in the code).

The groove is rendered as a black line along the beam CE, and the "D" ball is its 'slider'.

A line is drawn by the "D" ball as it moves left and right, but the groove obscures it in this example.

Although LEGO doesn't offer a beam with a groove, the standard way of implementing one is with an axle, as in the crank on p.1. Two more versions:





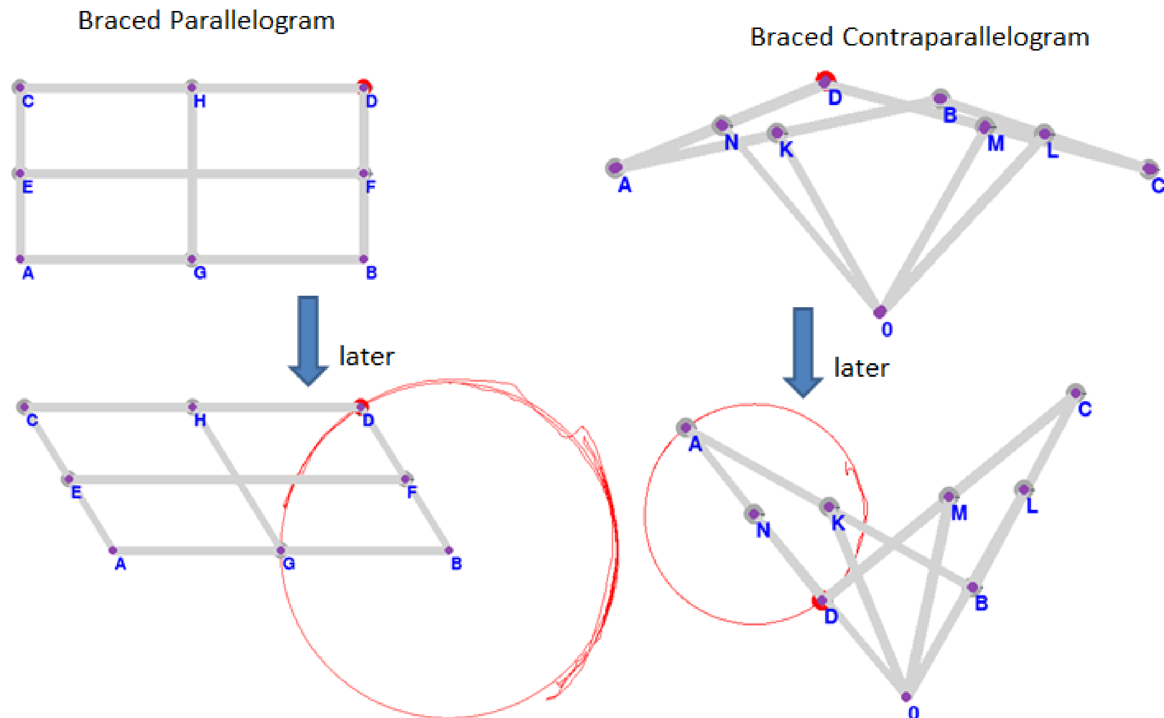
2.2. Parallelograms

The parallelogram and contraparallelogram (where the beams cross to form an "X") are the building blocks for A.B. Kempe's Universality Theorem (https://en.wikipedia.org/wiki/Kempe%27s_universality_theorem), which can be summarized by the catch-phrase:

There is a linkage that signs your name.

The idea is that any 2D curve can be traced out by a linkage made up of a mix of three basic linkages: a *translator* for translating a motion, an *additor* for adding two angles, and a *multiplicator* for multiplying an angle by a positive integer. An excellent description can be found in chapter 3 of Demaine and O'Rourke's *Geometric Folding Algorithms* [2].

Kempe admitted that "this method would not be practically useful on account of the complexity of the linkwork employed, a necessary consequence of the perfect generality of the demonstration". For example, it's quite easy for the beams in a parallelogram to inadvertently cross to change it into a contraparallelogram. In real-world mechanisms, the linkages must be braced, and it's often far from obvious what level of bracing is sufficient. **parallelogramBraced.py** and **contraParBraced.py** allow the user to experiment with different amounts of bracing by commenting in/out a few lines of code:



These examples use a new feature: the pinning of balls to beams. For instance, the "E", "F", "G", and "H" balls in the parallelogram are pinned midway along several beams:

```
pin(space, beamAC, ballE, (coordC-coordA)/2)
pin(space, beamBD, ballF, (coordD-coordB)/2)
pin(space, beamAB, ballG, (coordB-coordA)/2)
pin(space, beamCD, ballH, (coordD-coordC)/2)
```

Also, in the contraparallelogram, the ON beam is prevented from rotating around O to stop the mechanism from turning counterclockwise:

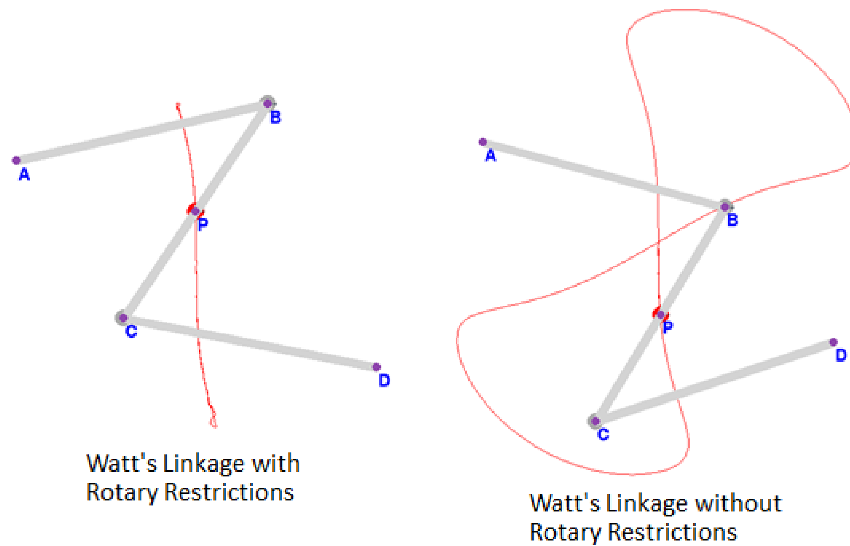
```
rotary(space, pt0, beam0N, 0,0) # no rotation
```

The beam is allowed a minimum and maximum rotation around the O ball of 0 degrees (i.e. no rotation at all).

3. Straight Line Linkages

The first (approximately) straight line drawing mechanism was developed by James Watt in the 1780s, for guiding the pistons of steam engines (https://en.wikipedia.org/wiki/Watt%27s_linkage). Mathematically perfectly straight line linkages weren't devised for nearly another 100 years, with the Peaucellier–Lipkin linkage in 1864. A great single source for LEGO builds of these mechanisms is chapter 7 of *The Unofficial LEGO Technic Builder's Guide* by P. Kmieć [4].

watts.py utilizes three beams and a pinned "P" ball at the midpoint of the central beam for drawing the line:



Watt's Linkage with Rotary Restrictions

Watt's Linkage without Rotary Restrictions

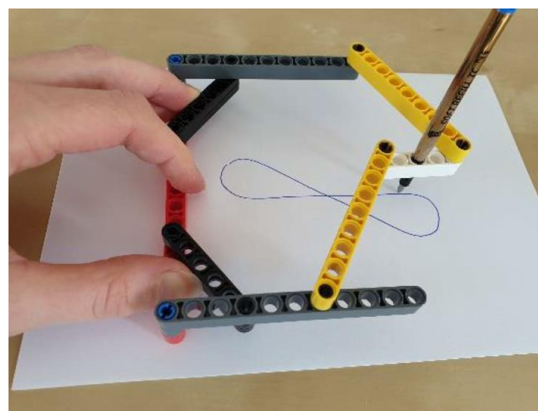
Two crucial rotary restrictions are applied to the AB beam rotating around the "A" ball and DC turning around "D":

```
rotary(space, ptA, beamAB, -45, 45)
rotary(space, ptD, beamDC, -45, 45)
```

The beams can move a maximum of 45 degrees left and right of their starting orientation .

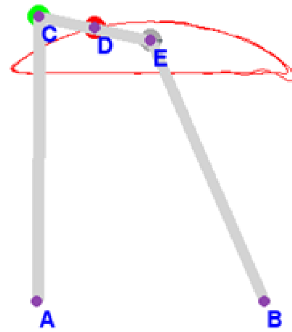
If these two lines are commented out then the linkage generates a complete lemniscate as shown in the right hand image above.

