Hypotrochoids: iOS Application and Laser Project

A Report of Senior Study

Maryville College

Musa Nami Falhan

May 2016

Acknowledgments

I would like to express my gratitude to my supervisor Dr. Jesse Smith. Without his help, I would not have been able to derive an equation for the hypotrochoid curves and understand the periodicity of the functions. His comments not only improved the quality of this thesis' content of this thesis, but it also helped me structure it better. He was always willing to help me, and he supported me to the fullest.

I also would like to thank my greatest friend Marie Lamblin who proofread my work and corrected all of my grammar mistakes. My thesis is clearer as a result of her careful annotations, and her suggestions helped me communicate my ideas better.

Finally, I would like to thank my advisors Dr. Maria Siopsis and Dr. Jennifer Bruce who gave me the idea of creating an iOS Spirograph Application.

Abstract

The purpose of this thesis is to comment on the process of making an iOS Application that mimics the Spirograph toy electronically. The project focuses on the hypotrochoid roulette curves since these are the curves drawn by the simplest version of the toy. To program our application for iPad Air 2, we used the Swift language of Apple, as well as scholarly sources and online tutorials as guidelines.

This thesis is divided into 4 chapters: 1. The Spirograph Toy, 2. The iOS Application, 3. The Laser Spirograph, and 4. Conclusions. The first chapter covers the history of the Spirograph toy and mathematical concepts surrounding it, such as a parametric equation and the period. It also summarizes several practical uses of the hypotrochoid curves. The second chapter dwells on our iOS application writing experience. The next chapter explores the making of a laser project that creates illusions of hypotrochoid curves using mirrors, motors, and a laser. Finally, the last chapter brings an end to this thesis by reflecting on the entire process.

This project enabled me to apply what I have learnt during my education in Maryville College and gave me the opportunity to add another skill that I have been aspiring to gain.

Contents

Chapter 1

The Spirograph Toy

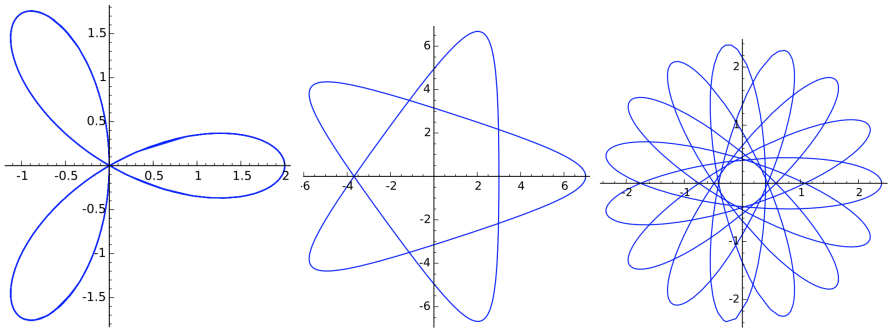
1.1 What is the Spirograph toy?

Spirograph drawing sets successfully won the hearts of many children in the 1960s, and its popularity is still expanding today. The Spirograph toy proves that mathematics can be fun and beautiful at the same time. This chapter will focus on explaining how the Spirograph toy works with mathematical concepts.

In more technical terms, a Spirograph is a toy used to create special roulette curves called hypotrochoids and epitrochoids. We can see some examples of these curves in Figures 1.2 and 1.3, both made with SageMathCloud.

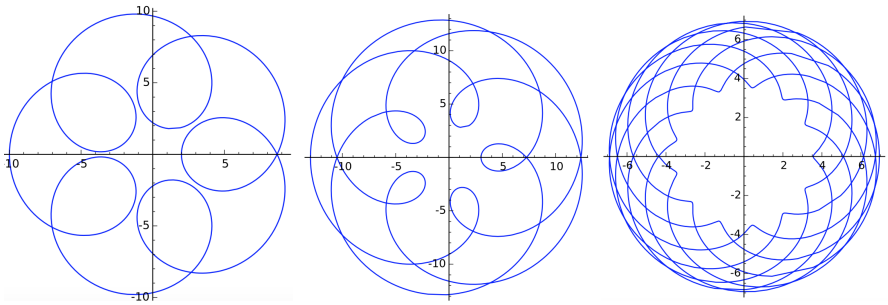
(a) **(b)** **(c)**

Figure 1.1: Today's Spirograph Set Amazon.com/Kahootz (2015).



(a) (b) (c)

Figure 1.2: Sample hypotrochoid curves The Sage Developers (2015).



(a) (b) (c)

Figure 1.3: Sample epitrochoid curves The Sage Developers (2015).

To have this result using the Spirograph toy, we set a pen in one of the holes of the small circle which will trace the curve as it rotates around the big circle.

Before the Spirograph toy was invented, we could use a mechanical kit made by Wondergraph to create these roulette shapes Paget (2014). Figure 1.4 shows an old Wondergraph kit. We can see from the design illustrations that a Wondergraph has three circular disks attached on a board, which are connected with a rubber band, as well as a ruler. The pencil is secured on the ruler that connects the middle disk to the secondary disk, and the third disk is used to rotate the mechanism. This can be seen in Figure 1.5 Tuck (1913).

F. E. Tuck explains how to build a Wondergraph drawing set in several publications, including *The Boy Mechanic Book I* Tuck (1913). Wondergraph kits

required a more hands on approach, and they were more accessible because people could build one with materials that they would already have at home.

Figure 1.4: Wondergraph Drawing Kit Kaveney (2004).

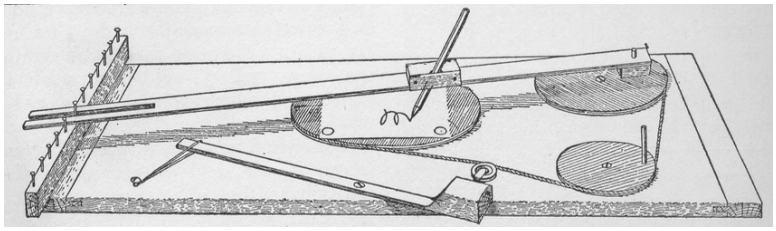


Figure 1.5: Wondergraph Drawing Kit illustration Tuck (1913).

Wondergraph kits were first created around 1908. Then, English mechanical engineer Denys Fisher introduced a better solution to draw hypotrochoid graphs Paget (2014). Originally, Fisher's tool-set was intended to draw drafts of sine and cosine curves, but he improved upon his idea and released the tool as a toy during The Nuremberg International Toy Show in 1965 Paget (2014). The toy gained popularity for its ease of use and creativity. There were also various sizes of wheels that could create many different designs Paget (2014). It became an instant hit, and in 1967, it was elected "Toy of the Year" by The British Association of Toy Sellers. At the same time it was one of the top-selling toys in the U.S.A. Coope (2015). Kenner Products acquired the rights of the toy and released various editions of it such as the Super

Spirograph (with additional pieces), the Spirotot (targeting younger population), the Magnetic Spirograph, the Spiroman, and the refill sets Coope (2015). Today, Kahootz and Hasbro are still selling Spirograph Kits Coope (2015).

Compared to the Wondergraph, the Spirograph toy has a simpler mechanism and a simpler form factor. It has two gears made of plastic so that one fits inside the other. The small gear has holes to allow a pen or a pencil to trace the hypotrochoid curve.

1.2 Hypotrochoid Curves

In his book called *A Book of Curves*, mathematician E.H. Lockwood describes a *roulette* as follows:

"If a curve rolls, without slipping, along another, fixed, curve, any point or line which moves with the moving curve describes a roulette" (Lockwood, 1961, p. 139).

Then, Lawrence, another mathematician, describes a *hypotrochoid* curve in his book *A Catalog of Special Curves*. He writes:

"a roulette traced by a point P attached to a circle S rolling about the inside of a fixed circle C" (Lawrence, 1972, p. 165).

In this section, we will try to explain an equation that produces hypotrochoid curves using Daniel Fishman's Youtube video as our main source Fishman (2013). In addition, we will look at the period of a given hypotrochoid curve. All of the illustrations are drawn using GeoGebra International GeoGebra Institute (2015). To understand how hypotrochoid curves are drawn, it is necessary to notice that we have a moving circle with radius r and a fixed circle with larger radius R. To make it simpler, we will assume that the center of the fixed circle is at (0,0) in the Cartesian coordinate system. We will also use the standard angle measurement (i.e., angles are measured from positive x-axis in the counterclockwise direction.)

The general equation of a circle is given by

$$x^2 + y^2 = R^2.$$

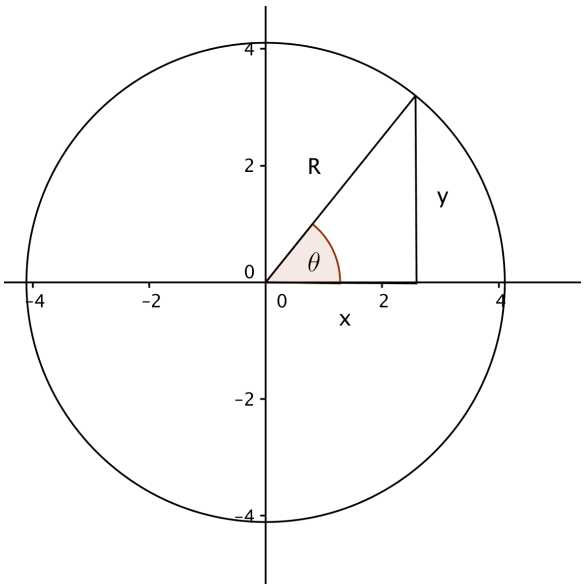


Figure 1.6: Simple circle with radius R.

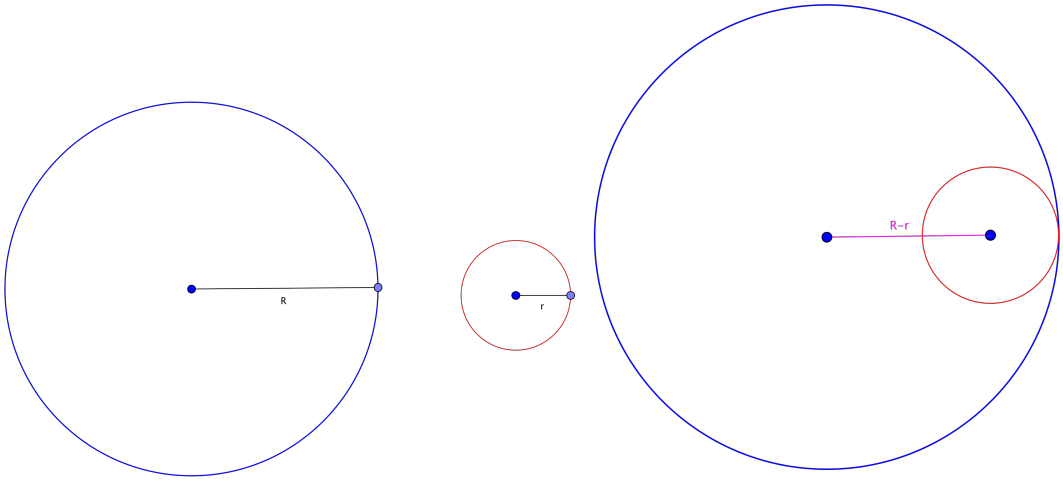
Then, using trigonometry, we can express x and y coordinates in parametric equations. For the given circle, we obtain

$$\sin\theta = \frac{y}{R}, \text{ and}$$
$$\cos\theta = \frac{x}{R}.$$

Then,

$$y = R\sin\theta, \text{ and}$$
$$x = R\cos\theta.$$

Since a hypotrochoid is produced by rotating a small circle inside a big circle, consider two circles as in Figure 1.7 with radius R and r. It becomes clear, then, that the distance from the center of the big circle to the center of the small circle is $R - r$. Again, for simplicity, we will assume that the big circle is centered at $(0, 0)$.



(a) Two circles with radius R and r. **(b)** Small circle inside big circle.

Figure 1.7: Hypotrochoid formation.

Hypotrochoid roulettes form when a point on the small circle is traced as it rotates inside the big circle without slipping. Now, we will consider Figure 1.8, which illustrates the position of the small circle after it has rotated in an anti-clockwise direction.

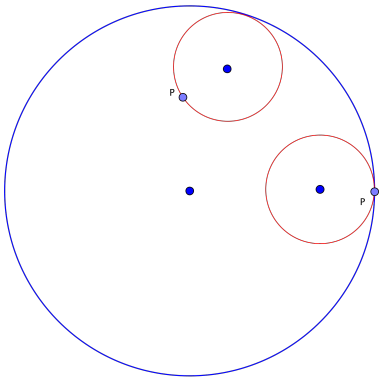


Figure 1.8: Small circle after rotation.

Let θ be the angle measured between the positive x axis (where point P lies on the original position of the small circle) and its position after it has rolled along the big circle. Figure 1.9 illustrates this. Let α be the angle which the small circle has

rotated and which is measured from positive horizontal. We will look at how α relates to θ first, then derive a parametric equation for the hypotrochoid curve.

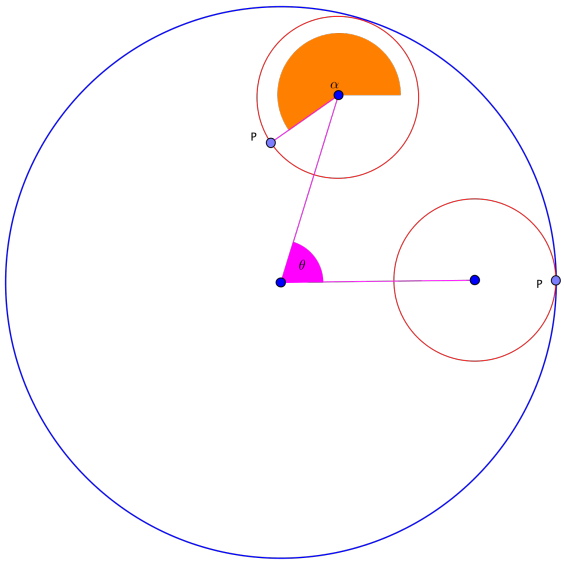


Figure 1.9: Angles θ and α.

We should not forget that the whole circumference of a circle is $2\pi R$, where 2π is the angle corresponding to a full rotation around a point (2π). This will help us understand the relationship between the small circle and the distance it rolled along the large circle. This distance is simply the arc length which corresponds to angle θ, and it is found by $R\theta$. Then, to find the arc length created by an angle, we multiply the angle with the radius. This is a fundamental property of radian measure, and we will use radians to make all of our calculations.

Looking at the small circle, we can see that there should be be some angle β which produces this same arc length as well. Figure 1.10 illustrates this concept.

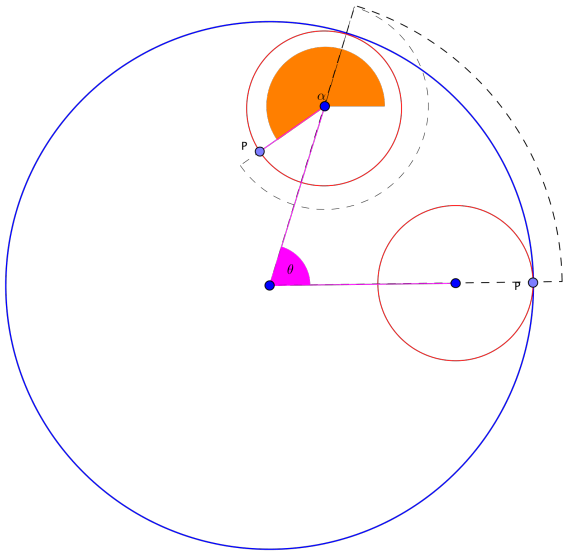


Figure 1.10: Arc length corresponding to θ and β.

Since the arc length corresponding to θ is $R\theta$, $r\beta = R\theta$, thus $\beta = \frac{R\theta}{r}$, then α, β and θ are connected by the equation $\alpha + \beta - \theta = 2\pi$. This is because α is measured from x axis, and the region α and β are overlapping equals to θ. We will call a full rotation around the big circle τ, and since $\beta = \frac{R\theta}{r}$, we can rewrite the equation as

$$\alpha + \tfrac{R\theta}{r} - \theta = \tau.$$

Then, we can solve for α,

$$\begin{aligned}
\alpha &= \tau + \theta - \frac{R\theta}{r} \\
\alpha &= \tau + \left(1 - \frac{R}{r}\right)\theta \\
\alpha &= \tau - \left(\frac{R}{r} - 1\right)\theta.
\end{aligned}$$

Subtracting τ from α will make no difference since τ is a complete turn around the circle. Then,

$$\alpha = -\left(\tfrac{R}{r} - 1\right)\theta.$$

If point P traces our curve, we need to find P's parametric equations. We will use a vector addition to find the parametric equations of P. First, from Figure 1.11 we can see that the center of the small circle has the following parametric equations:

$$x = (R - r)\cos\theta, \text{ and}$$
$$y = (R - r)\sin\theta.$$

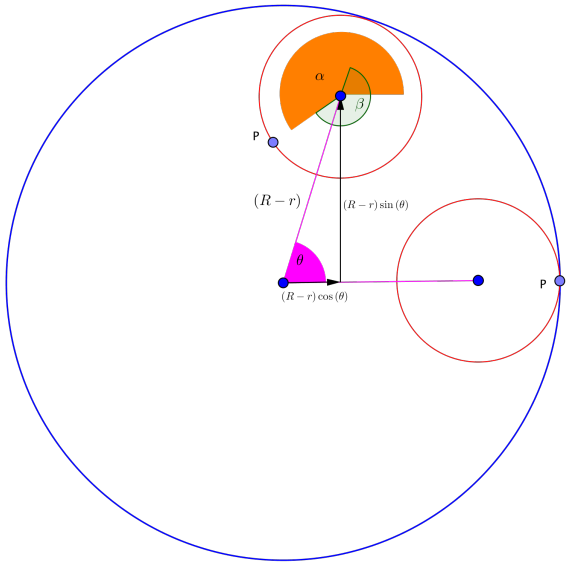
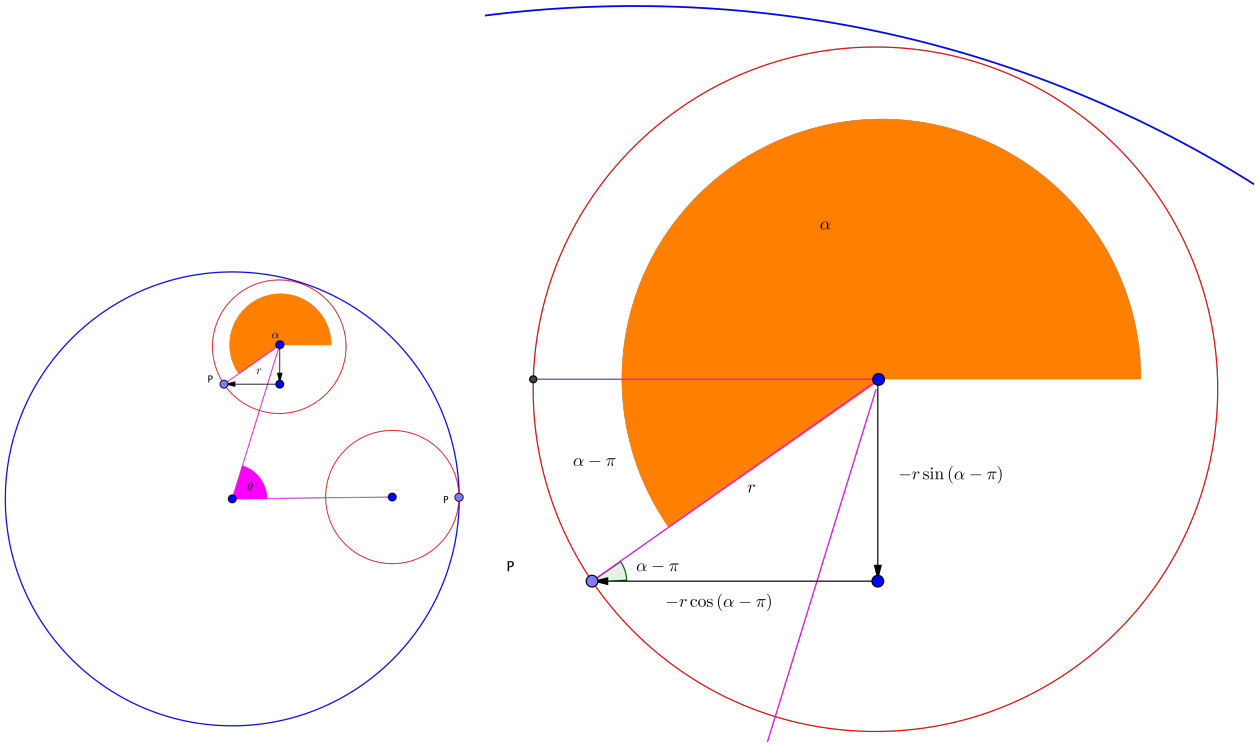


Figure 1.11: The center of the small circle is $R - r$ away from the origin. This information enables us to perform the vector addition.

Now, we will perform a vector addition to find P. Notice that the components of r going to P are in the $-x$ and $-y$ direction.

Figure 1.12a illustrates this, and Figure 1.12b is a more detailed version of 1.12a.



(a) figure a **(b)** figure b

Figure 1.12: Second vector addition from the center of the small circle to P.

Using trigonometry, we notice that the following parametric equations are equivalent:

$$-r\cos(\alpha-\pi) = r\cos(\alpha), \text{ and}$$
$$-r\sin(\alpha-\pi) = r\sin(\alpha).$$

Then, combining the vector additions, we get:

$$x = (R-r)\cos(\theta) + r\cos(\alpha), \text{ and}$$
$$y = (R-r)\sin(\theta) + r\sin(\alpha).$$

As we have found earlier that $\alpha = -\left(\frac{R}{r} - 1\right)\theta$, then the equation becomes:

$$x = (R - r)\cos(\theta) + r\cos\left(-\left(\frac{R}{r} - 1\right)\theta\right), \text{ and}$$
$$y = (R - r)\sin(\theta) + r\sin\left(-\left(\frac{R}{r} - 1\right)\theta\right).$$

Since *cosine* is an even function and *sine* is an odd function, we can further simplify this to:

$$x = (R - r)\cos(\theta) + r\cos\left(\left(\frac{R}{r} - 1\right)\theta\right), \text{ and}$$
$$y = (R - r)\sin(\theta) - r\sin\left(\left(\frac{R}{r} - 1\right)\theta\right).$$

As a reminder, $R-r$ is the distance to the center of the small circle, as illustrated in figure Figure 1.11. Moreover, we assumed that our point P lies on the circumference of the small circle when we calculated the second vector addition (Figure 1.12). Point P is where we would put our pencil if we were to use a Spirograph toy. Thus, we used radius r as the distance from the center of the small circle to point P, while in fact, point P can lie anywhere from the center of the small circle. We can place our pencil inside of the circle or outside of the circle, and if we calculate the distance from the center to the location where we chose to place the pencil d, the equation becomes:

$$x = (R - r)\cos\theta + d\cos\left(\left(\frac{R}{r} - 1\right)\theta\right) \tag{1.1}$$

$$y = (R - r)\sin\theta - d\sin\left(\left(\frac{R}{r} - 1\right)\theta\right) \tag{1.2}$$

We can also put the pencil in the center of the small circle, making $d = 0$. However, this just draws another circle with a radius $R - r$ distance away from the center of the big circle. The equation reflects this fact as well.

Since θ is the variable in 1.1 and 1.2, the equations take the final form written below.

$$x(\theta) = (R - r)\cos\theta + d\cos\left(\left(\frac{R}{r} - 1\right)\theta\right) \tag{1.3}$$

$$y(\theta) = (R - r)\sin\theta - d\sin\left(\left(\frac{R}{r} - 1\right)\theta\right) \tag{1.4}$$

θ can take any value, but *sine* and *cosine* are periodic functions. Therefore, after a certain number of iterations, the curve will repeat itself. We will focus on the instances of this happening in the next chapter.

1.3 Periodicity

The period is defined as the first time a curve repeats itself. In other words, there will be angle α produced by the small circle while it rotates inside the large circle, and the period is when the small circle returns to its original position to complete the curve, with the exact same orientation it had initially. One complete revolution of a circle corresponds to an angle 2π, where α is the angle which the small circle rotates, and θ is the angle that corresponds to this rotation on the big circle. These three angles are related to each other up to equivalence of multiples of 2π with the following equation (negative sign is included in the parametric equations of x and y):

$$\alpha = \left(\frac{R}{r} - 1\right)\theta, \text{ or equivalently}$$
$$\alpha = \frac{R - r}{r}\theta.$$

Let m be the number of full rotations of the small circle around itself, and let n be the number of full rotations around the big circle. Then, n will be the number we are interested in as it will tell us when we can stop rotating the small circle inside the big circle. We can then derive the angle corresponding to this number of full rotations.

For this equation to work, m and n need to be whole numbers. Studying the pencil movement will help us understand why this is necessary.

If we draw a simple circle, we would trace one complete rotation which will result in an angle of 2π. When the circle is complete, the pencil should be back to its the starting point. However, if we stop tracing halfway through the circle, the pencil will be on the opposite side of the starting point, or at π angle from its origin. Therefore, our circle will not be complete. Then, if we consider two circles with one rotating inside the other, the small circle needs to go back to where it has started in order to meet with its starting point. For this reason, n needs to be a whole number.

Additionally, we need to consider the orientation of the small circle when it goes back to its starting point. To have a circle that starts and ends at the same point, m needs to be a whole number as well. Indeed, if m is not a whole number and we place our pencil on the circumference of the small circle to trace a full rotation around the big circle, the circle will not end at its starting point.

Thus, m and n are both whole numbers that determine the revolutions. Then, α is $2\pi m$ if the small circle rotates m times. Similarly, $\theta = 2\pi n$. Then, the equation presented above becomes:

$$2\pi m = \frac{R-r}{r}(2\pi n)$$
$$\frac{R-r}{r} = \frac{m}{n}.$$

We can see that the denominator of $\frac{R-r}{r}$ will give us n. Also, after a period, the curve will repeat itself. As it is the case with the Spirograph toy, we may continue to draw once we have a closed curve, but we would be tracing over that same curve. In other words, if we continue to draw after one period has ended, we will be drawing the same curve over and over again (similar to how sine or cosine curve repeats). This will happen if $\frac{m}{n}$ is not reduced and if it is a multiple of some number. Then, $\frac{R-r}{r}$

has to be reduced so that $gcd(m, n) = 1$ and we stop rotating, which renders n as is needs to be.

1.4 Epitrochoid curves

Epitrochoid curves resemble hypotrochoid curves with the exception that the rotating circle is in this case outside the fixed circle. Then, instead of subtracting radii, we add the radii to find the center of the small circle. According to *A Catalog of Special Plane Curves* by J. Dennis Lawrence, the parametric equations presented below are for the Epitrochoid curves (Lawrence, 1972, p. 161).

$$x = (R + r)\cos\theta - d\cos\left(\left(\frac{R}{r} + 1\right)\theta\right), \text{ and}$$
$$y = (R + r)\sin\theta - d\sin\left(\left(\frac{R}{r} + 1\right)\theta\right).$$

However, we will only use hypotrochoid curves for the iOS application. We are only mentioning epitrochoid curves for completeness.

1.5 Where else can we find Hypotrochoids and Epitrochoids?

Although hypotrochoid and epitrochoid curves are mostly associated with the Spirograph toy, there are several other instances where these curves appear.

1.5.1 Epicycles

Up to this day, we have been constantly researching the universe for it to reveal its secrets. In his self-published website *astronomynotes.com*, Nick Storbel relates the history of some of the models developed by ancient philosophers.

According to Pythagoras (c 569-475 BCE) and his paradigm, Earth was at the center of the universe, and planets moved with constant speed in perfect circular orbits Strobel (2001a). Many of its contemporary philosophers agreed with the Pythagorean Paradigm Strobel (2001a). Plato (427-347 BCE, Socrates' Student), Aristotle (384-322 BCE, Plato's student), and Ptolemy (85-165 CE) developed their own views on the universe in accordance to this paradigm Strobel (2001b).

Plato and his students tried to explain the movement of the universe and developed many different theories Strobel (2001b). Eudoxus, one of Plato's students, said that Earth was at the center, and that other planets and stars were located in concentric spheres from our planet Strobel (2001b). Aristotle supported this idea as well Strobel (2001b). This model is known as the Geocentric Model, which puts Earth at the center of the universe Strobel (2001b).

However, something was missing in the Geocentric Model. Indeed, some planets did not seem to follow the perfect circular motion when observed from Earth Strobel (2001b). Therefore, epicycles, which are "small circles attached to larger circles centered on the Earth", were used to improve the Geocentric Model Strobel (2001b). Epicycles are Strobel (2001b). In other words, planets that were on the concentric spheres with Earth as the center also followed a circular motion Strobel (2001b).

Ptolemy developed the Geocentric Model further with the epicycles Strobel (2001b). However, it contradicted Aristotle's theory and the Pythagorean Paradigm because Earth was no longer the center for planetary motion Strobel (2001b). Yet, he still supported their ideas, and this was only for calculation purposes Strobel (2001b). Figure 1.13 illustrates Ptolemy's epicycle model.

The epicycles are also illustrated by Khan Academy in Figure 1.14. What Mars trajectory look like is illustrated in Figure 1.15 Collingridge (nd).

The illustration in Figure 1.16 by Andrew Bell, which was published on Encyclopdia Britannica based on Giovanni Cassini's drawings, shows the positions of Earth, Sun, Mercury and Venus based on the Geocentric Model Bell (1771).

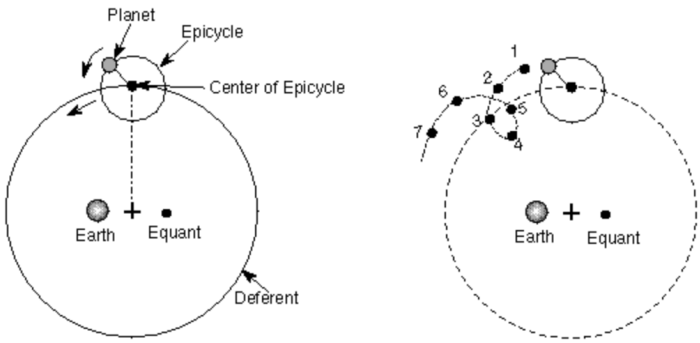
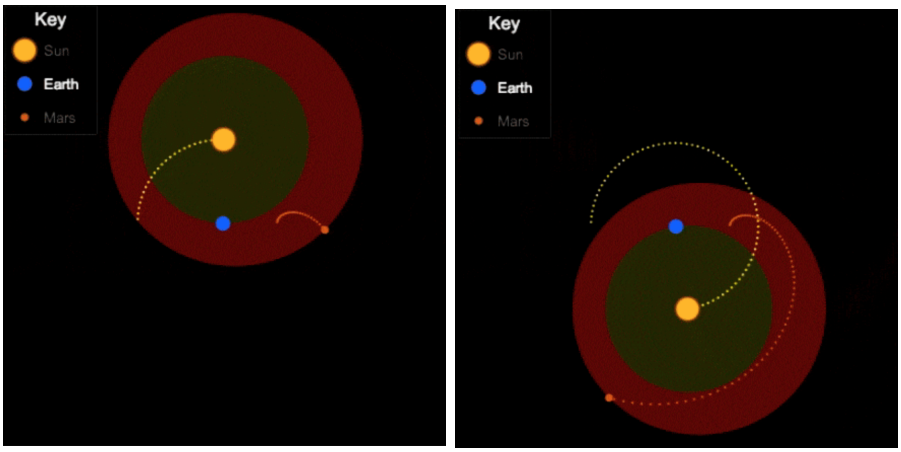


Figure 1.13: An illustration of Ptolemy's Geocentric Model with epicycles Strobel (2001b).