A LEGO implementation:

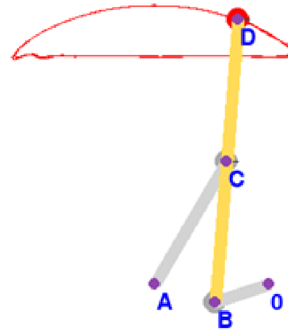


The LEGO device has drawn a lemniscate because there are no rotary limitations on its beams.

chebyshev.py implements the Chebyshev's linkage (https://en.wikipedia.org/wiki/Chebyshev_linkage) that converts rotational motion to approximately linear motion, while the Chebyshev *Lambda* linkage (https://en.wikipedia.org/wiki/Chebyshev_lambda_linkage) is demonstrated by **chebLambda.py**:

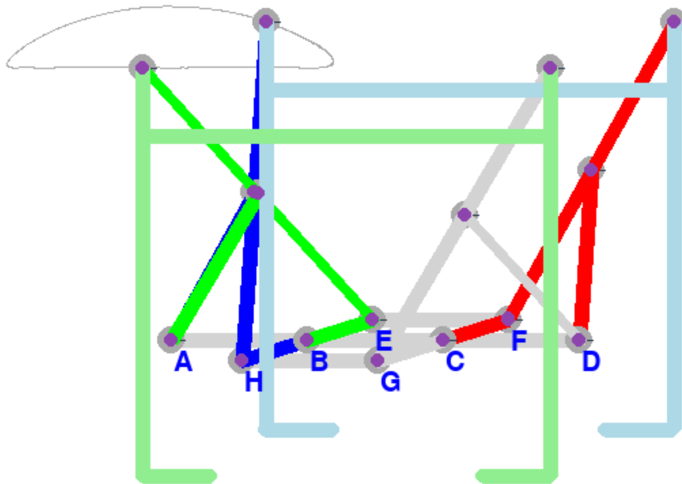


chebyshev.py



chebLambda.py

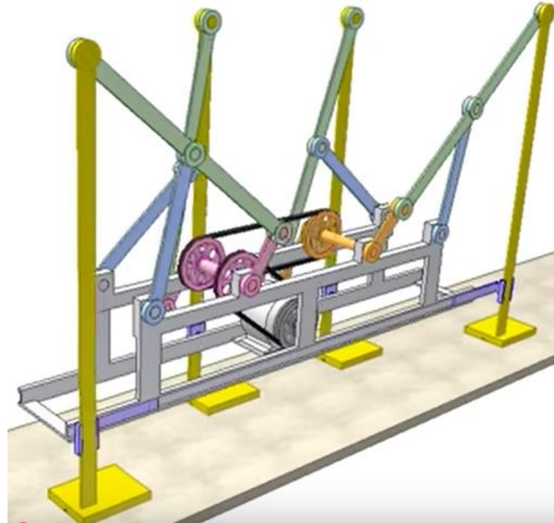
The lambda linkage can be employed as a building block for walking behavior (https://en.tcheb.ru/plantigrade-machine/), as demonstrated by **plantigrade.py**:



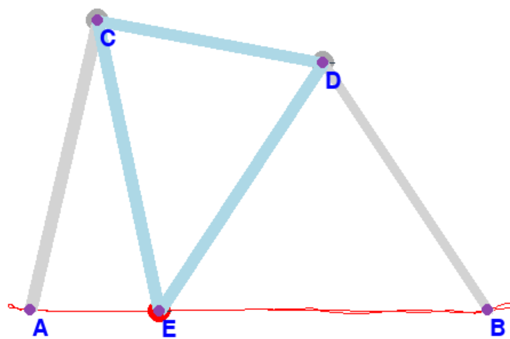
Four Chebyshev lambda linkages are driven by the rotating parallelogram EFGH. Each linkage makes a 'leg' move forward and back in imitation of a walking step. The two legs on opposite diagonals are linked by a "legs pair" shape built from five Pymunk segments. The two pairs are pale blue and green in the image above.

The starting positions for the balls were determined by drawing the device on graph paper. The parallelogram starts in a position where E, F, G, and H are collinear, which allows the balls initial velocities to be straight up for H and G and down for E and F.

This example is somewhat misleading since a real-world plantigrade is a 3D machine, as illustrated in Nguyen Duc Thang's video at https://www.youtube.com/watch?v=ISfVS4mDTKs:



roberts.py implements the Roberts linkage (https://en.wikipedia.org/wiki/Roberts_linkage), which utilizes a triangular frame:



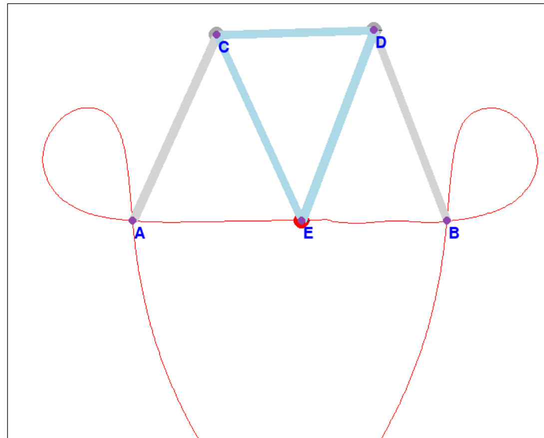
CDE is created by calling linkUtils.triangle(), supplying the three balls making up its vertices in clockwise order:

```
triCDE = triangle(space, ballC, ballD, ballE)
```

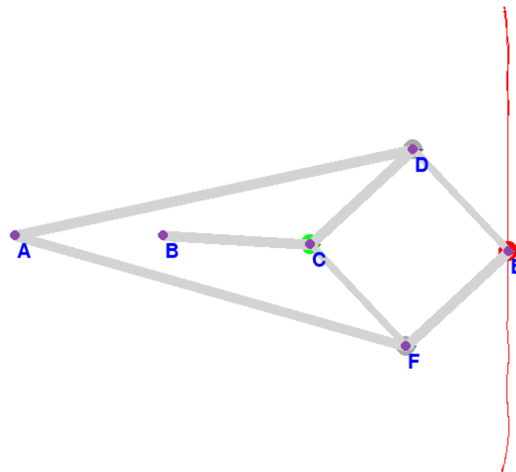
As with the Watt's linkage it's necessary to constrain the rotation of the beams to limit the curve to a straight line:

```
rotary(space, ptA, beamAC, -30, 30)
rotary(space, ptB, beamBD, -30, 30)
```

Without these, the curve becomes:



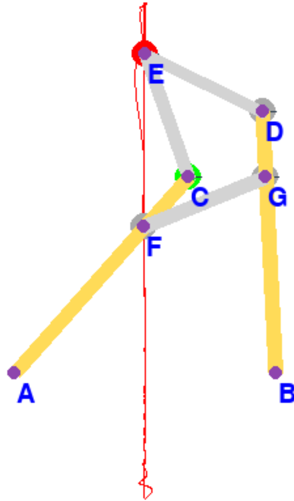
The Peaucellier–Lipkin linkage (**peaucellier.py**) was the first true planar straight line mechanism (https://en.wikipedia.org/wiki/Peaucellier%E2%80%93Lipkin_linkage), in the sense that the curve can be proven to be a straight line:



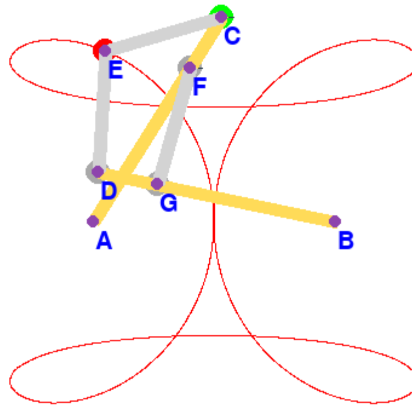
Nevertheless, in the real-world, the quality of the line is much improved if the BC beam is restricted to rotate 45 degrees around its initial horizontal orientation:

```
rotary(space, ptB, beamBC, -45, 45)
```

Hart's A-frame (**hartAFrame.py**) is shaped like a capital A constructed from a stacked trapezium and triangle (https://en.wikipedia.org/wiki/Hart%27s_inversors). It has the useful property that the motion perpendicularly bisects the base points at "A" and "B":



The problem with this linkage is that it's easily 'persuaded' to divert into a leaf-based pattern when the points momentarily become collinear when the frame passes between "A" and "B". The structure can turn left or right rather than head up or down:

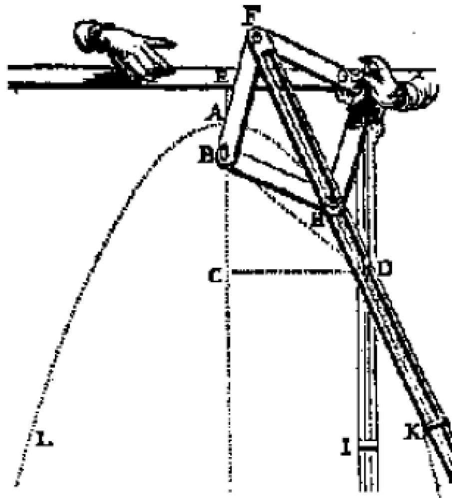


This can be mostly avoided by adding rotary constraints to the AC and BD beams:

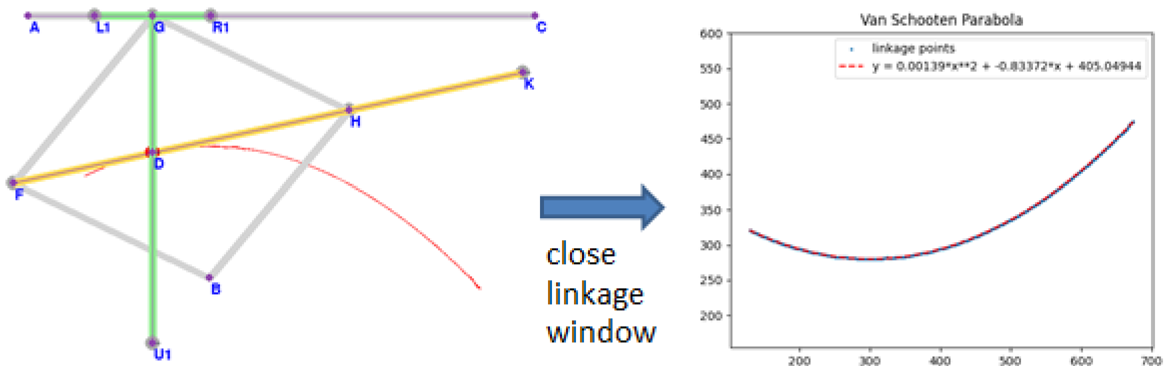
```
rotary(space, ptA, beamAC, -30, 140)
rotary(space, ptB, beamBD, -140, 30)
```

4. Conic Section Curves

Frans van Schooten (1615-1660), a Dutch professor of mathematics, translated Descartes' *La Géométrie* into Latin and wrote a popular commentary on it. In the process, he devised several instruments for drawing conic section curves (https://old.maa.org/press/periodicals/convergence/historical-activities-for-calculus-module-1-curve-drawing-then-and-now). One of his parabola linkages:



This is implemented in **parabolaVS.py**:



A major feature is that the termination of the animation (by clicking on the window's close box) is followed by a call to scipy's curve_fit() to fit a parabola to the drawn path's data points (https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html). The resulting equation is plotted over the path points by matplotlib:

```
pts = animate(screen, space, sliderD)

# collect multiple y-values for each x-coord
coordsMap = {}
for (x,y) in pts:
 coordsMap.setdefault(x, []).append(y)

# average y-values for each x, and sort according to x-values
vs = []
for key, vals in coordsMap.items():
 vs.append((key, round(sum(vals)/len(vals))))
vs = sorted(vs)
# print(*vs)

# curve fit to (x,y)'s
xs, ys = zip(*vs)
paraArgs, _ = curve_fit(parabolaEqu, xs, ys)
a, b, c = paraArgs
equStr = f"y = {a:.5f}*x**2 + {b:.5f}*x + {c:.5f}"
```

```
print("Fitted:", equStr)

# plot points and fitted curve
plt.axis('equal')
plt.scatter(xs, ys, s=2, label="linkage points")
yCurve = [ parabolaEqu(x, a, b, c) for x in xs]
plt.plot(xs, yCurve, '--', color='red', label=equStr)
plt.legend()
plt.title(title)
```

The fitted equation is:

```
y = 0.00139*x**2 + -0.83372*x + 405.04944
```

The linkage code in parabolaVS.py utilizes the linkUtils.tBeam() function to create a green t-beam. It is passed the center point of the T and the length of an arm and its trunk. It returns four values – the t-shape beam, and three balls located at its base and the ends of its arms.

```
tbeam, _, vLeft, vRight = \
 tBeam(space, sliderG, 80, 450, name="1", color=LIGHTGREEN)
```

In the code snippet above, I don't use the ball placed at the t-beam's base. However, the two arm balls (labeled "L1" and "R1" in the screenshot on the previous page) are connected to the groove running along AC.

The mustard-colored FK is also grooved, and allows balls "D" and "H" to slide along it:

```
groove(space, beamFK, ballH)
groove(space, beamFK, sliderD)
```

The most complicated groove is the one running along the t-beam's trunk (from "G" to "U1" in the screenshot), which further constrains "D":

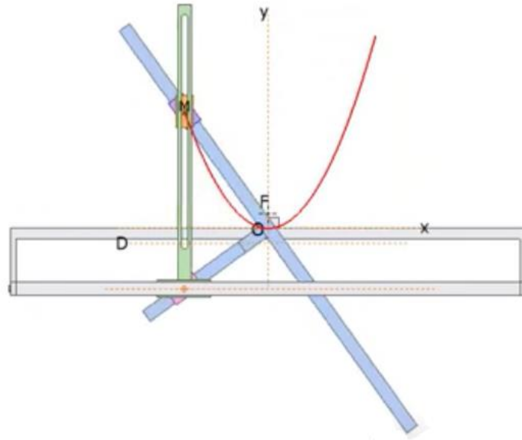
```
groove(space, tbeam, sliderD, (0,0), (0,450))
```

The two coordinate arguments of groove() specify the position of the groove on the t-beam relative to its center point at "G".

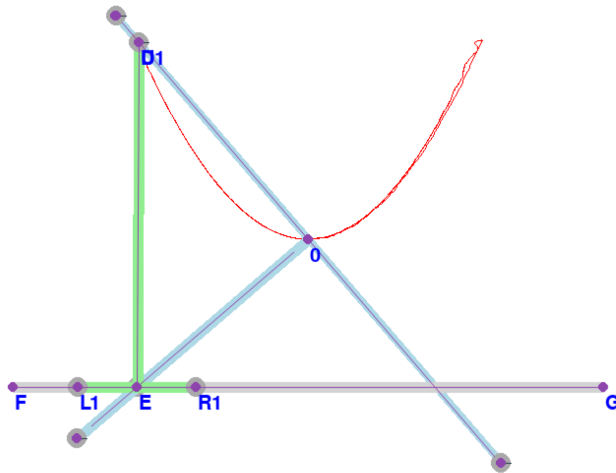
LEGO Technic contains several t-shape beams, such as:

3×3 Beam, T-Shape
60484

I recommend Nguyen Duc Thang's YouTube channel (https://www.youtube.com/@thang010146/videos) where he animates 3D models for a large range of mechanisms, including drawing devices. His parabola drawing linkage (https://www.youtube.com/watch?v=HBMYzPprNGQ) employs two t-beams:



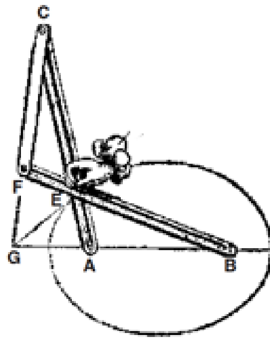
parabolaT.py implements it by calling linkUtils.tBeam() twice:



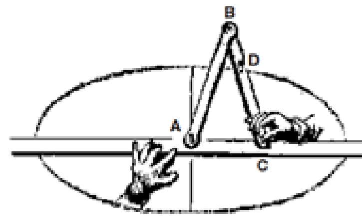
It's more complex than the van Schooten mechanism because it requires two grooves on the blue t-beam, and a groove along the trunk of the green t-beam. The relevant code:

```
# slider along blue t-beam's arm
groove(space, blueTB, sliderD, (-290,0), (290,0))

# sliderE along blue t-beam's tip
groove(space, blueTB, sliderE, (0,20), (0,290))

# attach sliderD to green t-beam
groove(space, greenTB, sliderD, (0,-350), (0,0))
```

Van Schooten designed several mechanisms for drawing an ellipse; **ellipseCT.py** and **ellipseBS.py** implement his "congruent triangles" and "bent straw" devices:

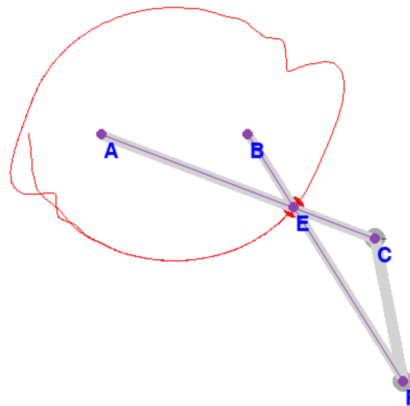


van Schooten’s Ellipse
(Congruent Triangles)

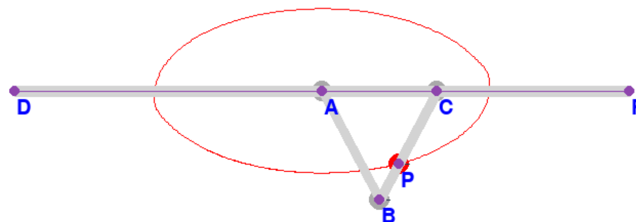


van Schooten’s Ellipse
(Bent Straw)