(a) figure a **(b)** figure b

Figure 1.14: Khan Academy Epicycle illustrations Collingridge (nd).

Today, it is commonly accepted that Earth is not the center of the Galaxy, and we use different set of rules to describe planetary motion. Yet, epicycles remain interesting. During a conference held by The Association for Biblical Astronomy, a part of Mantua Country Baptist Church, Dr. Frank Wolff talked about the epicycles of planets Wolff (2007) and linked epicycles to epitrochoids and hypotrochoids. In his presentation entitled *Will the Real Number of Epicycles Please Stand Up?*, he compares the number of epicycles required fir each different model and connects the inconsistency in the models Conference Report (2007). However, today we know that

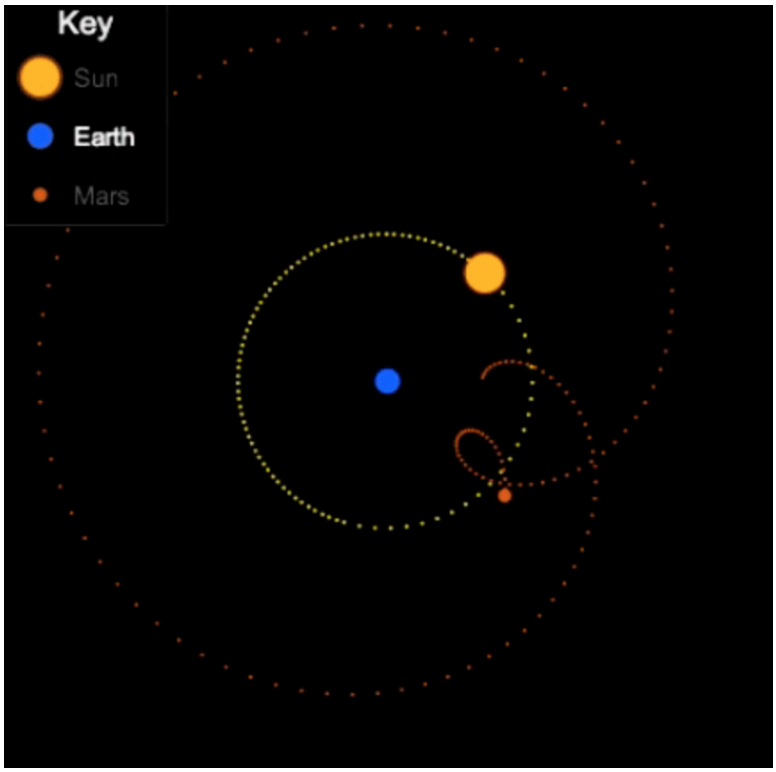


Figure 1.15: Illustration of Mars moving according to Geocentric Model with epicycles Collingridge (nd).

planetary motion is more complicated than what epicycles can explain; yet, Dr. Wolff shows how to derive hypotrochoid, epitrochoid, and some other curves using epicycles.

1.5.2 Relative Satellite Motion

Hypotrochoid curves also appear in the relative satellite motion. In her paper on satellite motion, Soung Sub Lee studied how satellites move with respect to each other Lee (2009). She also stated that parametric equations for cycloids and trochoids can be used to describe the motion in the case of a circular orbit Lee (2009). California State University Curve Bank describes cycloids and trochoids as similar to hypotrochoids, but instead of having a circle rotating inside the other, the circle is rolling along a straight line Gray et al. (2002). Soung Sub Lee wrote that "when dealing with circular orbits of satellites, the relative motion geometry as seen from

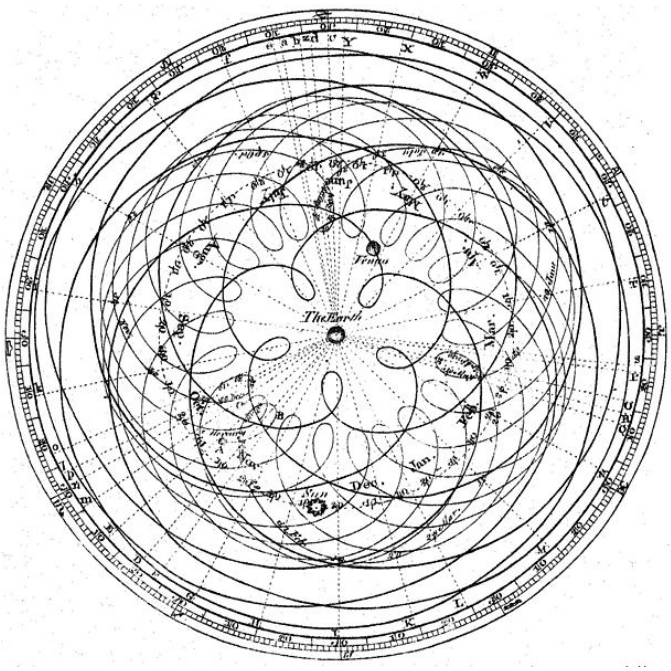


Figure 1.16: Encyclopdia Britannica illustration based on the Geocentric Model Bell (1771).

the polar view is the same as the mathematical curves created by a Spirograph" (Lee, 2009, p. 51).

1.5.3 Tweet Trends

Ye Lin and Romain Vuillemot developed an interesting usage of hypotrochoid curves when they used hypotrochoid curves to describe Tweet trends Lin and Vuillemot (2013). These authors mention the previous examples of hypotrochoid curves used in analyzing "time related data and connectivity among nodes in a network" and say that they will utilize Spirograph curves to explore time sensitive data Lin and Vuillemot (2013). The line drawn represents time, and the holes and the teeth count in the Spirograph tool kit represent other variables. Lin and Vuillemot used these variables, simulated Spirograph curves in a computer program, and created various shapes of hypotrochoid curves Lin and Vuillemot (2013). They also wrote that the petals of the curves make them suitable for the kind of visualization they needed, and

they used each tweet to "decorate" the graph Lin and Vuillemot (2013). Petals can be days in a week, in a month or hours in a day, and by changing the thickness or the color of the line, Lin and Vuillemot were able to summarize Tweet data in a certain period of time Lin and Vuillemot (2013). They called this "decorating" the graphs, making some of the petals appear darker to represent multiple tweet occurrences. Figure 1.17 illustrates products for the CHI 2013 conference.

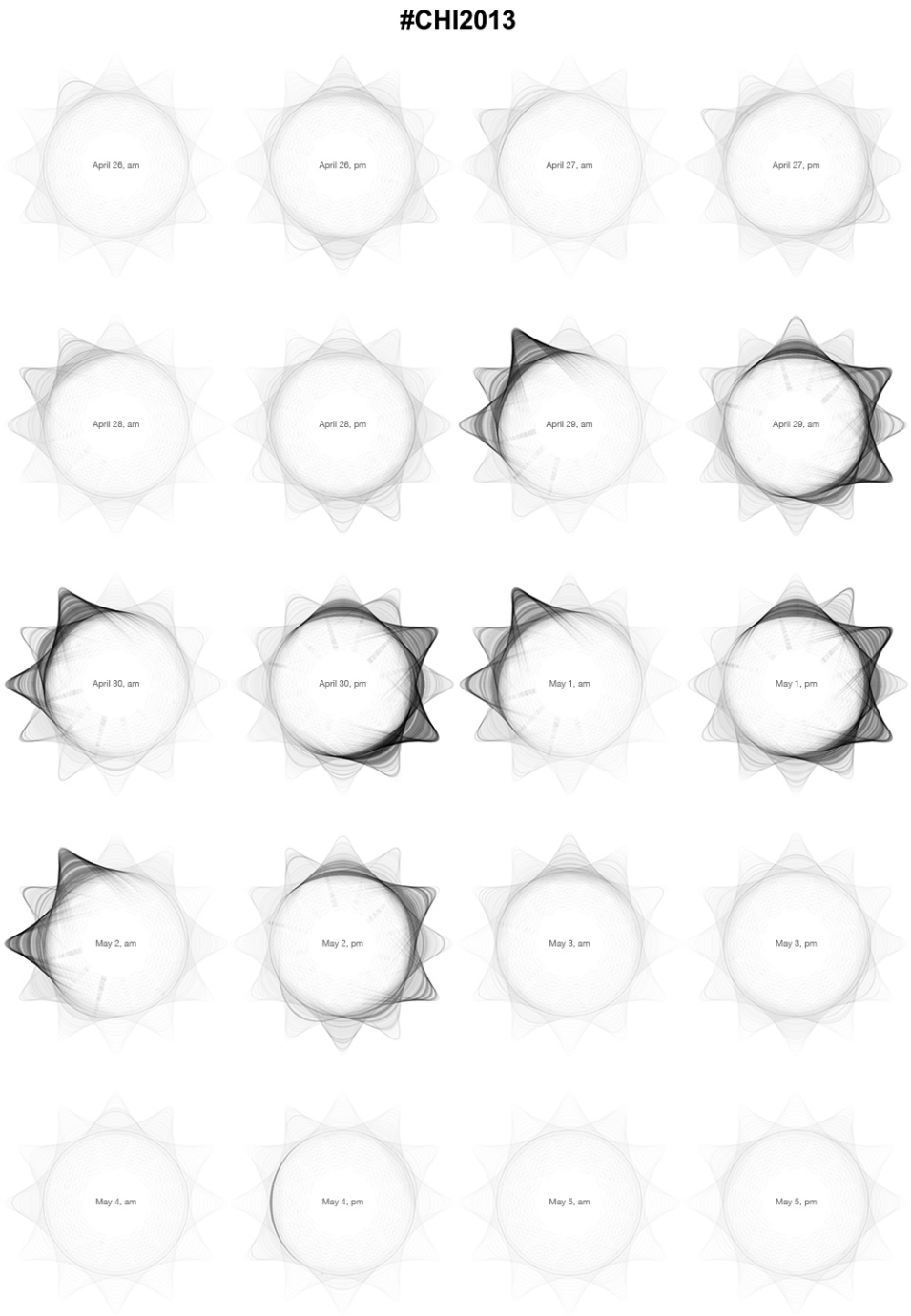
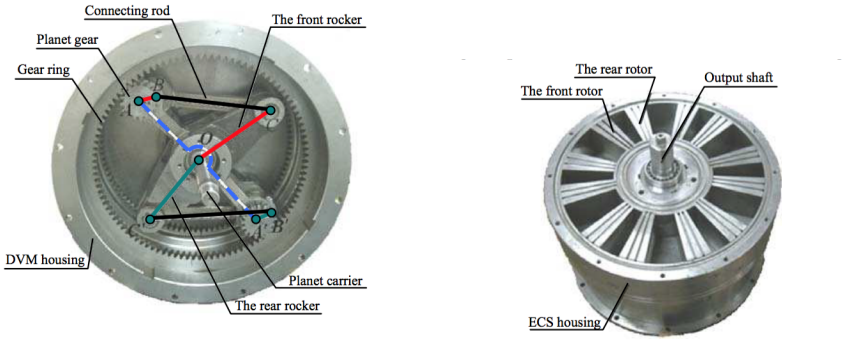


Figure 1.17: Summarizing Tweets during the CHI 2013 conference for a 10 day period Lin and Vuillemot (2013).

1.5.4 Twin Rotor Piston Engine

Xu and his fellow researchers studied Twin Rotor Piston Engines. Reciprocating piston engines and rotary engines are two types of engines that we can find in cars today Xu et al. (2014). Although piston engines advanced, there are several drawbacks which shifted the focus to rotary engines.

The Wankel Engine is a type of rotary engine that is said to have a high potential Xu et al. (2014). However, even the Wankel Engine has some drawback features that led to the development of other rotary engines, such as the Twin-Rotor Piston Engine (TRPE for short) Xu et al. (2014). Moreover, the TRPE has two mechanical systems: the Energy Conversion System, or ECS, which "converts the heat energy into mechanical energy", and the Differential Velocity Mechanism, or DVM, which "converts two non-uniform rotations into a uniform one" Xu et al. (2014). Figure 1.18 shows the pictures of the ECS and the DVM that the authors used in their research.



(a) DVM configuration and pars **(b)** ECS configuration and parts

Figure 1.18: Prototype illustrations of the TRPE Xu et al. (2014).

In order to improve the DVM in the TRPE, the Hypocycloid Twin-Rotor Piston Engine (HTRPE) was developed Xu et al. (2014).

The hypocycloid curve is a special type of hypotrochoid. Indeed, the tracing point is on the circumference of the circle Hall (1992). In Section 1.2, we initially assumed that this was the case for point P. Then, we said that P lies within the small circle.

This is because if we are using a Spirograph gear, we cannot place our pen at the edge of a gear and draw. If we do so, the pencil would slip.

The new HTRPE system is analyzed by separating the DVM "into two non-uniform mechanisms and a hypocycloid mechanism" and by examining the hypocycloid curve Xu et al. (2014). This is possible because rotor engine dynamics are very similar to the Spirograph curve. However, we now have two cylinders instead of having two circles. To increase the power density, the authors suggest to "repeat the oscillatory motion of the driven parts so that they can achieve multiple engine cycles in one revolution of the output shaft"Xu et al. (2014). To do so, they would use hypocycloid curves to analyze when this would happen Xu et al. (2014).

The authors define $i = \frac{r_1}{r_2} = \frac{N}{M}$ where N and M relatively prime integers Xu et al. (2014). Then, two of the requirements on the DVM set to make the HTRPE possible are $M = 1$ and $M = N - 1$ Xu et al. (2014). Based on these two conditions, Xu and his peers further formulated the conditions on the lengths of the parts of the cylinder Xu et al. (2014). Then, the authors make the three arrangements that satisfy all the conditions in Fig 1.19. For simplicity, we are omitting the specifics of the designs and the names of the parameters that control the DVM's structure.

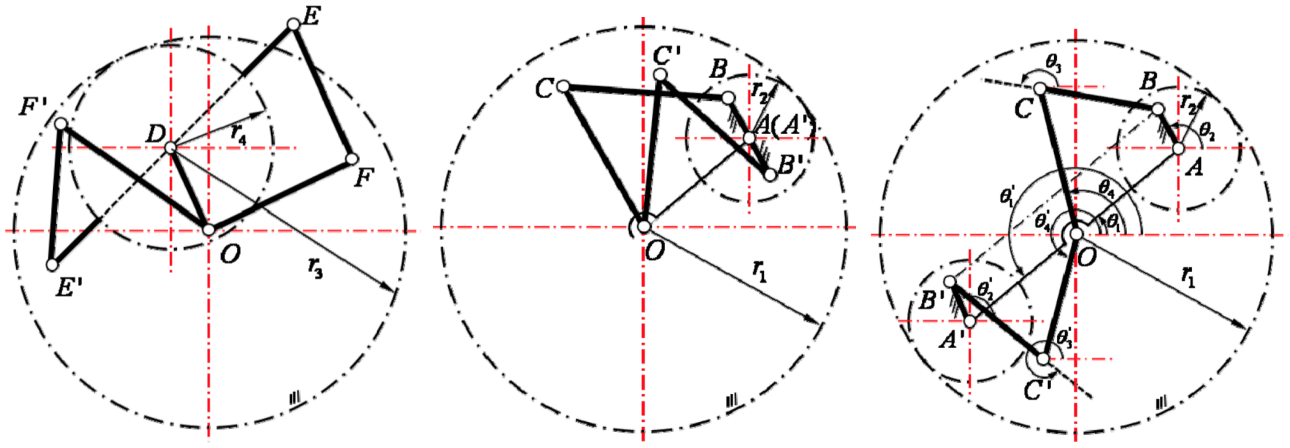


Figure 1.19: Three options for DVM Xu et al. (2014).