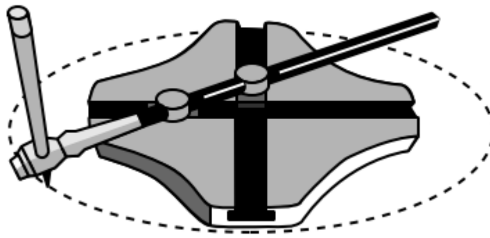
The disappointing ellipse drawn by ellipseCT.py is due to the "E" ball's instability when the AC and BF beams are nearly parallel:



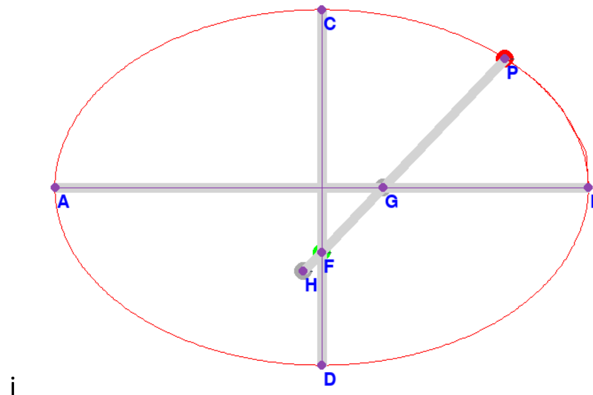
The bent straw mechanism in ellipseBS.py is much better:



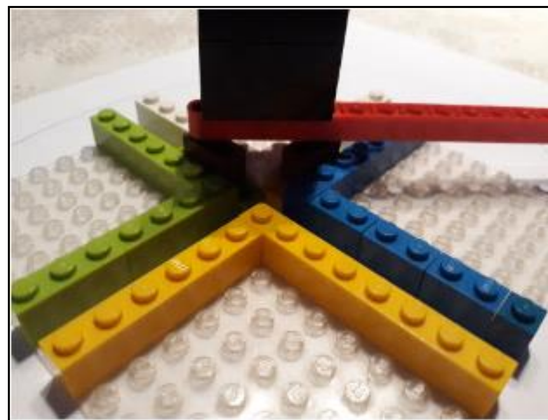
The most popular ellipse-drawing mechanism is probably the ellipsograph, or trammel of Archimedes (https://en.wikipedia.org/wiki/Ellipsograph). It utilizes two balls, linked by a beam, which move along perpendicular grooves:



trammel.py generates a very respectable ellipse, but has a hard-to-see problem:

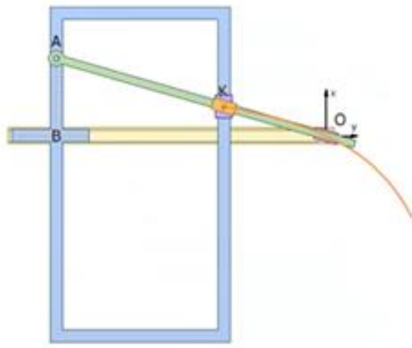


The grooves along AB and CD cross each other, which doesn't matter in the Pymunk code since there's no collision detection, but can pose a problem for LEGO:

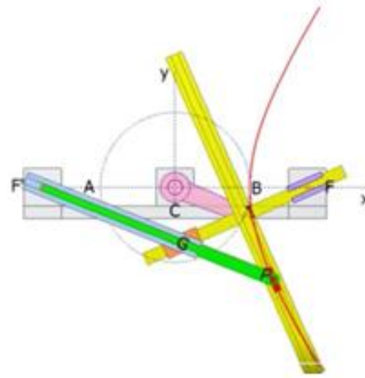


When a ball passed through the intersection, it may hit one of the brick corners it is approaching causing the drawing to wobble or even stop.

Nguyen Duc Thang's videos include several that draw hyperbolae, including one using a rectangular frame (https://www.youtube.com/watch?v=iUJmd0_7sGw), and another based around an x-shape beam (https://www.youtube.com/watch?v=rQB6abVhLo4):

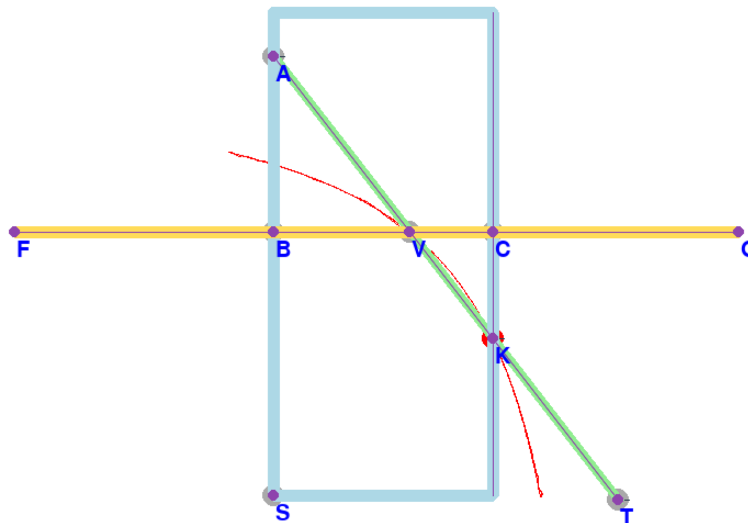


Nguyen Duc Thang Hyperbola
(Frame)



Nguyen Duc Thang Hyperbola
(X-beam)

hyperbolaFrame.py implements the frame using four Pymunk segments.



The frame() function is passed the top-left corner point of the frame as a ball, the desired width and height, and returns the structure:

```
frame = frame(space, ballS, d, -a-7*b)
```

The trickiest parts are determining the starting position of ball "A", which requires a careful graph paper diagram, and defining the coordinates of the groove down the right-hand side of the frame:

```
groove(space, frame, ballK, (d,-a-7*b), (d, 0))
```

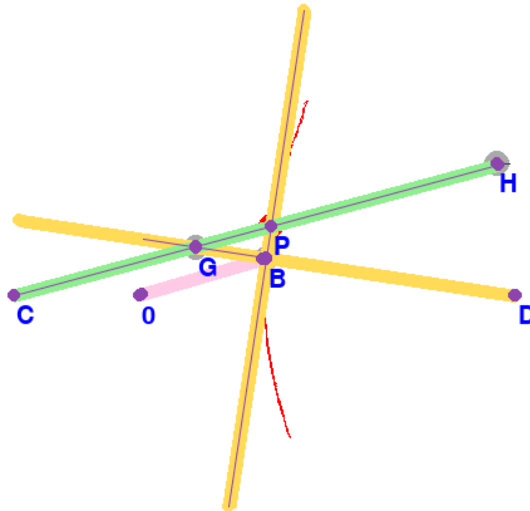
LEGO Technic offers a few different frames, such as:

However, as with other beams, there's no built-in groove.

hyperbolaX.py utilizes linkUtils.xBeam(), which is called with its center point and its arm and trunk lengths:

```
xb = xBeam(space, ballB, 200, 200) # arm & trunk lengths
```

The animation produces a very wobbly line due to the over extension of OB as the x-beam reaches the top and bottom of its swings:



LEGO Technic doesn't have an x-shaped beam, but probably the simplest way to build one is to attach two small beams to a long one with L-shaped connectors (part 55615):

Another possibility would be to use standard bricks connected by two corner plates or a cross plate:

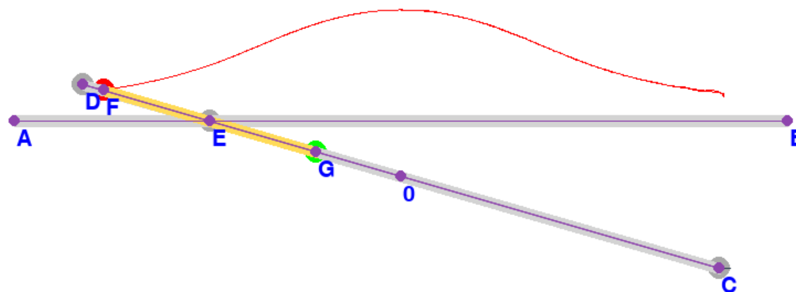
None of these have grooves, which in hyperbolaX.py run all the way up the trunk and partially along the left arm:

```
# sliderP move along the trunk of X
groove(space, xb, sliderP, (0,-200), (0, 200))

# ballG moves along left arm of X
groove(space, xb, ballG, (0,0), (-100,0))
```

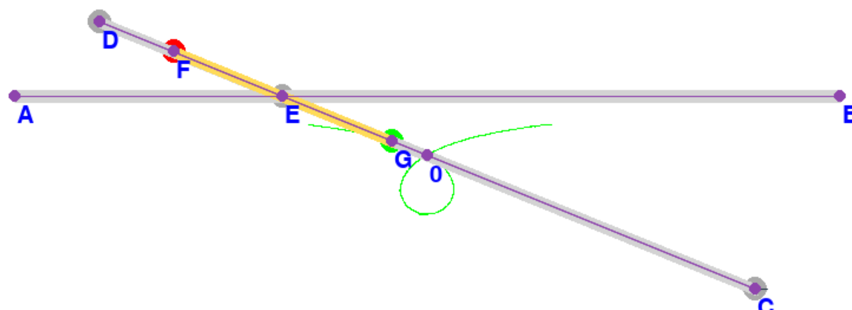
5. Other Curves

This section is about the drawing of non-conic section curves, such as the conchoid, cycloid, Lissajous curves, and spirals. As we'll see, they require two new linkUtils elements: *wheels* (oversized balls with pinned extras and angular velocity), and *couplers* (two wheels constrained to rotate at the same speed and linked by a beam).

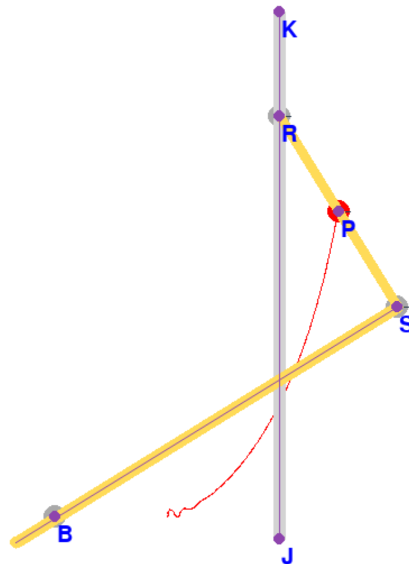
conchoid.py implements the Conchoid of Nicomedes (https://en.wikipedia.org/wiki/Conchoid_(mathematics)):



For every position of line DC rotated through O and intersecting AB at E, the two points F and G, always at some fixed distance from E, lie on the conchoid. The image above shows the path traced by F. If the code is edited to draw G's path instead, the result is:

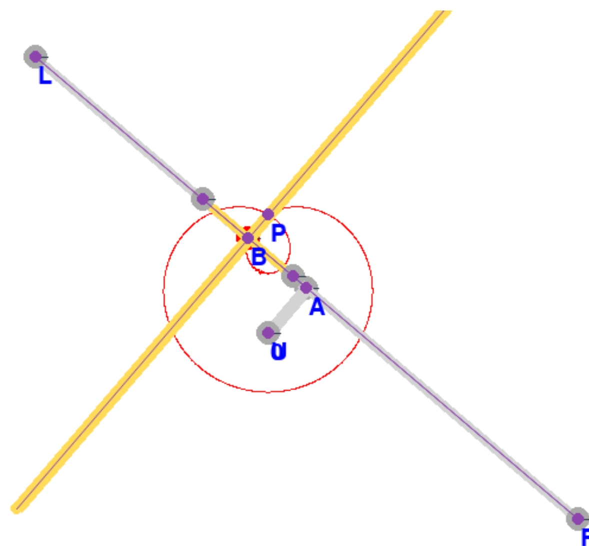


cissoid.py uses Newton's carpenter's square construction to draw the upper branch of the cissoid of Diocles (https://en.wikipedia.org/wiki/Cissoid_of_Diocles#Newton's_construction):



The carpenter's square makes an appearance in several curve drawing mechanisms, so is implemented by linkUtils.lBeam(). It takes the corner point (e.g. "S" in the example above), and the length of the arm and trunk. In a cissoid, the l-shape is constrained by the balls "R" and "B". "R" is fixed to the end of the arm but can slide up and down KJ. "B" is fixed in space and constrains the l-beam by being in the groove on its trunk.

limacon.py draws the limaçon of Pascal, by treating it as a pedal curve for a circle (https://en.wikipedia.org/wiki/Lima%C3%A7on#Relation_to_other_curves).

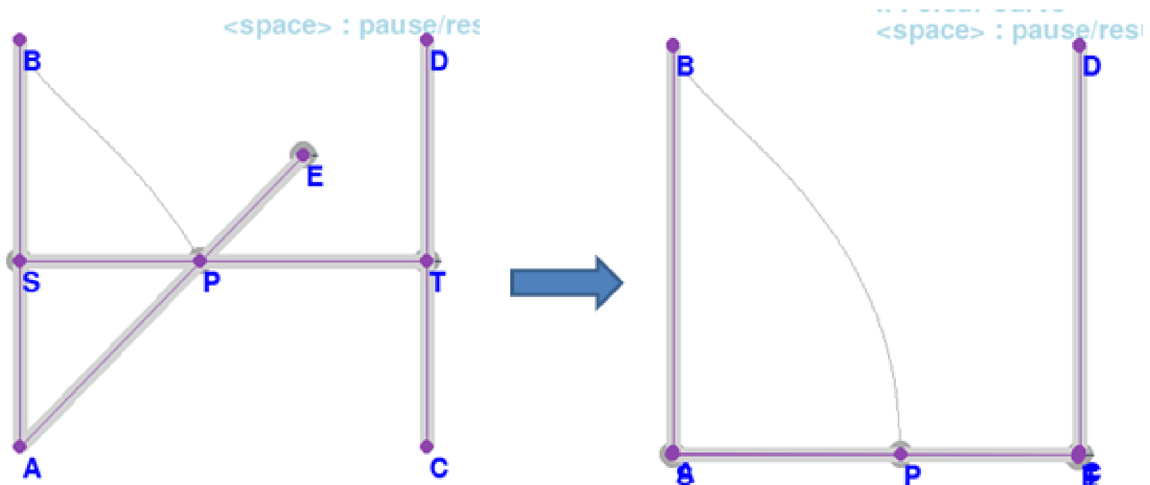


The circle is implemented by a t-beam that rotates clockwise. An x-beam is attached to the arms of the t-beam, so that RL is a tangent to the t-beam's circle. The x-beam trunk is

grooved and holds the static "P" (the pedal point). The center of the x-beam, "B", is used to draw the curve.

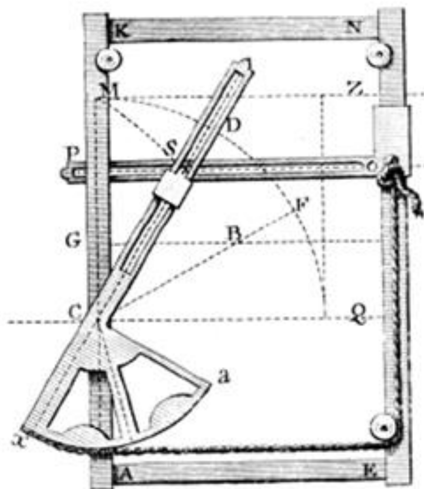
The limaçon is more commonly viewed as a roulette curve, which means that it could also be drawn using a *wheel*, in a similar way to the cycloid in section 4.1 below.

quadratrix.py draws the quadratrix of Hippias (https://en.wikipedia.org/wiki/Quadratrix_of_Hippias):

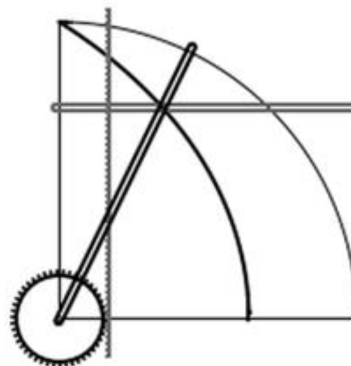


The curve is traced out by the crossing point "P" on two beams (ST and AE), the former falling at a uniform speed, and the other rotating uniformly around "A". The speeds should be such that the two beams reach the bottom at the same time.

Two ways to draw a quadratrix are to use a string/chain mechanism or a cogwheel and rack:



Suardi (1752) Quadratrix
(String-based)



Quadratrix
(Cogwheel and Rack)

Neither of these would produce good results in Pymunk since they require collision detection, either to ensure that the chain stays tight, or the cogwheel stays in the rack. Section 5 discusses a simple chain drive coded in Pymunk.

The only approach that produces reasonable results is to implement the speeds constraint by using Pymunk's velocity callback functions. A function is attached to a body at initialization time and then called automatically at each time step.

Before the animation loop is entered, the linear velocities for balls "S" and "T" are set, as well as the angular velocity for beam AE:

```
ballS.velocity = LIN_VEL # move down
ballS.velocity_func = velPos
ballT.velocity = LIN_VEL # same as S
beamAE.angular_velocity = ANG_VEL
```

The velocity_func field of "S" is assigned the callback function:

```
def velPos(body, gravity, damping, dt):
 # attached to ball S
 # keep velocities and positions in sync
 pymunk.Body.update_velocity(body, gravity, damping, dt)
 body.velocity = LIN_VEL # velocity of S
 ballT.velocity = LIN_VEL
 beamAE.angular_velocity = ANG_VEL

 # keep y positions of S and T the same
 _, y = body.position # of S
 xT, _ = ballT.position
 ballT.position = (xT, y)
```

The function’s arguments are used to call the standard update_velocity() method for the body (i.e. for ball "S"). In addition, "S","T", and AE are re-assigned their initial velocities, and the vertical positions of "S" and "T" are kept the same.