So far, this is one of the most direct applications of hypocycloid curves, and it enables engineers to create an engine that has more advantages than the previous models.

1.5.5 Spirograph iOS Application

The Spirograph toy is probably the most popular direct application of the hypotrochoid and epitrochoid curves. Since we have a formula to draw these curves, we will write an iOS Application that functions as a Spirograph toy electronically. The next chapter will focus on the iOS Application making process.

Chapter 2

The iOS Application

The main portion of this project involves the creation of an iOS Spirograph Application that mimics the Spirograph toy digitally. This was a challenging task since we do not have any experience with writing mobile applications. Indeed, Maryville College only teaches C, C++, Java, MySQL, PHP and HTML in its curriculum.

During The 2014 Apple World Wide Developers Conference, WWDC, Apple introduced Swift as their new programming language Apple Inc. (2014). We will be using it for our application. Moreover, many tutorials about Swift are available online since iOS is a popular platform. We will use these tutorials to create our application.

The main reason why we chose iOS as our platform for the application is because it was a personal goal that I set for myself years ago. By using our background in programming, we were able to learn and adapt programming practices for a completely new device. This not only gives us a sense of accomplishment, but it also shows the skills one gains after majoring in Computer Science.

2.1 Xcode

Xcode is an Integrated Development Environment, or IDE, that Apple provides for free to application developers Apple Inc. (2016f). While Xcode enables developers to run, debug, and test their applications either on a virtual device or on a real device, it also has tools such as the *InterfaceBuilder*, the *Test Navigator*, the *Debug Navigator* Apple Inc. (2016f). Developers can then improve and analyze their applications with the help of these tools.

InterfaceBuilder enables users to design the user interface of an application Apple Inc. (2016f). In other words, it is a tool used for placing objects that users interact with an application. These objects include buttons, labels, and text areas among others.

Test Navigator enables developers to run specific tests on their applications Apple Inc. (2016f). Finally, *Debug Navigator* appears when a developer debugs an application. It shows how much CPU, memory, and energy is being consumed by the application. It was interesting to see how the meters changed as we interacted and tested our application.

An experienced Xcode user can probably list more utilities, but we did not need to use all the tools that Xcode offers. Therefore, we only know a few. Xcode also includes warnings about features that will be removed in Swift's upcoming versions. For example, we discovered that the $x++$ increment feature will be removed in Swift 3.

In order to download Xcode, one only needs a Mac computer and an Apple account. This gives users several benefits, such as testing on an actual device and the access to developer forms Apple Inc. (2016d). In order to publish an application on the App Store, one needs to be enrolled in the Apple Developer Program, which costs 99 USD Apple Inc. (2016d). Apple Developer Program gives users more benefits, such as TestFlight Beta Testing Apple Inc. (2016d). TestFlight enables developers

to test their applications with selected users before the application is released on the App Store Apple Inc. (2016c).

We always used the most recent version of Xcode. At the moment, this is 7.3. Every time we installed an update on our iPad, we needed the newest, equivalent version of Xcode. Thus, after updating our iPad to iOS 9.3 from iOS 9.2, we also updated Xcode from 7.2 to 7.3. We are not sure why the version numbers for iOS and Xcode do not match, but a new version of Xcode is available whenever there is a new update for our iOS device. Figure 2.1 shows what users see when Xcode first starts, and Figure 2.2 shows an instance when we were working on our application.

Figure 2.1: Xcode startup window.

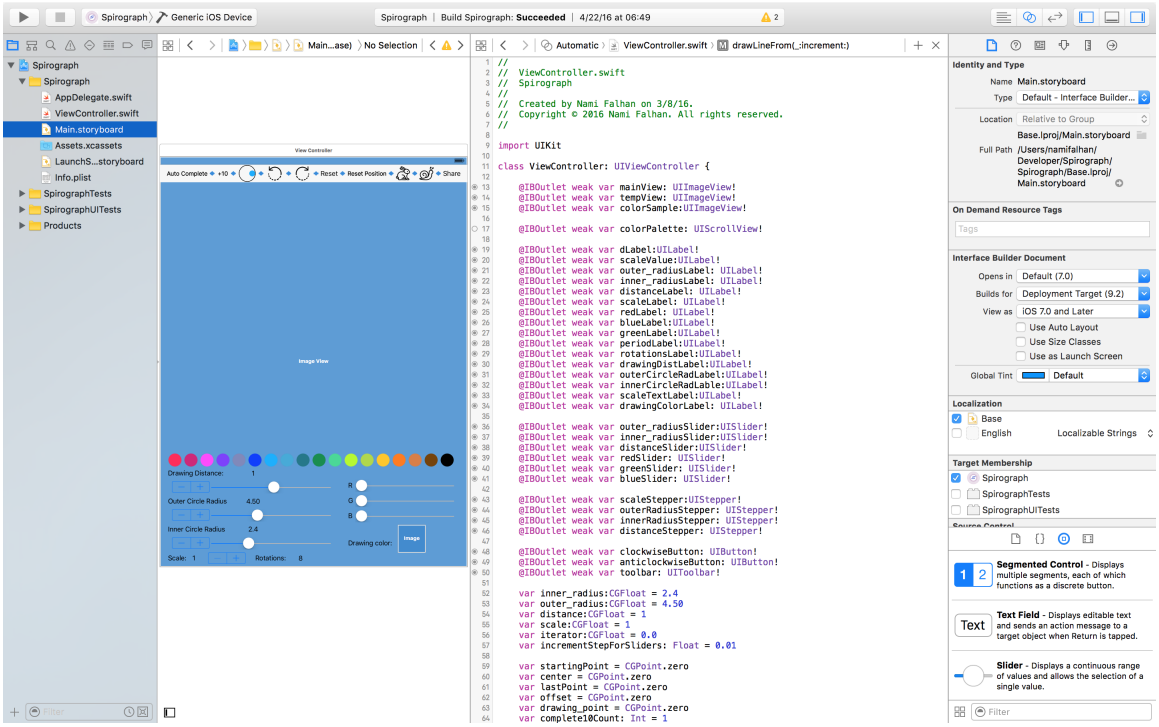


Figure 2.2: *InterfaceBuilder* and our main code side by side.

2.2 Swift

Before Swift, Objective-C was the programming language Apple used for its products. In addition to Objective-C, developers can also use Swift, which Apple improved as a successor to both C and Objective-C Apple Inc. (2016a). For this reason, Objective-C and Swift are compatible Apple Inc. (2016a). The current version of Swift is 2.2, and Apple has been improving it since its release Apple Inc. (2016a).

Swift introduced many new features, but it also feels rather familiar for programmers. Swift uses type-safety to continuously check the variable types in order to prevent accidentally passing variables with the wrong data type as parameters Apple Inc. (2016b). It also features optional types, which are useful when declaring a variable without a value Apple Inc. (2016b).

We were used to using *const* keyword to mark constants, but Swift does not use this. It distinguishes between *let* and *var* keywords when a variable is declared; the former is used for constants, and the later is for actual variables Apple Inc. (2016b).

Swift also uses Automatic Reference Counting and manages memory for the applications (Apple Inc., 2016e, p. 535). It allocates memory when an instance of a class is created and frees memory when that instance is no longer needed (Apple Inc., 2016e, p. 535). If Swift deallocates an instance, then this instance is no longer accessible (Apple Inc., 2016e, p. 535). As a result, the application would crash if it was to try accessing that deallocated instance (Apple Inc., 2016e, p. 535).

Thus, Swift needs to know when it can deallocate an instance (Apple Inc., 2016e, p. 535). Swift knows when to do this by checking what kind of instance reference is being used. Placing *weak, strong* or *unowned* keywords indicate when Swift can deallocate the memory used by an instance (Apple Inc., 2016e, p. 536). More on this subject can be read on *The Swift Programming Language*, a manual provided by Apple (Apple Inc., 2016e, p. 547). We have not used any *strong* referencing since none of the tutorials we have followed used a one of those references. The next section covers the concepts taught in the tutorials we used and what we have learnt from them.

2.3 Tutorials

We followed four main tutorials in order to learn Swift and Objective-C syntax. These tutorials are:

- Developing iOS 7 Apps for iPhone and iPad Steinberg (2013)
- Core Graphics Tutorial Part 1: Getting Started Begbie (2015)
- How To Make A Simple Drawing App with UIKit and Swift Distler and Azeem (2015)
- Bezier Paths and Gesture Recognizers Pop (2015)

The first tutorial was an Objective-C based card game tutorial. It provided a good introduction to the basics because it was designed as a supplemental class material for

Stanford University Steinberg (2013). The next tutorial was to create an application called *Flo*, which counts the number of glasses of water one drinks daily. It was published in Swift by RayWenderlich, a popular website of tutorials Begbie (2015). Then, we followed "How To Make A Simple Drawing Application with UIKit and Swift", which is also by RayWenderlich, to learn how to draw on a screen Distler and Azeem (2015). The last tutorial, "Bezier Paths and Gesture Recognizers", helped us to learn about gestures in Swift.

While we used "How To Make A Simple Drawing Application with UIKit and Swift" and its instructions on how to draw on a screen as the basis of my application, "Bezier Paths and Gesture Recognizers" taught us how to move, rotate, and pinch to zoom objects on a screen Pop (2015). The *Flo* tutorial taught us about *Core Graphics* Begbie (2015). This topic will be a brief portion of the next section.

2.4 iOS Basics

To get familiar with the programming for iOS, we have studied Daniel Steinberg's Stanford University course material. The booklet is designed for iOS 7 and Objective-C even though the current version is iOS 9 Steinberg (2013). When an application is created, we call a *view* what the user is interacting with. A *view controller* is the code that controls the *view* Steinberg (2013). *Views* can have buttons, interactive elements, labels, or widgets Steinberg (2013). Different *views* or scenes, as Steinberg calls them, have different *view controllers*. Information is passed between the *view* and *view controller* Steinberg (2013). To illustrate this, we will take the example of a simple application that has a main *view* as well as a button that takes us into the settings. Since there are two *views*, we will need two *view controllers* for each Steinberg (2013).

We were also introduced to the *Storyboard* and *outlets* concepts. *Storyboard* is also called *Interface Builder* (*IB*), and it is a visual tool that allows one to create the User Interface of the applications Steinberg (2013). One can alternatively design

the interface without using *InterfaceBuilder* and can position elements using coding. However, it is easier to set up the look of the *view* in *Storyboard* by dragging and dropping elements Steinberg (2013).

Outlets are the connections between the objects in our *view* and the *view controller* Steinberg (2013). One can code the *view controller* to interact with a button, label or a similar element in *view*. A button in *view* can also interact with the *view controller* using outlets. For example, when we change the color of the line that draws the hypotrochoid curve in our application, the button sets the color in the *view controller* so that it becomes the new color we chose.

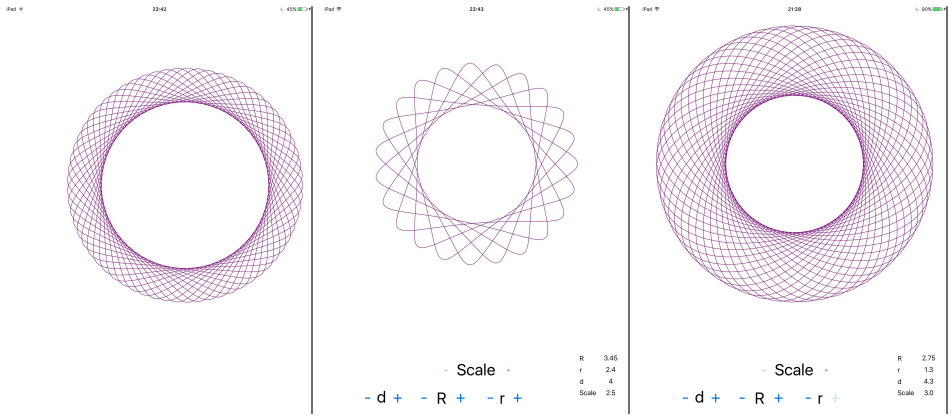
Then, we learnt about *Core Graphics* class, which is what we use to draw on a screen Begbie (2015). Apple describes *quartz* as "the main drawing interface, providing support for path-based drawing" and calls it the"general name for the native drawing technology in iOS"Apple Inc. (2012). In order to use *quartz*, we use functions, data types, and objects that *Core Graphics* and *UIKit* built upon *quartz* Apple Inc. (2012). *Core Graphics* provides context for us draw on. The context is similar to the canvas an artist paints on. Similarly, *UIKit* provides objects, classes, and functions such as *UIImageView*, which is essentially a *view* that displays images Apple Inc. (2016g). We will use the context of the *UIImage* in order to draw on the screen.

The last tutorial, which taught us how to draw on a screen, was extremely helpful since this is essentially what the Spirograph application will do. The result of this drawing tutorial was a simple application which users can draw freely on a blank canvas. Then, by limiting the behavior of the same application, we were able to draw hypotrochoid curves by moving a finger on the screen.

Once we learnt enough on Swift, Xcode, and iOS, we created our own version of the application without using any tutorial files downloaded from the Internet. The next section briefly covers this process, and it also explains how our application developed over time.

2.5 Application's Progress

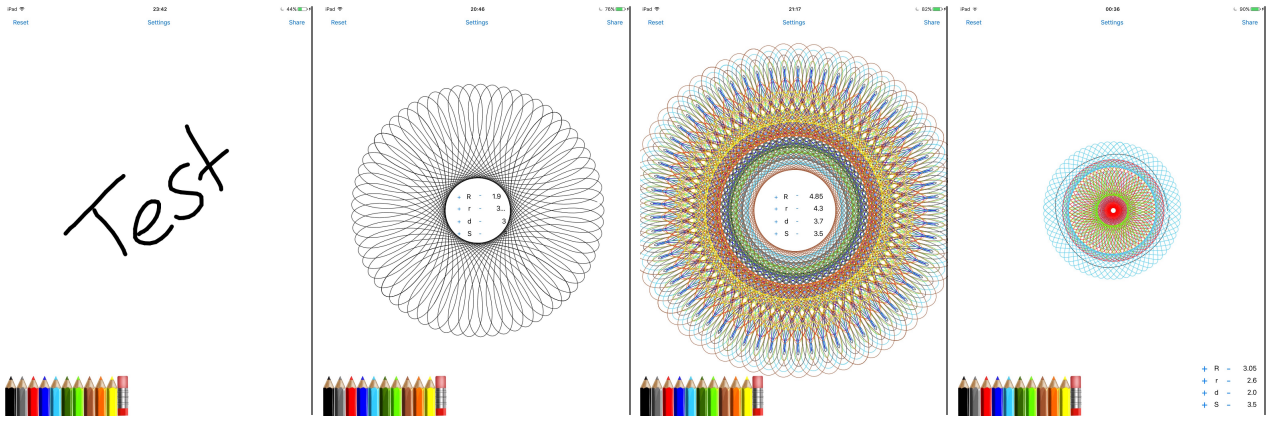
Figure 2.3 (a) shows what our application looked like when we first started coding. Initially, it did not have any elements, and it only drew the hypotrochoid curve based on the parameters given in the code. We then improved this version to allow the user to change the parameters as it is shown in Figures 2.3 (b) and (c). As a result, our application could update the curve based on new parameters and redraw the curve after parameters has changed.



(a) **(b)** **(c)**

Figure 2.3: Developing the application step by step.

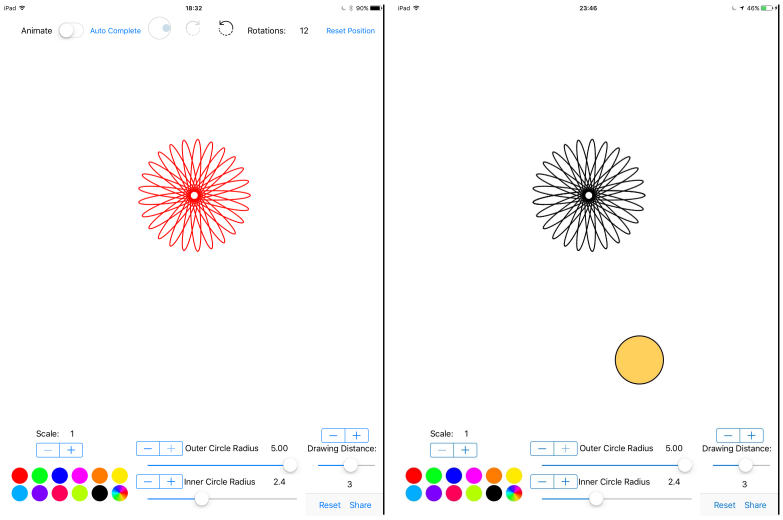
Figure 2.4 (a) shows the simple application we developed for users to draw on a blank canvas using the "How To Make A Simple Drawing App with UIKit and Swift" tutorial. Subsequently, we modified the code so that the users would draw hypotrochoid curves instead of random shapes whenever they touched the screen. Figures 2.4 (b) and (c) illustrates this version of the application. We included simple ways to change the parameters, but soon the positioning started to be a problem. Indeed, we wanted the controls to be on the lower right corner, but they appeared in the center. These positioning issues can be seen in Figures 2.4 (b) and (c). Fortunately, when we disabled *Auto Layout*, we were able to position these controls as we wished. We will dwell more on the *Auto Layout* in the the Challenges section.



(a) (b) (c) (d)

Figure 2.4: Making more improvements to the application.

We also made it possible to change the color of the curves as well as to draw one curve over another. Finally, we fixed the positioning of objects on the screen. Figure 2.4 (d) illustrates what our application looked like after these changes.



(a) (b)

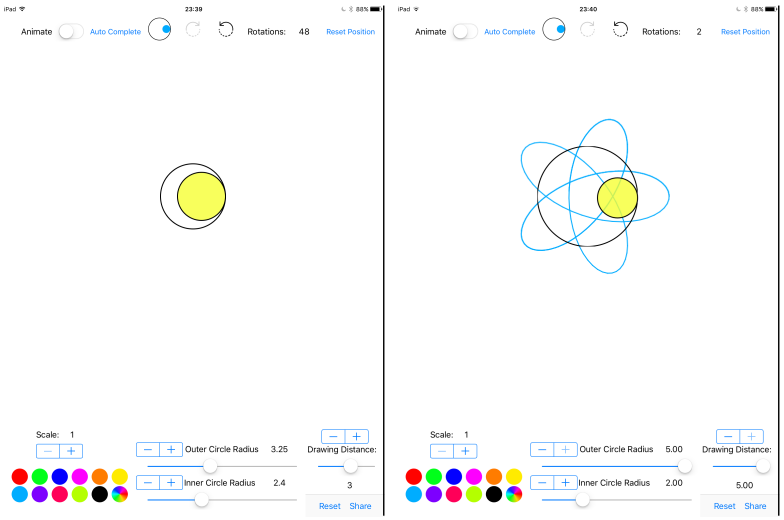
Figure 2.5: Adding more features to the application.

At this stage, we had enough knowledge to start our own application instead of using the modified drawing pad application. Therefore, we created our own version

from what we previously learnt The changes to User Interface can be seen in Figure 2.5 in (a) and (b).

First, we added gesture recognizers because we needed our application to behave like a Spirograph toy. These recognizers enable users to rotate the inner circle around the outer circle. While we rotate the inner circle, it traces the hypotrochoid curve. Initially, we could move this circle anywhere. This version of the application can be seen in Figure 2.5 (b). Later, we adjusted this circle so that it would only move inside the outer circle.

We also added some other functionality, such as automatically completing the curve and changing the rotation's direction of the small circle. All of these changes can be seen in Figure 2.6. The next chapter gives a detailed list of the features of the final application, and Figure 2.7 shows the several changes we made to the User Interface.



(a) **(b)**

Figure 2.6

2.6 The Final Version of the Application

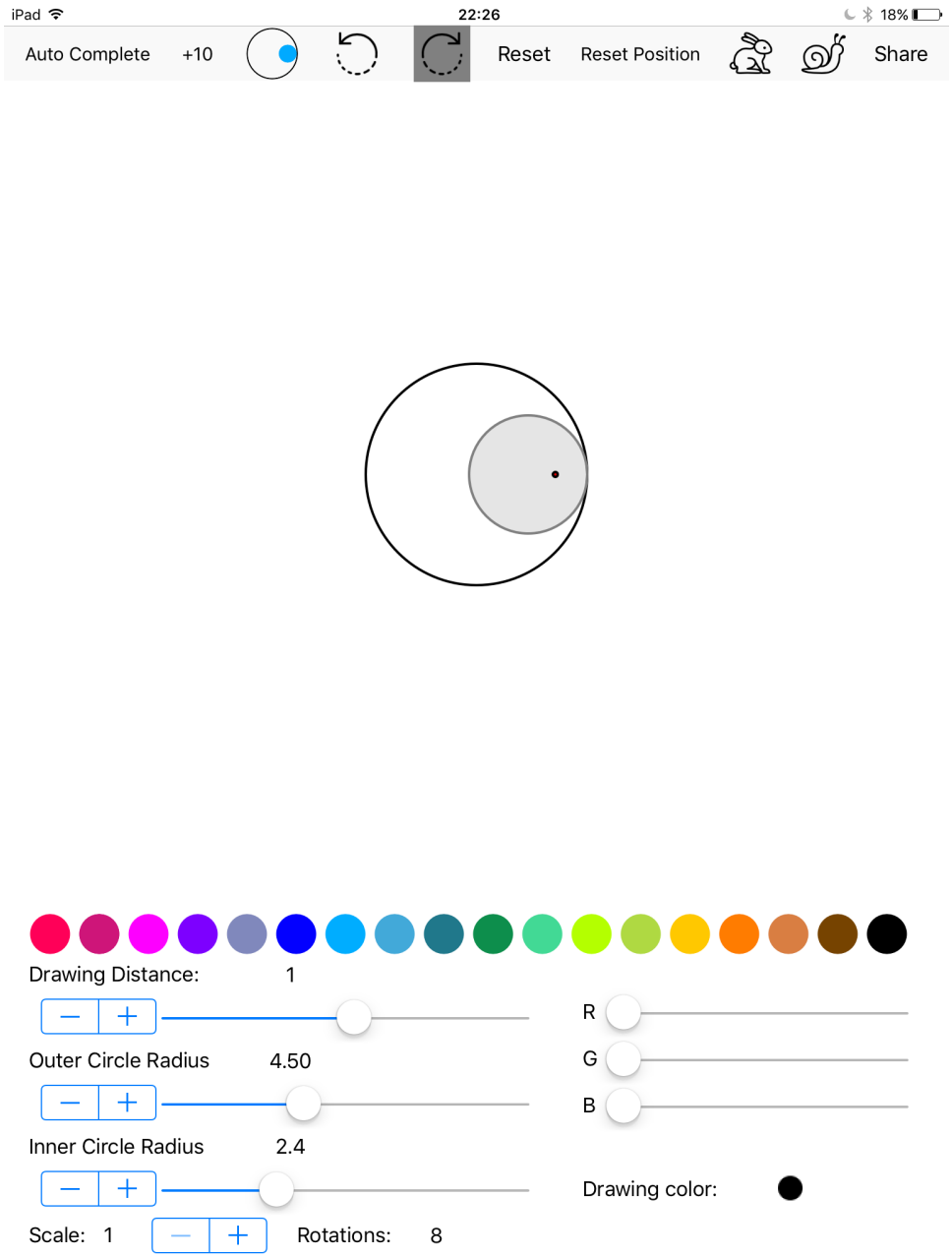


Figure 2.7: The final application.

The final version of our application can be seen in Figure 2.7. By then, we have changed the layout of our application considerably. We added a point on the small

circle to represent the location of the tracing point. If this was a real Spirograph toy, this would be the location where we would put the pencil to trace the curve.

The following features are all present on the final version of the application:

- Ability to complete the curve without moving the inner circle with the Auto Complete button
- Ability to draw the curve as if the inner circle has rotated 10 times inside the outer circle with the +10 button
- Ability to hide the circle so that users can look at their creation without the circles and the pen point
- Ability to draw in an anticlockwise direction
- Ability to erase everything and start over
- Ability to move the inner circle to the starting point without drawing any lines in between the transition
- Ability to change the speed at which the inner circle moves
- Ability to share the image of the hypotrochoid curve produced
- Ability to change the color of the curve being drawn with color buttons or a color mixer
- Ability to alter the parameters, such as the drawing distance from the center of the inner circle (as we mentioned before, this would be the point we place our pencil on the Spirograph gear) and radii of the circles
- Ability to enlarge the circles and the hypotrochoid curves with a scale stepper

2.7 How do we actually draw on the screen?

We used the code shown below in order to draw on the screen. We learnt how to do this in "How To Make A Simple Drawing App with UIKit and Swift" Distler and Azeem (2015). This code basically generates a graphics context on the *UIImageView* on which we want to draw, calculates the coordinates of the next point, and draws a line between the two points. In order to fully grasp these concepts, one needs to refer to Apple's guide on iOS Drawing Concepts Apple Inc. (2012).

```
func drawLineFrom(fromPoint: CGPoint, increment: CGFloat)
{
    UIGraphicsBeginImageContext(view.frame.size)
    let context = UIGraphicsGetCurrentContext()
    tempView.image?.drawInRect(CGRect(x:0, y:0, width:view.frame.size.width, height:view.frame.size.height))

    let T:CGFloat = ((outer_radius-inner_radius) / inner_radius)
    drawing_point.x = ((outer_radius - inner_radius) * cos(iterator)) + (distance * cos(T * iterator))
    drawing_point.y = ((outer_radius - inner_radius) * sin(iterator)) - (distance * sin(T * iterator))

    CGContextSetLineCap(context, CGLineCap.Round)
    CGContextSetLineWidth(context, 2)
    CGContextSetRGBStrokeColor(context, red, green, blue, 1.0)
    CGContextSetBlendMode(context, CGBlendMode.Normal)


    CGContextMoveToPoint(context, fromPoint.x, fromPoint.y)
    lastPoint.x = offset.x + (scale * 20 * drawing_point.x)
    lastPoint.y = offset.y + (scale * 20 * drawing_point.y)
    CGContextAddLineToPoint(context, lastPoint.x, lastPoint.y)


    CGContextStrokePath(context)

    tempView.image = UIGraphicsGetImageFromCurrentImageContext()
    tempView.alpha = opacity
    UIGraphicsEndImageContext()
}
```

2.8 Challenges

One of the challenges we faced is the rapidity at which iOS is evolving. We found numerous outdated tutorials as well as plenty of old forms that do not give good solutions in Swift. A lot has changed since the introduction of the first iOS device,

and Swift has considerably simplified the coding process in almost every aspect since then.

A second challenge we faced was learning the positioning on the screen. We did not follow what Ole Begemann suggests on the *Auto Layout* because it was frustrating to use this feature Begemann (2013). In *Interface Builder*, we position elements by placing constraints on them such as leading space, trailing space, and size. *Auto Layout* handles the rest regardless of the screen size of the iOS device Begemann (2013). This enables developers to write applications that can run on various devices. Because we are new to this method, and because Begemanns suggestions were meant for an older version of Xcode, we decided to not use *Auto Layout*. By disabling this feature, we selected iPad Air 2 as the only device our application could run on since we positioned elements strictly for this device. However, if we decide to publish our application, we will need to use *Auto Layout* to guarantee maximum compatibility with all iOS devices.

We encountered a third challenge when we tried to implement the Animation functionality in our application. This feature would have drawn the curve without any other input from the user and would have made it look as if it was animated. We tried to use a loop which called the function that draw lines (*drawLineFrom* , written above) until the curve is complete. Unfortunately, this created memory problems. Indeed, when this feature was activated, *Debug Navigator* showed a steady increase in the memory consumption, and the application crashed. The *drawLineFrom* function uses *Core Graphics* to create a graphics context, and a line is drawn between the points that are on this context. However, creating this graphics context and stroking the lines requires a lot of memory.

The difference between this feature and Auto Complete feature is that Auto Complete uses a loop to calculate all the points in order to add them to the context. The line is then drawn outside of the loop. In other words, the Animation feature contained all the necessary steps within the loop: starting the context, calculating the points, adding the points to the context, drawing the line between points, and

ending the context. Yet, Auto Complete starts one context and then draws the line between all the points at the end. Initiating and ending context object in a loop is very inefficient. Therefore, we left out this feature because we did not know how to best implement it.

Some of the other challenges involved getting familiar with Xcode and the programming conventions, such as the optional semi-colon, connecting objects to our main code using outlets, and new data types and classes. It was, and it still is, hard to understand the *Quartz, Core Graphics* and *UIKit* concepts. However, practice and making mistakes helped us to get better.

Another challenge we encountered was related to the sizing of the inner and outer circles. The circles are a subclass of *UIView*, which means that they are essentially rectangular areas in which we can draw by manipulating their context Apple Inc. (2016h). Initially, we did not calculate the exact size we needed for these *UIViews*. As a result, while the inner circle moved around the outer circle, it covered the buttons located at the top of the application. *UIViews* were then bigger than they needed to be. Since the background was transparent, we did not realize that this was happening, but we were able to fix this sizing problem soon after. Figure 2.8 shows the circles, and the background has been changed to black to illustrate the concept of drawing on *UIview*.

Even though we had challenges and we chose to simplify our task as much as we could (by not using *Auto Layout*, for instance), our application could be improved further.

2.9 Further Improvements

Although our application was complete for the purpose of this project, we could have made several improvements. The first involves using *UIKit* and Bezier Path to draw paths rather than *Core Graphics* functions. This may improve the quality of the overall image as well as the overall performance.

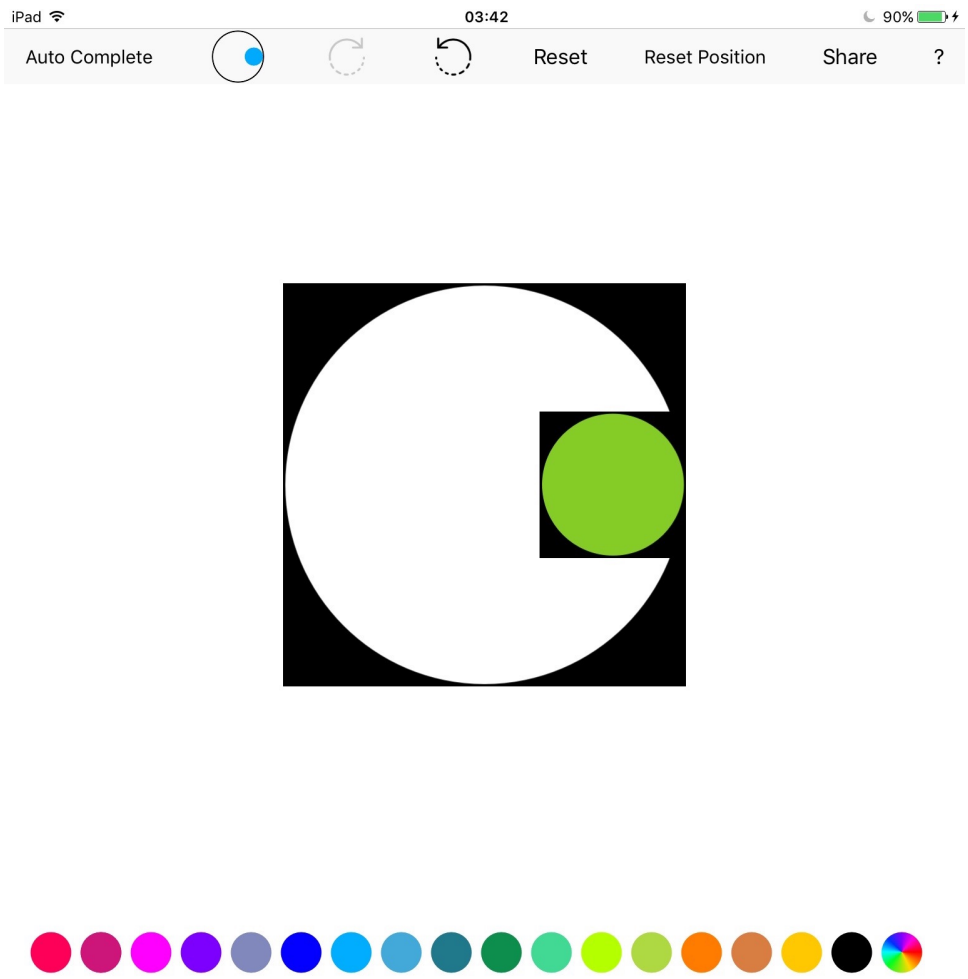


Figure 2.8: The background of the inner and the outer circles has been changed to black to show the borders of *UIView*.