This approach is far from ideal because it has no obvious physical equivalent that could be built using LEGO. Also, it isn't that great a solution anyway since the resulting curve doesn't look much like a quadratrix. One issue is that I had to basically guess values for LIN_VEL and ANG_VEL:

```
LIN_VEL = (0, 80) # move down
ANG_VEL = 0.14*math.pi # cw
```

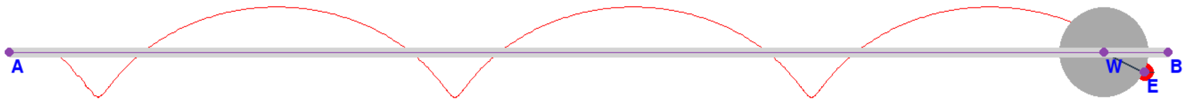
If ANG_VEL Is even slightly smaller or larger than the curve changes shape drastically.

4.1. Wheels

Many curves are drawn by tracing out the path when a curve is rolled along another curve without slipping. The most common types of these roulette curves are when a circle is rolled over a straight line or another circle. This is achieved in linkUtils by using wheels.

A wheel is a ball with a larger radius, initialized with some angular velocity, and usually with a drawing ball attached to its rim. This drawing ball is assigned a very small mass so its rotation doesn't affect the wheel.

cycloid.py draws a cycloid (https://en.wikipedia.org/wiki/Cycloid) – the curve traced by point "E" on the wheel as it rolls along the groove in AB:

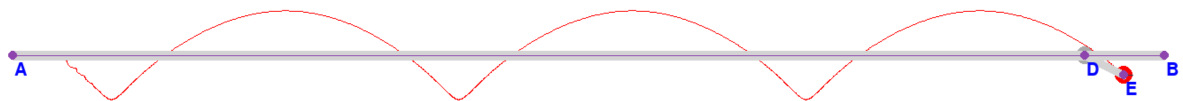


The wheel and "E" are defined using:

```
wheel = ball(space, "W", coordA, radius=r)
beamAB = beam(space, ptA, ptB)
groove(space, beamAB, wheel)

ballE = ball(space, "E", coordE, color=RED)
ballE.mass = 0.00001 # so will not affect wheel when pinned
pin(space, wheel, ballE, (drawPos, 0))
# right edge of wheel; will be off the edge if drawPos > r

wheel.velocity = (100,0) # move right
wheel.angular_velocity = 0.5*math.pi # turns clockwise
```

It's quite possible to position "E" closer to the wheel's center or beyond its rim.

cycloidBeam.py uses a DE beam to implement rotation, which produces a similar curve, but only with different velocity settings.

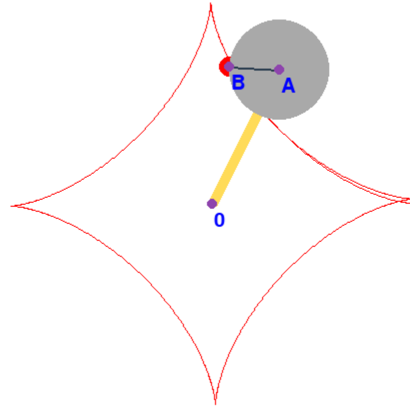


The wheel is assigned both linear and angular velocity (see above), while for the beam, the linear velocity is assigned to the center point "D", and the angular velocity to DE:

```
ballD.velocity = (100,0) # move right
beamDE.angular_velocity = 0.65*math.pi # turns clockwise
```

The angular speed of DW is greater compared to the value used by the wheel

hypocycloid.py traces a fixed point "B" on a wheel "A" that rolls within a larger circle (https://en.wikipedia.org/wiki/Hypocycloid). However, I’ve implemented this circle as a rotating beam OA, attached to the wheel's center:



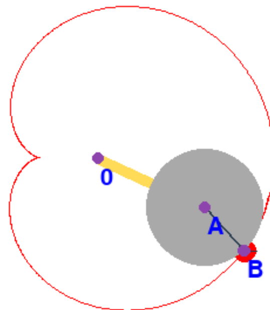
The wheel's velocities:

```
wheel.velocity = (0, -100) # move up, so beam turn ccw
wheel.angular_velocity = 0.27*math.pi # clockwise
```

No initial velocity is assigned to OA, but the wheel's upward velocity will make it turn counterclockwise anyway. Meanwhile the wheel turns clockwise around "A".

This is sufficient to almost generate an astroid (https://en.wikipedia.org/wiki/Astroid).

cardioid.py draws an epicycloid with one cusp (https://en.wikipedia.org/wiki/Cardioid). An epicycloid (also sometimes called an hypercycloid, somewhat confusingly) traces the path of a point "B" on the circumference of a wheel "A" which is rolling over the surface of another circle:



Just as in the previous example, I implement the circle as a beam, but now OA rotates *clockwise* (it was counter-clockwise for the hypocycloid), but the wheel still rotates clockwise around "A":

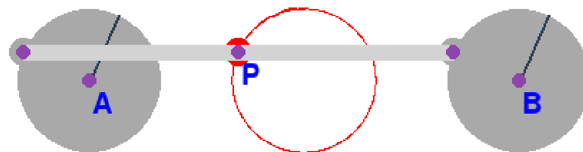
```
wheel.velocity = (0, 100) # move down, so beam turns clockwise
wheel.angular_velocity = 0.27*math.pi # clockwise
```

4.2. The Coupler

Implementing a quadratrix was problematic because it requires that linear and angular velocities stay synchronized. However, if only the angular velocities of two bodies need to be linked at some fixed ratio then Pymunk offers the GearJoint constraint (https://www.pymunk.org/en/latest/pymunk.constraints.html). linkUtils.py simplifies the constraint's creation via a geared() function:

```
def geared(space, b1, b2, ratio=1):
 # default is to use the same gearing for both bodies
 joint = pymunk.GearJoint(b1, b2, 0, ratio)
 joint.collide_bodies=False
 space.add(joint)
```

In practice, what seems more useful is a beam that can move forwards and backwards at a constant speed. This can be implemented using two wheels which have synchronized angular velocities and are joined by that beam; **trainCoupler.py** illustrates the idea:

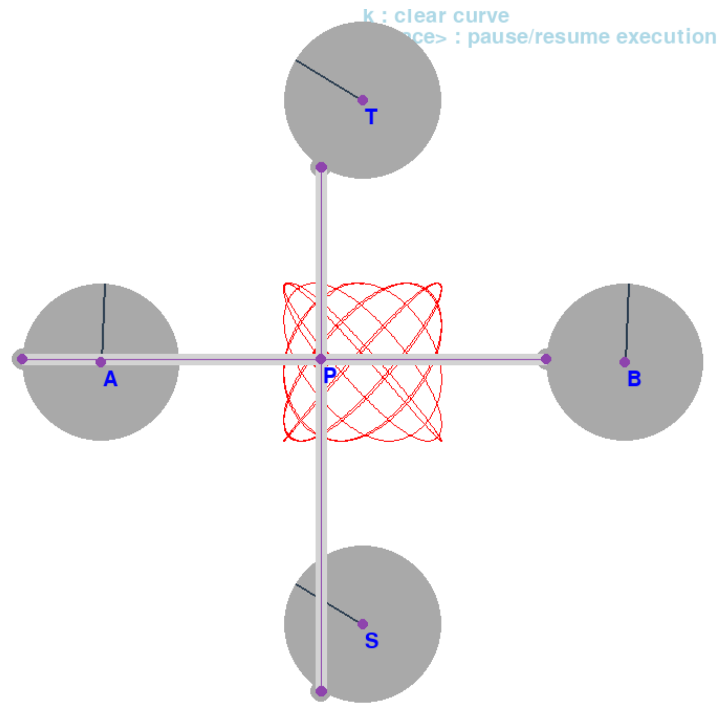


linkUtils.coupler() takes as arguments the centers of the two wheels, their radius, and the offset of the pinned beam ball relative to the center. The function returns the linked beam, and the balls at its two ends:

```
ptA = staticBall(space, "A", coordA)   # fix the points
ptB = staticBall(space, "B", coordB)
offset = Vec2d(0, -r) # starts above the center
beam, ball1, ball2 = coupler(space, ptA, ptB, r, offset)
```

In trainCoupler.py, a drawing point "P" is pinned to the beam's midpoint, and the mechanism is set moving by giving the left hand wheel (ball1) an initial velocity.