The final version of our application includes the option to draw in a clockwise or an anticlockwise direction. The direction of the rotation and the tracing are controlled with two buttons. We could improve this by removing the buttons and by changing the direction of the rotation based on the dragging gesture's direction.

Moreover, some modern day Spirograph toys include a mechanism that produces the epitrochoid curves. The Spirograph toy has evolved since its initial stages, and our application is very simple compared to some modern Spirograph toys. A considerable improvement would have been to add this feature to our application.

We would have also liked to add the animation feature. It was in our original plans, but in order to accomplish this, we needed a deeper understanding and more practice with Swift.

Since the scope of this thesis does not involve publishing our application, *Auto Layout* was not our priority. However, if we decided to publish our application, we would need to use the *Auto Layout* feature in order to make our application more universal for all iOS devices. The *Auto Layout* feature is definitely something we need to invest more time in because it is a feature that any iOS application with a complicated user interface would use.

Finally, it would be ideal if we could alter and move the the hypotrochoid curves we create. In order to do this, we would need to implement the hypotrochoid curves as a class, and every creation of a hypotrochoid curve would become an object itself. Then, we could manipulate, delete, enlarge, move, and color the curves individually. Right now this is not possible since we are just drawing on context of the *UIImageViews*. Once something is added to the context, it is permanently part of that *view*. This feature would have made our application more interactive.

Even though there is still room for improvements, overall, the implementation of this iOS application was fun, educational, and most importantly, successful. Now, we will look at a more physical occurrence of hypotrochoid curves.

Chapter 3

The Laser Spirograph

Beyond its mathematical uses, we can also find the hypotrochoid curves in light shows that are used by dance clubs and discos. The lasers of these shows are capable of "drawing" hypotrochoid curves. The last portion of this project used instructions on *Instructables.com* in order to build a laser Spirograph machine. We will describe how we built the machine in the next section, and how the image of hypotrochoid curves is produced in the last section of this chapter.

3.1 Building the Laser

We followed user *randofo*'s instructions on *Instructables.com* to build our laser project. We needed 3 motors, 3 round mirrors, 3 rheostats, a laser source (laser pointer works well), a DPDT switch, 2 AA and 2 AAA batteries, and some other nontrivial items, such as small paint storage cups, glue gun, zip ties, project enclosure, among others Randy (2011).

A DPDT switch is a Double Pole-Double Throw switch littelfuse (2015). Pole number determines the number of individual circuits that the switch controls littelfuse (2015). In this case, there are two circuits, one of which contains the laser and its battery, while the other one consists of the motors, the rheostats and their battery. Throw number determines the positions of the toggle and essentially which circuits

are closed littelfuse (2015). This allows the switch to turn one circuit on while turning the other off, and vice versa. For this project, we will turn on both switches at the same time.

A rheostat is a variable resistor, and it can change the current flowing by altering the resistance ResistorGuide.com (2016). We will use rheostats to alter the speed at which the motors rotate. This will produce different designs that will resemble hypotrochoid curves.

We attached the mirrors on the motors using a glue gun. We did not need to be precise, but we centered the mirrors as well as possible. We used paint storage cups as a mount so that all the components were at the same level, and we attached the motors on the paint cups using zip ties.

According to *randofo*'s instructions, we had to position the motors so that the mirrors reflect the laser, which consequently follows a zig-zag path Randy (2011). *Randofo*'s instructions does not give specific instructions on how to position the motors, but he suggests that the two motors are side by side while the third one is on the opposite side Randy (2011). We will need to make some adjustments after we assemble everything so that the laser hits all three mirrors, but there is no need to be precise on the first setup. Using the instructions *randofo* gave us, we created the circuit diagram in Figure 3.1 using Digikey's *Scheme-It* tool Digi-Key EDA and Design Tools (2016).

We then assembled all the pieces together. Figure 3.2 shows the assembled project without its cover.

3.2 What is happening?

Specular reflection occurs when light hits a smooth surface, such as a mirror (Young et al., 2012, p. 1083). We talk about reflection in terms of an imaginary line *Normal*, which is perpendicular to the surface, and angles of incidence and reflection the light rays make with the *Normal* (Young et al., 2012, p. 1083). According to the law

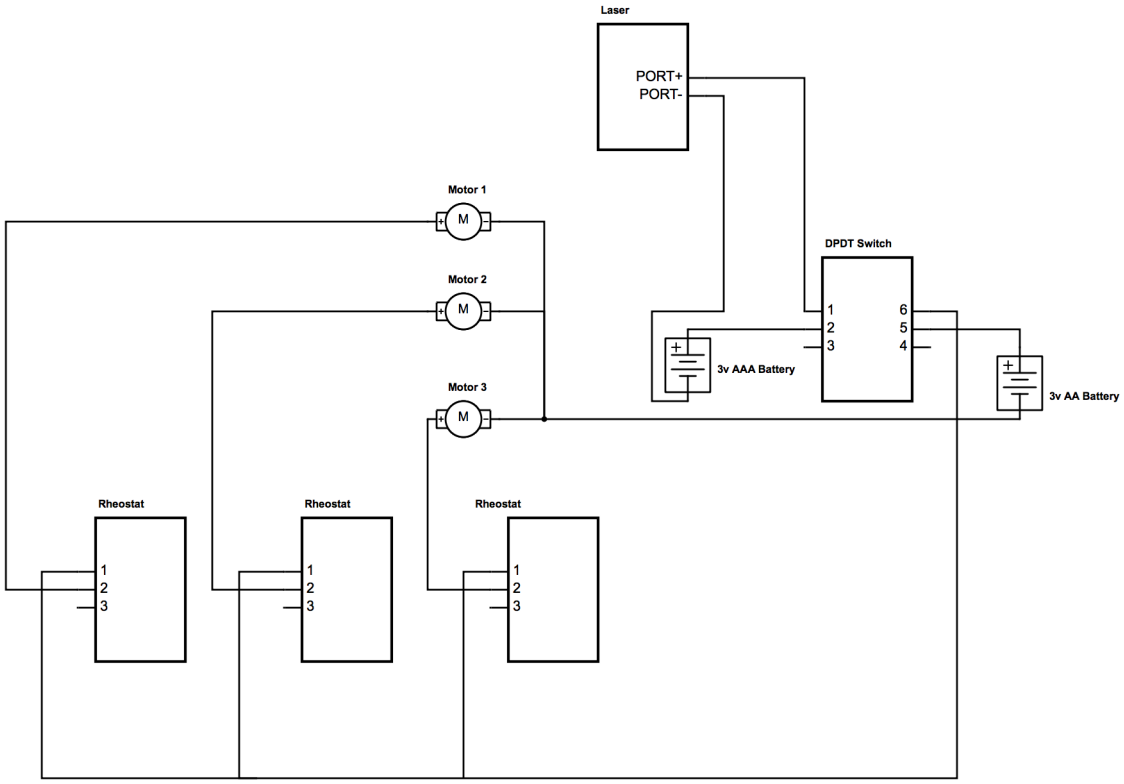
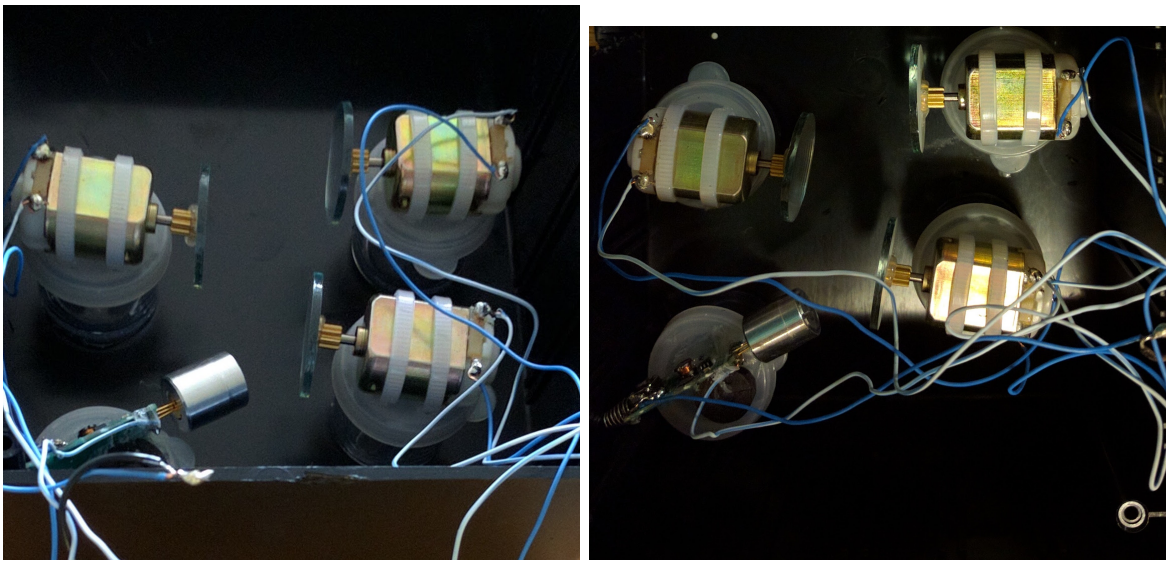


Figure 3.1: Circuit diagram for the Laser Project Digi-Key EDA and Design Tools (2016).

of reflection, the angle of incidence, Θ_i, will be equal to the angle of reflection, Θ_r. (Young et al., 2012, p. 1084). When the motors are turned off, the laser light reflects from all three mirrors. We need the concept of *object* and *image* to understand what is happening.

As discussed in University Physics, anything that light radiates from is an *object*, and an *image* is what appears to be the source of the reflected rays (Young et al., 2012, p. 1115). Thus, when the laser reflects from the first mirror, its image becomes the object for the second mirror and so on. When one of the motors is turned on, the reflected laser light makes a circular or elliptical pattern on the second mirror Chiaverine (2008). This image becomes the object for the second mirror, and if the second motor is also turned on, "the patterns superimpose" to produce more complex shapes Chiaverine (2008). Viewers can sometimes see an elliptical shape rotating if



(a) Motors and laser positioned by *randofo* **(b)** A closer look at *randofo*'s assembled project

Figure 3.2: Assembled Laser Spirograph Randy (2011)

the speeds of the motors are adjusted. We can create different shapes by altering the speed at which the motors or the mirrors rotate. We did not have the time to look at different designs that we could have produced with different motor speeds, but this would be a potential future work.

Chapter 4

Conclusions

This senior project aimed to explore hypotrochoid curves in three different fields: Computer Science, Mathematics, and Physics. Therefore, it combined both of my undergraduate majors with the field of Physics, which will hopefully be my area of specialization in the future.

4.1 Project Summary

We started this thesis project by understanding what a Spirograph is, and we explored the mathematical terminology around Spirograph toys. We derived the following parametric formulas that draw hypotrochoid curves using vector addition, geometric properties, and trigonometric identities.

$$x = (R + r)\cos\theta - d\cos\left(\left(\frac{R}{r} + 1\right)\theta\right), \text{ and}$$
$$y = (R + r)\sin\theta - d\sin\left(\left(\frac{R}{r} + 1\right)\theta\right).$$

Two kinds of revolutions happen in the mechanism of the Spirograph toy. One is the rotation of the small circle around itself whilst the other is the rotation of the

small circle inside the big circle. In order to find the period, we developed on the necessity of the two rotations to be whole numbers. Moreover, we assessed that the hypotrochoid curve would be completed if and only if the pencil returns to its starting location. We concluded that this would happen only when the small circle rotates at least the denominator of the reduced fraction $\frac{R-r}{r}$.

Finally, we looked at some examples where hypotrochoid curves have been used as a tool. Hypotrochoid curves have practical uses when we are analyzing anything that resembles the Spirograph toy, such as some types of motor engines and satellite motions. We also discovered that hypotrochoid curves can be used to analyze time related data.

Then, we created an iOS application from online tutorials and previous knowledge. We used Xcode as the IDE and Swift as the programming language. Since this was something new that we had to teach ourselves, we gradually added features to the application instead of designing the whole application and implementing it. Although we followed several tutorials, the most useful one was "How To Make A Simple Drawing Application with UIKit and Swift", which taught us how to draw on an empty "canvas" on a screen Distler and Azeem (2015). Using this tutorial, we were able to improve upon drawing on a screen using Core Graphics, and we were able to draw the hypotrochoid curves that the Spirograph produces by moving our finger on the screen. The final application now allows users to move an inner circle around an outer circle. They are then mimicking the Spirograph toy since they are drawing an hypotrochoid curve by rotating this inner circle inside the outer circle.

Next, we designed a project that creates illusions of the hypotrochoid curves using laser. Indeed, we set up a laser light which follows a zig zag pattern. We then reflected this light in three different mirrors which are attached to motors. We were able to create these curves because we placed our laser at a certain angle while rotating the first mirror. This forms an elliptical pattern on the second mirror which creates an hypotrochoid curve since the successive mirrors are also rotating.

4.2 Reflections

I knew already as a freshman that I would have to complete a senior thesis. Maryville College is a small liberal college where faculty, staff, and students communicate and build relationships throughout the years. These relationships and some discussions with my advisers helped me find the idea of this thesis project.

As a Computer Science and Mathematics double major, I was inspired by both fields. It was also one my ambitions to learn how to program for the iOS platform. Thus, I wanted to bring both of my majors together while fulfilling my ambition. One of the classes that influenced me the most and that made me realize what interested me was Physics. Therefore, I wanted to incorporate Physics in my thesis as well. When my advisor gave me this idea of an iOS Application that would mimic the Spirograph toy, I was thus very tempted to follow it because it was combining all of my interests. This is why I decided to write an iOS Application about the Spirograph toy. Learning how to write applications for iOS was also one of my goals before I started college.

During this process, I also explored and analyzed the hypotrochoid curves that the Spirograph toy produces. As a small side project, I found a tutorial on how to create hypotrochoid curves using lasers, mirrors, and motors. I was then able to connect Physics, Mathematics, and Computer Science all at the same time.

To implement the application, I particularly used the skills I learnt from my Computer Science classes. Notwithstanding that there are many different platforms and that coding for different systems varies significantly, the basics of computer programming allowed me to understand and do a lot of various things. Adapting to program for different systems is possible, and this is one of the skills we gain from a Computer Science major. In addition to learning specific programming language, ideas, and algorithms, I also learnt how to teach myself new concepts and how to build upon what I already know. I learnt how to think like a programmer in order to create, analyze, and identify bugs in my programs during my education in Maryville

College. I was able to take what I learnt and applied it to a completely different platform. This alone was the most rewarding part of my thesis.

Even though I do not consider Computer Science as a potential future career, it was my only major when I started my undergraduate education. Now that I have completed this project, I have realized that there are countless possibilities for which I can program. This is a considerable accomplishment for me. Being able to program is indeed an important tool for anybody since scientific advancements today depend a lot on computer calculations and simulations.

The Spirograph application is a perfect example of this statement, despite the fact that it was only a small scale project. I was able to verify the parametric equation we derived by using it in my application. I am now grateful for my decision to double major because I was able combine my two areas of specialization.

Bibliography

Bibliography

Amazon.com/Kahootz (2015). Spirograph deluxe design set. http://goo.gl/fd1O32. Accessed: 2015-11-11. 1

Apple Inc. (2012). ios drawing concepts. https://developer.apple.com/library/ios/documentation/2DDrawing/Conceptual/DrawingPrintingiOS/GraphicsDrawingOverview/GraphicsDrawingOverview.html. 30, 36

Apple Inc. (2014). Introducing swift. https://developer.apple.com/videos/play/wwdc2014/402/. 24

Apple Inc. (2016a). About swift. https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/. 27

Apple Inc. (2016b). About swift. 27

Apple Inc. (2016c). Apple developer program. https://developer.apple.com/programs/. 26

Apple Inc. (2016d). Choosing a membership. https://developer.apple.com/support/compare-memberships/. 25

Apple Inc. (2016e). The swift programming language. 28

Apple Inc. (2016f). Tools youll love to use. https://developer.apple.com/xcode/ide/. 25

Apple Inc. (2016g). Uikit framework reference. https://goo.gl/1jIhMa. 30

Apple Inc. (2016h). Uiview class reference. https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIView_Class/. 38

Begbie, C. (2015). Core graphics tutorial part 1: Getting started. https://www.raywenderlich.com/90690/modern-core-graphics-with-swift-part-1. 28, 29, 30

Begemann, O. (2013). 10 things you need to know about cocoa auto layout. http://oleb.net/blog/2013/03/things-you-need-to-know-about-cocoa-autolayout/. 37

Bell, A. (1771). Astronomy. https://www.khanacademy.org/partner-content/nasa/measuringuniverse/spacemath1/a/planets-epicycles. 15, 18

Chiaverine, C. (2008). Mini laser spirograph. http://www.discoveriescience.com/Mini_Laser_Spirograph-Puebla_Conference.pdf. 43

Collingridge, P. (n.d.). Planets and epicycles. https://www.khanacademy.org/partner-content/nasa/measuringuniverse/spacemath1/a/planets-epicycles. 15, 16, 17

Conference Report (2007). Conference report. http://www.geocentricity.com/ba1/no121/confrpt.html. 16

Coope, T. (2015). Spirograph. https://toytales.ca/spirograph/. Accessed: 2015-23-10. 3, 4

Digi-Key EDA and Design Tools (2016). *Scheme It Software (Version 3.1)*. 42, 43

Distler, J.-P. and Azeem, A. (2015). How to make a simple drawing app with uikit and swift. https://www.raywenderlich.com/87899/make-simple-drawing-app-uikit-swift. 28, 29, 36, 46

Fishman, D. (2013). Spirograph mathematics. https://goo.gl/TAFrAq. Accessed: 2015-23-10. 4

Gray, S. B., Venit, S., and Abbott, R. (2002). Epicycloids etc. - animated. http://goo.gl/Bmst9K. 17

Hall, L. M. (1992). Trochoids, roses and thorns - beyond the spirograph. *The College Mathematics Journal*, 23(1). 21

International GeoGebra Institute (2015). *Sage Mathematics Software (Version 5.0.164.0)*. 4

Kaveney, W. (2004). The marvelous wondergraph. http://goo.gl/op0w0S. Accessed: 2015-23-10. 3

Lawrence, J. D. (1972). *A Catalog of Special Plane Curves*. Dover Publications, Inc. 4, 14

Lee, S. S. (2009). Dynamics and control of satellite relative motion: Designs and applications. Master's thesis, Virginia Polytechnic Institute and State University. 17, 18

Lin, Y. and Vuillemot, R. (2013). Spirograph designs for ambient display of tweets. 18, 19, 20

littelfuse (2015). Spst, spdt, dpst, and dpdt explained. http://goo.gl/vbymxr. 41, 42

Lockwood, E. H. (1961). *A Book of Curves*. Cambridge University Press. 4

Paget, S. (2014). The art of math - you could do that with spirograph. https://goo.gl/Xg5wWA. Accessed: 2015-23-10. 2, 3

Pop, S. (2015). Bezier paths and gesture recognizers. https://www.raywenderlich.com/87899/make-simple-drawing-app-uikit-swift. 28, 29

Randy (2011). Laser spirograph. http://www.instructables.com/id/Laser-Spirograph/?ALLSTEPS. 41, 42, 44

ResistorGuide.com (2016). Rheostat. http://www.resistorguide.com/rheostat/. 42

Steinberg, D. H. (2013). *Developing iOS7 Apps*. Stanford University. 28, 29, 30

Strobel, N. (2001a). Philosophical backdrop. http://www.astronomynotes.com/history/s2.htm. 15

Strobel, N. (2001b). Plato's homework problem. http://www.astronomynotes.com/history/s2.htm. 15, 16

The Sage Developers (2015). *Sage Mathematics Software (Version 6.7)*. 2

Tuck, F. E. (1913). *The Boy Mechanic Book I*. Popular Mechanics Press Chicago. 2, 3

Wolff, F. (2007). Will the real number of epicycles stand up. In *Will the Real Number of Epicycles Stand Up*, 3. Geocentricity, International Conference on Absolutes. 16

Xu, X., Xu, H., Deng, H., Gu, F., Gu, T., and Chris, J. (2014). An investigation of a hypocycloid mechanism based twin-rotor piston engine. *Journal of Mechanical Engineering Science*, Part C. 21, 22

Young, H. D., Freedman, R. A., and Ford, A. L. (2012). *Sear's and Zemansky's University Physics with Modern Physics*. Pearson Education, Inc., 13 edition. 42, 43