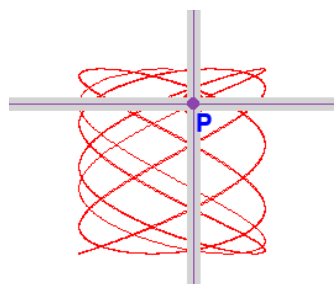
A good use for a coupler is to implement Lissajous curves (https://en.wikipedia.org/wiki/Lissajous_curve), which combine oscillations in the x- and y-directions. **lissajous.py** utilizes two couplers, one for each axis, and grooves are added to the couplers' beams, and a drawing point ("P") placed in both grooves:



The crucial parameters are the speeds of the wheels. The vertical beam is set in motion by giving its wheels an upwards velocity, and the horizontal beam is started rocking by assigning its wheels the same velocity but to the left:

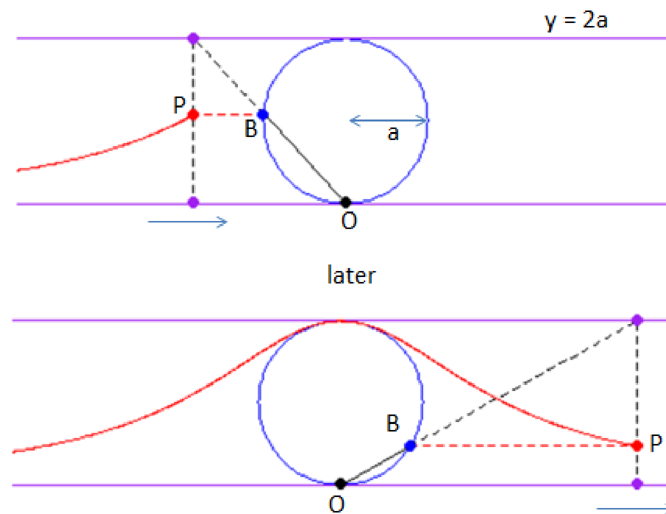
```
velCD = -200 # try -100
ballC.velocity = (velCD, 0) # move up initially
ballD.velocity = (velCD, 0)

velVW = -200 # try -100
ballV.velocity = (velVW, 0) # move left initially
ballW.velocity = (velVW, 0)
```

This makes the relative frequencies of the two beams the same, resulting in a curve with the same number of peaks along both axes. If one of the velocities is changed, the relative frequencies change, affecting the curve. For instance, if the upward velocity is halved then the outcome is:

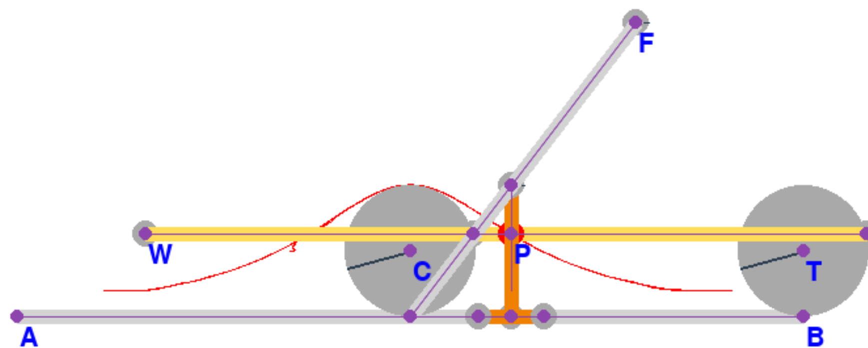


The "witch of Agnesi" is a curve first studied by Maria Agnesi in 1748 (https://en.wikipedia.org/wiki/Witch_of_Agnesi). It can be obtained by rotating a line OB attached to the origin over a circle of radius a, centered at (0,a), The y-coordinate of the

drawing point "P" is set to be the same as "B"s where it intersects the circle. "P"s x-value comes from the point where the extension of OB meets the line y=2a:



The hardest part of implementing **agnesi.py** is ensuring that BP always stays horizontal. This is achieved by adding another wheel of the same radius and linking them as a coupler:



The coupler is defined as:

```
_, ballV, ballU = coupler(space, ptC, ptT, a, Vec2d(-a, 0))
```

The beam's endpoints are utilized to create the longer mustard-colored beam (beamWU) visible in the screenshot:

```
coordW = coordC + Vec2d(-300, 0)
ballW = ball(space, "W", coordW)
beamWU = beam(space, ballW, ballU, color=MUSTARD)
```

The line up through "P" must also be kept vertical at all times, and this is done with an orange t-beam that slides along AB:

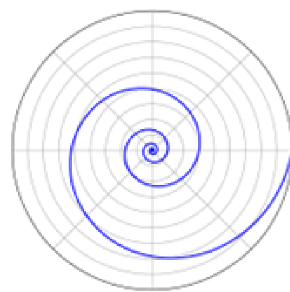
```
tb, ballUp, ballL, ballR = \
 tBeam(space, ballE, 0.5*a, -2*a, color=TANGERINE)
```

ballUp is located at the top of the t-beam, and is attached as a slider to OF. Balls "L" and "R" on the arms of the t-beam become sliders on AB:

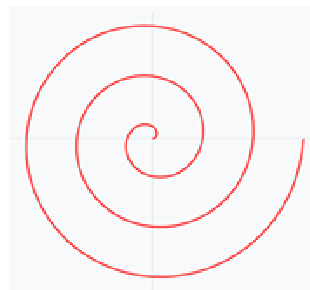
```
groove(space, beamF0, ballUp) # the top of the inverted t-beam
groove(space, beamAB, ballL)
groove(space, beamAB, ballR)
```

4.3. Spirals

There are many kinds of spiral (e.g. see https://en.wikipedia.org/wiki/Spiral), but I'll focus just on the logarithmic and Archimedean ones. At first glance they look fairly similar but the distances between the turns of the logarithmic spiral increase in a geometric progression, while the distances in an Archimedean remain constant:



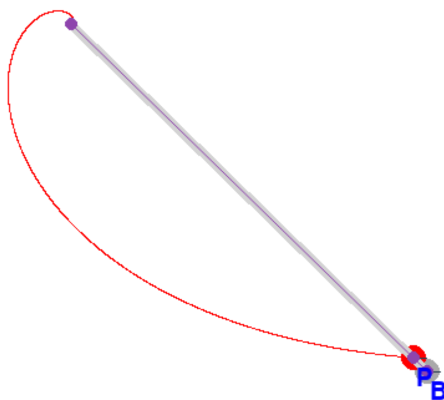
A Logarithmic Spiral
$r = ae^{k\varphi}$



An Archimedean Spiral
$r = a\varphi$

Also, as we'll see, their simulation code is quite different.

logSpiral.py employs a long beam OB which rotates around the origin. Drawing ball "P" is attached to the beam as a slider, and initially located at the origin. As the beam rotates, "P" picks up speed and moves out to the end of the beam:

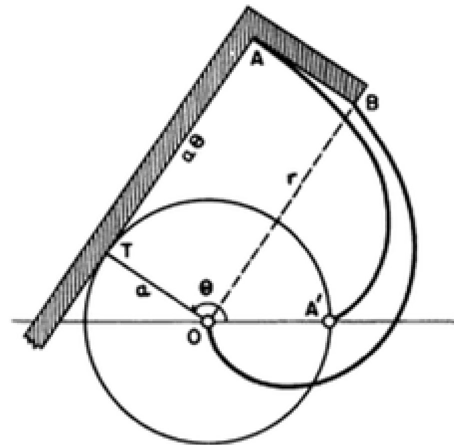


One issue with this approach is that it takes almost half a rotation of OB before "P" noticeably starts sliding away from the origin. Also, it seems almost impossible to 'tighten up' the spiral by adjusting the ball mass or velocity.

The relevant lines of code:

```
beam0B = beam(space, pt0, ballB)
groove(space, beam0B, ballP)
ballB.velocity = (0,-100) # move up
```

Yates [8] employs a carpenter's square rolling over a circle to draw an Archimedean spiral. One advantage of this is that by changing the drawing point from "B" to "A" the device can trace out the involute of a circle instead:

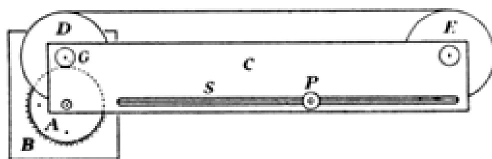


A: Circle Involute
B: Archimedes Spiral
Yates (Carpenter's Square, p.209)

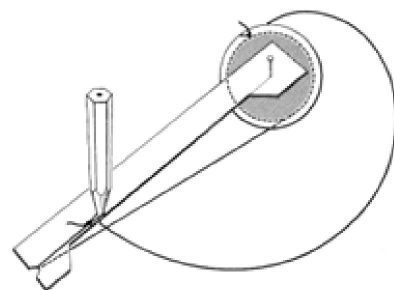
OT = AB = a, the radius, and "A" start at A', while "B" begins at O. AT is then equal to the arc A'T = r = aΘ. Note that the center of rotation is at "T" so TA and TB are normals to the paths of "A" and "B".

The l-beam must turn in such a way that "B" moves away from O at a constant speed along a line that also rotates with constant velocity. This requires that these linear and angular speeds stay synchronized, posing the same problem that we encountered with the quadratrix.

It's interesting to note that both the quadratrix and the Archimedean spiral have been used to solve the classical problem of squaring the circle. Another link between them are the possibility of using chains/strings to generate their curves. Two approaches [1; 3] for drawing the Archimedean:

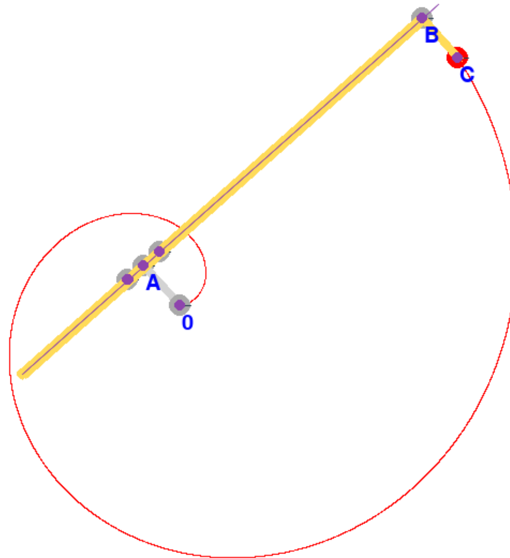


Cundy and Rollett, p.244



Gardner, p105

The answer to the speeds synchronization problem is the same as for the quadratrix – a velocity callback function. **archSpiralL.py** generates the curve:



The inner circle is encoded as a t-beam that rotates counter-clockwise around O, and a mustard-colored l-beam is attached to its arms, acting as a tangent.

The callback needs to maintain the speed of ball "B" (which gradually pulls the l-beam along the t-beam's arms) and the t-beam’s angular velocity.

These two velocities are initialized before the animation begins:

```
# set the linear and angular velocities
ballB.velocity = (LIN_VEL,0) # move right
ballB.velocity_func = limit_velocity
tangentBeam.angular_velocity = ANG_VEL
```

tangentBeam is the name of the t-beam.

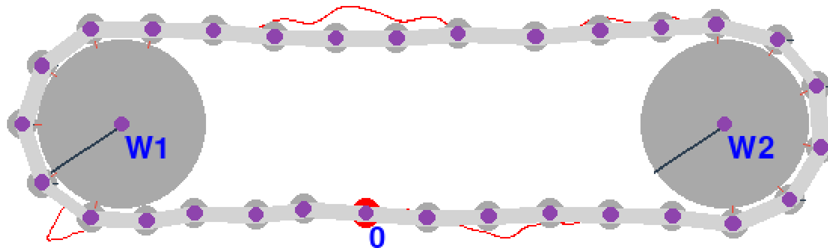
The velocity function is:

```
def limit_velocity(body, gravity, damping, dt):
 # keep velocities near to their initial values
 pymunk.Body.update_velocity(body, gravity, damping, dt)
 vMag = body.velocity.length # for ball B
 tangentBeam.angular_velocity = ANG_VEL
 if vMag > LIN_VEL:
 scale = LIN_VEL / vMag
 body.velocity = body.velocity * scale
```

This callback is more complex than the function used for the quadratrix since "B" isn’t moving in a constant direction; it's velocity is directed along the long edge of the l-beam which is rotating. The standard Pymunk solution is to ensure that the linear velocity's magnitude stays close to its initial value, while allowing its trajectory to change.

5. Chains

A mechanism that keeps turning up in curve drawing is the chain or string. Unfortunately, it's unsuitable for linkUtils since it requires collision detection. But even if this restriction is lifted, it's still very time-consuming and tricky to create a viable chain simulation. This is illustrated by **chain.py**, which implements a chain rotating around two wheels:



Ball "O" is the drawing point, and its erratic path can be partially seen behind the chain.

One difficulty is determining the initial coordinates for the balls that make the beams in the chain. They should all be about the same length and also tightly fit around the wheels.

The simplest way to handle the collision detection is to only switch it on for the two wheels. Indeed, if collision detection is enabled between the beams then it becomes impossible to link them together into a tight chain.

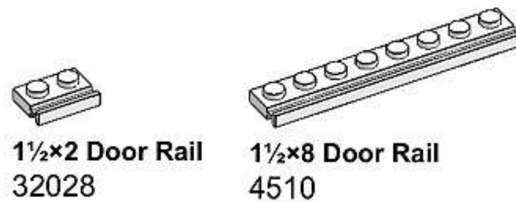
Another issue is that the presence of collision detection causes the velocity of the chain to decrease rapidly, so we need to add a callback function to maintain the speed.

6. Some Thoughts

The aim of this work was to develop a simple library for building simulations of curve drawing mechanisms which better reflect the kinematic issues when these mechanisms are built in the real-world. In my case, this means having high-school kids, and younger, construct linkages using LEGO Technic pieces.

There's a functional mismatch between the tools that Pymunk and my linkUtils module offer and the capabilities of LEGO. The most obvious difference on the LEGO side is the lack of a simple prismatic joint. Using an axle, as below, is usually adequate but results in a beam that is twice as thick as normal:

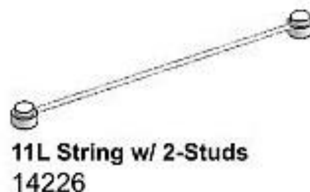
It would be much easier to construct sliding mechanisms if there was a LEGO beam with a built-in groove down one side. It could be designed to accept the existing LEGO door rail as a slider:



The issue with the thickness of slider beams is a symptom of a larger problem – while linkUtils beams and balls can happily pass through each other without colliding, this is obviously not the case with LEGO pieces.

I'm not suggesting that collision detection be added to linkUtils. It would needlessly complicate most simulations, and this mismatch between simulation and reality is actually a useful opportunity to discuss the differences between the two approaches during a class.

LEGO Technic offers a wide range of tools based around gears and chains which would make some curve drawing devices easier to build. While Pymunk simulations of these is possible, their implementation is tricky and the resulting simulations quite poor. Pymunk needs to offer something less complex than a chain made from segments as I employed in chain.py. My current opinion is that something like a LEGO string might be easier to work with:



6.1. An Implementation Hierarchy?

The most interesting outcome of this work is the beginnings of a hierarchy of shapes and joints required for building curve drawing linkages.

Kempe's universality theorem (https://en.wikipedia.org/wiki/Kempe%27s_universality_theorem) implies that beams and balls (rotational joints) are sufficient for any problem, but this is akin to saying that the control flow of any program can be implemented with conditional GOTOs only.

The addition of sliders (prismatic joints) simplifies most designs, and rotary constraints and composite beams (e.g. T, L, and X- shapes) improve the quality of the generated curves.

The next step up in complexity are mechanisms that need speed synchronization between their elements. This requires a form of gearing constraint, as in linkUtils' geared() and coupler() functions. Pymunk's GearJoint() constraint only synchronizes angular velocity, but it can be used to build linear constraints, albeit in a rather indirect way. A simpler approach is to use Pymunk's velocity callback functions, but these have no obvious real-world equivalents.

The fourth stage of functionality seems to require collision detection to implement the movement of string and chains around cogwheels and/or pulleys. My view is that the addition of a LEGO-like string to Pymunk may be sufficient for these use cases.

References

[1] Cundy, H.M., and Rollett, A,.P., 1961, *Mathematical Models*, OUP, 2nd ed., https://archive.org/details/MathematicalModels-

[2] Demaine, E., and O'Rourke, J., 2007, *Geometric Folding Algorithms*, Cambridge University Press, pp. 31–40.

[3] Gardner, M., 1991, *The Unexpected Hanging and Other Mathematical Diversions*, Univ. of Chicago Press, ch. 9, "Spirals"

[4] Kmieć, P., 2016, *The Unofficial LEGO Technic Builder's Guide*, 2nd ed. No Starch Press

[5] Kovács, Z. & Kovács, B. 2017. "A compilation of LEGO Technic parts to support learning experiments on linkages", https://arxiv.org/abs/1712.00440

[6] Scher, D., 1995, *Exploring Conic Sections: With the Geometer's Sketchpad*, Key Curriculum Press

[7] Taimina, D., 2007, "Historical Mechanisms for Drawing Curves, in *Hands on History A Resource for Teaching Mathematics*, MAA, pp. 89-104., https://ecommons.cornell.edu/server/api/core/bitstreams/a9783fdf-8167-47ef-8fd4-2561fd2ede00/content

[8] Yates, R.C., 1952, *Curves and their Properties*, The National Council of Teachers of Mathematics, https://archive.org/details/curvestheirprope0000robe