Appendix

Appendix A

The iOS Application Swift Code

```
//
//  ViewController.swift
//  Spirograph
//
//  Created by Nami Falhan on 3/8/16.
//  Copyright    2016 Nami Falhan. All rights reserved.
//

import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var mainView: UIImageView!
    @IBOutlet weak var tempView: UIImageView!
    @IBOutlet weak var colorSample:UIImageView!

    @IBOutlet weak var colorPalette: UIScrollView!

    @IBOutlet weak var dLabel:UILabel!
    @IBOutlet weak var scaleValue:UILabel!
    @IBOutlet weak var outer_radiusLabel: UILabel!
    @IBOutlet weak var inner_radiusLabel: UILabel!
    @IBOutlet weak var distanceLabel: UILabel!
    @IBOutlet weak var scaleLabel: UILabel!
    @IBOutlet weak var redLabel: UILabel!
    @IBOutlet weak var blueLabel:UILabel!
    @IBOutlet weak var greenLabel:UILabel!
    @IBOutlet weak var periodLabel:UILabel!
    @IBOutlet weak var rotationsLabel:UILabel!
    @IBOutlet weak var drawingDistLabel:UILabel!
    @IBOutlet weak var outerCircleRadLabel:UILabel!
    @IBOutlet weak var innerCircleRadLable:UILabel!
    @IBOutlet weak var scaleTextLabel:UILabel!
    @IBOutlet weak var drawingColorLabel: UILabel!

    @IBOutlet weak var outer_radiusSlider:UISlider!
    @IBOutlet weak var inner_radiusSlider:UISlider!
    @IBOutlet weak var distanceSlider:UISlider!
```

```
@IBOutlet weak var redSlider: UISlider!
@IBOutlet weak var greenSlider: UISlider!
@IBOutlet weak var blueSlider: UISlider!

@IBOutlet weak var scaleStepper:UIStepper!
@IBOutlet weak var outerRadiusStepper: UIStepper!
@IBOutlet weak var innerRadiusStepper: UIStepper!
@IBOutlet weak var distanceStepper: UIStepper!

@IBOutlet weak var clockwiseButton: UIButton!
@IBOutlet weak var anticlockwiseButton: UIButton!
@IBOutlet weak var toolbar: UIToolbar!

var inner_radius:CGFloat = 2.4
var outer_radius:CGFloat = 4.50
var distance:CGFloat = 1
var scale:CGFloat = 1
var iterator:CGFloat = 0.0
var incrementStepForSliders: Float = 0.01

var startingPoint = CGPoint.zero
var center = CGPoint.zero
var lastPoint = CGPoint.zero
var offset = CGPoint.zero
var drawing_point = CGPoint.zero
var complete10Count: Int = 1

var swiped = false
var opacity: CGFloat = 1.0
var red:CGFloat = 0.0
var green: CGFloat = 0.0
var blue: CGFloat = 0.0

var firstTime:Bool = true
var innerCircle = ShapeView(origin: CGPoint(x: 100 , y:100), radius: 2.4,
    fill: UIColor(hue: 0, saturation: 0, brightness: 0, alpha: 0.1),
    outsideColor: UIColor.grayColor())

var outerCircle = ShapeView(origin: CGPoint(x: 384, y: 512), radius: 4.50,
    fill: UIColor.clearColor(), outsideColor: UIColor.blackColor())

var pencilPoint = ShapeView(origin: CGPoint(x:150, y: 150), radius: 0.1,
    fill: UIColor.redColor(), outsideColor: UIColor.blackColor())

var circlesHidden: Bool = false
var iteratorIncrementValue: CGFloat = 0.1
var clockwiseSet: Bool = true

let colors: [(CGFloat, CGFloat, CGFloat)] =
[(0,0,0), (0.99,0.07,0.36), (0.8,0.12,0.47), (0.98,0.16, 0.99),(0.69, 0.15, 0.98),
(0.5, 0.54, 0.73), (0.05, 0.14, 0.98),(0.12, 0.69, 0.99), (0.28, 0.67, 0.84),
(0.15, 0.47, 0.54),(0.1, 0.55, 0.31), (0.29, 0.85, 0.59), (0.71, 0.99, 0.2),
(0.69, 0.84, 0.31), (0.99, 0.49, 0.14), (0.84, 0.49, 0.29),(1, 0.78, 0.18),
(0.46, 0.26, 0.05)]

override func viewDidLoad() {
    super.viewDidLoad()

    self.view.addSubview(innerCircle)
```

```
        mainView.addSubview(outerCircle)
        mainView.addSubview(pencilPoint)
        firstTimeCalculations()
        circleReposition(CGPoint(x: center.x + (scale * 20 * (outer_radius - inner_radius)),
                y: center.y), circle: innerCircle)

        circleReposition(center, circle: outerCircle)
        circleReposition(startingPoint, circle: pencilPoint)

        let panGR = UIPanGestureRecognizer(target: self, action: #selector(ViewController.didPan(_:)))
        innerCircle.addGestureRecognizer(panGR)
        drawSample()
        clockwiseButton.backgroundColor = UIColor.grayColor()

    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
    }

    func firstTimeCalculations(){
        center = CGPoint(x: self.view.bounds.size.width/2,
                y: self.view.bounds.size.width/2)

        offset = CGPoint(x: center.x + (outer_radius - inner_radius), y: center.y)
        let T:CGFloat = ((outer_radius-inner_radius) / inner_radius)

        drawing_point.x = ((outer_radius - inner_radius)*cos(iterator)) + (distance * cos(T * iterator))
        drawing_point.y = ((outer_radius - inner_radius)*sin(iterator)) - (distance*sin(T * iterator))

        startingPoint.x = offset.x + (scale * 20 * drawing_point.x)
        startingPoint.y = offset.y + (scale * 20 * drawing_point.y)

        lastPoint = startingPoint
        firstTime = false
    }

    func circleReposition(moveCircleToThisPoint: CGPoint, circle :ShapeView){
        circle.reposition(moveCircleToThisPoint)
    }

    func didPan(panGR: UIPanGestureRecognizer)
    {
        if(CGRectContainsPoint(innerCircle.frame, panGR.locationInView(tempView)))
        {
                if(firstTime)
                {
                        firstTimeCalculations()
                }

                innerCircle.superview!.bringSubviewToFront(innerCircle)

                drawLineFrom(lastPoint, increment: iterator)

                circleReposition(CGPoint(
                        x:center.x + ((outer_radius - inner_radius) * cos(iterator) * 20 * scale),
                        y:center.y + ((outer_radius - inner_radius) * sin(iterator) * 20 * scale)),
                        circle: innerCircle)
```

```
            circleReposition(lastPoint, circle: pencilPoint)
            iterator+=iteratorIncrementValue
            merge()
    }
}

func merge(){
    UIGraphicsBeginImageContext(mainView.frame.size)
    mainView.image?.drawInRect(CGRect(x:0, y:0, width:view.frame.size.width,
            height:view.frame.size.height), blendMode: CGBlendMode.Normal, alpha: 1.0)

    tempView.image?.drawInRect(CGRect(x:0, y:0, width:view.frame.size.width,
            height:view.frame.size.height), blendMode: CGBlendMode.Normal, alpha: opacity)

    mainView.image = UIGraphicsGetImageFromCurrentImageContext()
    UIGraphicsEndImageContext()
    tempView.image = nil
}

func drawLineFrom(fromPoint: CGPoint, increment: CGFloat){
    UIGraphicsBeginImageContext(view.frame.size)
    let context = UIGraphicsGetCurrentContext()
    tempView.image?.drawInRect(CGRect(x:0, y:0, width:view.frame.size.width,
            height:view.frame.size.height))

    let T:CGFloat = ((outer_radius-inner_radius) / inner_radius)
    drawing_point.x = ((outer_radius - inner_radius) * cos(iterator)) +
            (distance * cos(T * iterator))

    drawing_point.y = ((outer_radius - inner_radius) * sin(iterator)) -
            (distance * sin(T * iterator))

    CGContextSetLineCap(context, CGLineCap.Round)
    CGContextSetLineWidth(context, 2)
    CGContextSetRGBStrokeColor(context, red, green, blue, 1.0)
    CGContextSetBlendMode(context, CGBlendMode.Normal)

    CGContextMoveToPoint(context, fromPoint.x, fromPoint.y)
    lastPoint.x = offset.x + (scale * 20 * drawing_point.x)
    lastPoint.y = offset.y + (scale * 20 * drawing_point.y)
    CGContextAddLineToPoint(context, lastPoint.x, lastPoint.y)

    CGContextStrokePath(context)

    tempView.image = UIGraphicsGetImageFromCurrentImageContext()
    tempView.alpha = opacity
    UIGraphicsEndImageContext()
}

func drawLineContinuously(fromPoint: CGPoint,  periodValue: Int){
    UIGraphicsBeginImageContext(view.frame.size)
    let context = UIGraphicsGetCurrentContext()
    tempView.image?.drawInRect(CGRect(x:0, y:0, width:view.frame.size.width,
            height:view.frame.size.height))

    CGContextSetLineCap(context, CGLineCap.Round)
    CGContextSetLineWidth(context, 2)
    CGContextSetRGBStrokeColor(context, red, green, blue, 1.0)
    CGContextSetBlendMode(context, CGBlendMode.Normal)
```

```
    let T:CGFloat = ((outer_radius-inner_radius) / inner_radius)
    CGContextMoveToPoint(context, fromPoint.x, fromPoint.y)
    while(iterator <= (CGFloat(periodValue) * 2 * CGFloat(M_PI)) + 0.2)
    {
        drawing_point.x = ((outer_radius - inner_radius) * cos(iterator)) +
            (distance * cos(T * iterator))

        drawing_point.y = ((outer_radius - inner_radius) * sin(iterator)) -
            (distance * sin(T * iterator))

        lastPoint.x = offset.x + (scale * 20 * drawing_point.x)
        lastPoint.y = offset.y + (scale * 20 * drawing_point.y)
        CGContextAddLineToPoint(context, lastPoint.x, lastPoint.y)
        iterator+=iteratorIncrementValue
    }

    CGContextStrokePath(context)

    tempView.image = UIGraphicsGetImageFromCurrentImageContext()
    tempView.alpha = opacity
    UIGraphicsEndImageContext()
    startFromBeginning()
}

func drawLine10(fromPoint: CGPoint,  periodValue: Int){
    UIGraphicsBeginImageContext(view.frame.size)
    let context = UIGraphicsGetCurrentContext()
    tempView.image?.drawInRect(CGRect(x:0, y:0, width:view.frame.size.width,
                    height:view.frame.size.height))

    CGContextSetLineCap(context, CGLineCap.Round)
    CGContextSetLineWidth(context, 2)
    CGContextSetRGBStrokeColor(context, red, green, blue, 1.0)
    CGContextSetBlendMode(context, CGBlendMode.Normal)

    let T:CGFloat = ((outer_radius-inner_radius) / inner_radius)
    CGContextMoveToPoint(context, fromPoint.x, fromPoint.y)


    if(periodValue <= 10){
        while(iterator <= (CGFloat(periodValue) * 2 * CGFloat(M_PI)) + 0.2)
        {
            drawing_point.x = ((outer_radius - inner_radius) * cos(iterator)) +
                    (distance * cos(T * iterator))

            drawing_point.y = ((outer_radius - inner_radius) * sin(iterator)) -
                    (distance * sin(T * iterator))

            lastPoint.x = offset.x + (scale * 20 * drawing_point.x)
            lastPoint.y = offset.y + (scale * 20 * drawing_point.y)
            CGContextAddLineToPoint(context, lastPoint.x, lastPoint.y)

            iterator+=iteratorIncrementValue
        }
    } else {
        let previousRotations: CGFloat = iterator/CGFloat(2 * M_PI)

        let rotationAmount:CGFloat = 10 + previousRotations
```

```
        while(iterator <= (CGFloat(rotationAmount) * 2 * CGFloat(M_PI)) + 0.2)
        {
            drawing_point.x = ((outer_radius - inner_radius) * cos(iterator)) +
                    (distance * cos(T * iterator))

            drawing_point.y = ((outer_radius - inner_radius) * sin(iterator)) -
                    (distance * sin(T * iterator))

            lastPoint.x = offset.x + (scale * 20 * drawing_point.x)
            lastPoint.y = offset.y + (scale * 20 * drawing_point.y)
            CGContextAddLineToPoint(context, lastPoint.x, lastPoint.y)

            iterator+=iteratorIncrementValue
        }
    }

    CGContextStrokePath(context)

    tempView.image = UIGraphicsGetImageFromCurrentImageContext()
    tempView.alpha = opacity
    UIGraphicsEndImageContext()
}

@IBAction func reset(sender:AnyObject){
    mainView.image = nil
    center = CGPoint(x: self.view.bounds.size.width/2, y: self.view.bounds.size.width/2)
    offset = CGPoint(x: center.x + (outer_radius - inner_radius), y: center.y)
    startingPoint = CGPoint.zero
    lastPoint = CGPoint.zero
    iterator = 0.0
    firstTime = true
    circleReposition(CGPoint(x: center.x + (scale * 20 * (outer_radius - inner_radius)),
                    y: center.y), circle: innerCircle)

    firstTimeCalculations()
    circleReposition(startingPoint, circle: pencilPoint)

}

func startFromBeginning(){
    center = CGPoint(x: self.view.bounds.size.width/2, y: self.view.bounds.size.width/2)
    offset = CGPoint(x: center.x + (outer_radius - inner_radius), y: center.y)
    startingPoint = CGPoint.zero
    lastPoint = CGPoint.zero
    iterator = 0.0
    firstTime = true
    circleReposition(CGPoint(x: center.x + (scale * 20 * (outer_radius - inner_radius)),
            y: center.y), circle: innerCircle)

    firstTimeCalculations()
    circleReposition(startingPoint, circle: pencilPoint)

}

@IBAction func resetPosition(sender: UIButton){
    startFromBeginning()
    circleReposition(CGPoint(x: center.x + (scale * 20 * (outer_radius - inner_radius)),
            y: center.y), circle: innerCircle)
```

```
}


@IBAction func clockwiseDirection(sender: UIButton){
    sender.backgroundColor = UIColor.grayColor()
    anticlockwiseButton.backgroundColor = UIColor.clearColor()
    circleReposition(CGPoint(x: center.x + (scale * 20 * (outer_radius - inner_radius)),
            y: center.y), circle: innerCircle)

    iteratorIncrementValue = abs(iteratorIncrementValue)
    clockwiseSet = true
    startFromBeginning()
}

@IBAction func antiClockwiseDirection(sender: UIButton){
    sender.backgroundColor = UIColor.grayColor()
    clockwiseButton.backgroundColor = UIColor.clearColor()
    circleReposition(CGPoint(x: center.x + (scale * 20 * (outer_radius - inner_radius)),
            y: center.y), circle: innerCircle)

    if(iteratorIncrementValue > 0)
    {
        iteratorIncrementValue = -1 * abs(iteratorIncrementValue)
    }
    clockwiseSet = false
    startFromBeginning()
}

@IBAction func changeSpeed(sender: UIButton){
    if(sender.tag==1 && clockwiseSet){
        iteratorIncrementValue = 0.1
    } else if(sender.tag == 1 && !clockwiseSet){
        iteratorIncrementValue = -0.1
    } else if(sender.tag == 2 && clockwiseSet){
        iteratorIncrementValue = 0.05
    } else if(sender.tag == 2 && !clockwiseSet){
        iteratorIncrementValue = -0.05
    }
}

func period(numerator: CGFloat, denominator: CGFloat) -> Int{
    var top: Int = Int(numerator)
    var bottom: Int = Int(denominator)
        while(bottom != 0)
        {
            let temp = bottom
            bottom = top % bottom
            top = temp
        }

        return Int(denominator)/top
}

@IBAction func autoComplete(){
    startingPoint = lastPoint
    if(firstTime){
        firstTimeCalculations()
    }
    if(period(round((outer_radius - inner_radius) * 100), denominator: round(inner_radius*100)) > 80){
```

```
            let alertController = UIAlertController(title: "Too many iterations!", message:
                "In order to use this property, please change some parameters
                to decrease the number of rotations less than 80.",
                preferredStyle: UIAlertControllerStyle.Alert)

            alertController.addAction(UIAlertAction(title: "Dismiss",
                style: UIAlertActionStyle.Default,handler: nil))

            self.presentViewController(alertController, animated: true, completion: nil)
        } else {
            drawLineContinuously(lastPoint,  periodValue: period(round((outer_radius - inner_radius) * 100),
                denominator: round(inner_radius*100)))
        }

        merge()
    }

    @IBAction func complete10(){
        startingPoint = lastPoint

        if(firstTime){
            firstTimeCalculations()
        }
        if(period(round((outer_radius - inner_radius) * 100),
                        denominator: round(inner_radius*100)) < 10) {

            drawLine10(lastPoint,  periodValue: period(round((outer_radius - inner_radius) * 100),
                denominator: round(inner_radius*100)))

            circleReposition(CGPoint(
                x:center.x + ((outer_radius - inner_radius) * cos(iterator) * 20 * scale),
                y:center.y + ((outer_radius - inner_radius) * sin(iterator) * 20 * scale) ),
                circle: innerCircle)

            circleReposition(lastPoint, circle: pencilPoint)

        }
        else {
            drawLine10(lastPoint,  periodValue: 10 * complete10Count)
            circleReposition(CGPoint(
                x:center.x + ((outer_radius - inner_radius) * cos(iterator) * 20 * scale),
                y:center.y + ((outer_radius - inner_radius) * sin(iterator) * 20 * scale) ),
                circle: innerCircle)

            circleReposition(lastPoint, circle: pencilPoint)
            complete10Count+=1
        }
        merge()
    }



    @IBAction func share(sender: AnyObject)
    {
        outerCircle.removeFromSuperview()
        UIGraphicsBeginImageContext(mainView.bounds.size)
        mainView.image?.drawInRect(CGRect(x: 0, y: 0, width: mainView.frame.size.width,
                height: mainView.frame.size.height))
```

```
    let image = UIGraphicsGetImageFromCurrentImageContext()
    UIGraphicsEndImageContext()

    let activity = UIActivityViewController(activityItems: [image], applicationActivities: nil)
    activity.popoverPresentationController?.barButtonItem = sender as? UIBarButtonItem
    self.presentViewController(activity, animated: true, completion: nil)
    mainView.addSubview(outerCircle)
}

func updateMinsMaxs(){
    outer_radiusSlider.minimumValue = Float(inner_radius) + 1
    outerRadiusStepper.minimumValue = Double(inner_radius) + 1
    inner_radiusSlider.maximumValue = Float(outer_radius) - 1
    innerRadiusStepper.maximumValue = Double(outer_radius) - 1
    distanceSlider.maximumValue = Float(inner_radius) - 0.5
    distanceStepper.maximumValue = Double(inner_radius) - 0.5

    if((inner_radius-0.5) < (distance)){
        distance = CGFloat(distanceSlider.maximumValue)
        distanceLabel.text = String(format: "%.2f", distance)
        distanceSlider.value = Float(distance)
        distanceStepper.value = Double(distance)
    }

    outer_radiusLabel.text = String(format: "%.2f", outer_radius)
    inner_radiusLabel.text = String(format: "%.2f", inner_radius)
    distanceLabel.text = String(format: "%.2f", distance)
}

func changeCircleSize(circle: ShapeView, radius: CGFloat){
    circle.circleRadius = radius * scale
    circle.frame.size.width = (radius * 40 * scale) + 4
    circle.frame.size.height = (radius * 40 * scale) + 2
    circle.size = (radius * 40 * scale) + 2
    circle.setNeedsDisplay()
}

@IBAction func changeOuterRadius(slider: UISlider){
        outer_radius = CGFloat(slider.value)
        outer_radius = CGFloat(round(100 * outer_radius) / 100)
        updateMinsMaxs()
        outer_radiusLabel.text = String(format: "%.2f", outer_radius)
        outerRadiusStepper.value = Double(outer_radius)
        startFromBeginning()
        changeCircleSize(outerCircle, radius: outer_radius)
        circleReposition(CGPoint(
            x: center.x + (scale * 20 * (outer_radius - inner_radius)),
            y: center.y), circle: innerCircle)

        circleReposition(center, circle: outerCircle)
        circleReposition(lastPoint, circle: pencilPoint)
        periodLabel.text = String(period(round((outer_radius - inner_radius) * 100),
            denominator: round(inner_radius*100)))
}

@IBAction func changeInnerRadius(slider: UISlider){
        inner_radius = CGFloat(slider.value)
        inner_radius = CGFloat(round(100 * inner_radius) / 100)
        updateMinsMaxs()
```

```
        inner_radiusLabel.text = String(format: "%.2f", inner_radius)
        innerRadiusStepper.value = Double(inner_radius)
        startFromBeginning()
        changeCircleSize(innerCircle, radius: inner_radius)
        circleReposition(CGPoint(x: center.x + (scale * 20 * (outer_radius - inner_radius)),
            y: center.y), circle: innerCircle)

        circleReposition(lastPoint, circle: pencilPoint)
        periodLabel.text = String(period(round((outer_radius - inner_radius) * 100),
            denominator: round(inner_radius*100)))
}

@IBAction func changeDistance(slider: UISlider){
        distance = CGFloat(slider.value)
        distance = CGFloat(round(100 * distance) / 100)
        distanceLabel.text = String(format: "%.2f", distance)
        distanceStepper.value = Double(distance)
        startFromBeginning()
        changeCircleSize(innerCircle, radius: inner_radius)
}

@IBAction func changeDistanceStepper(stepper: UIStepper){
    distance = CGFloat(distanceStepper.value)
    updateMinsMaxs()
    distanceLabel.text = String(format: "%.2f", distance)
    distanceSlider.value = Float(distance)
    startFromBeginning()
}

@IBAction func changeScale(stepper: UIStepper){
    scale = CGFloat(scaleStepper.value)
    scaleLabel.text = String(scale)
    startFromBeginning()
    changeCircleSize(innerCircle, radius: inner_radius)
    changeCircleSize(outerCircle, radius: outer_radius)
    circleReposition(CGPoint(x: center.x + (scale * 20 * (outer_radius - inner_radius)),
            y: center.y), circle: innerCircle)

    circleReposition(center, circle: outerCircle)
}

@IBAction func changeOuterRadiusStepper(stepper: UIStepper){
    outer_radius = CGFloat(outerRadiusStepper.value)
    updateMinsMaxs()
    outer_radiusLabel.text = String(format: "%.2f", outer_radius)
    outer_radiusSlider.value = Float(outer_radius)
    startFromBeginning()
    changeCircleSize(outerCircle, radius: outer_radius)
    circleReposition(CGPoint(x: center.x + (scale * 20 * (outer_radius - inner_radius)),
            y: center.y), circle: innerCircle)

    circleReposition(center, circle: outerCircle)
    circleReposition(lastPoint, circle: pencilPoint)
    periodLabel.text = String(period(round((outer_radius - inner_radius) * 100),
            denominator: round(inner_radius*100)))
}

@IBAction func changeInnerRadiusStepper(stepper: UIStepper){
    inner_radius = CGFloat(innerRadiusStepper.value)
```

```
    inner_radiusLabel.text = String(format: "%.2f", inner_radius)
    inner_radiusSlider.value = Float(inner_radius)
    updateMinsMaxs()
    startFromBeginning()
    changeCircleSize(innerCircle, radius: inner_radius)
    circleReposition(CGPoint(x: center.x + (scale * 20 * (outer_radius - inner_radius)),
            y: center.y), circle: innerCircle)

    circleReposition(lastPoint, circle: pencilPoint)
    periodLabel.text = String(period(round((outer_radius - inner_radius) * 100),
            denominator: round(inner_radius*100)))
}

@IBAction func changeColor(colorButton: UIButton){
    if((colorButton.tag >= 0) && (colorButton.tag <= 17))
    {
        (red, green, blue) = colors[colorButton.tag]
        drawSample()
        pencilPoint.fillColor = UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        pencilPoint.outsideColor = UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        pencilPoint.setNeedsDisplay()
        toolbar.tintColor = UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        dLabel.textColor =  UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        scaleValue.textColor =  UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        outer_radiusLabel.textColor =  UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        inner_radiusLabel.textColor =  UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        distanceLabel.textColor =  UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        scaleLabel.textColor =  UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        redLabel.textColor =  UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        blueLabel.textColor =  UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        greenLabel.textColor =  UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        periodLabel.textColor =  UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        rotationsLabel.textColor =  UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        drawingDistLabel.textColor =  UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        outerCircleRadLabel.textColor =  UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        innerCircleRadLable.textColor =  UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        scaleTextLabel.textColor =  UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        drawingColorLabel.textColor =  UIColor.init(red: red, green: green, blue: blue, alpha: 1)
    }
}

@IBAction func colorSliderChanged(sender: UISlider){
        red = CGFloat(redSlider.value)
        blue=CGFloat(blueSlider.value)
        green=CGFloat(greenSlider.value)
        drawSample()
        pencilPoint.fillColor = UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        pencilPoint.outsideColor = UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        pencilPoint.setNeedsDisplay()
        toolbar.tintColor = UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        dLabel.textColor =  UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        scaleValue.textColor =  UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        outer_radiusLabel.textColor =  UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        inner_radiusLabel.textColor =  UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        distanceLabel.textColor =  UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        scaleLabel.textColor =  UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        redLabel.textColor =  UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        blueLabel.textColor =  UIColor.init(red: red, green: green, blue: blue, alpha: 1)
        greenLabel.textColor =  UIColor.init(red: red, green: green, blue: blue, alpha: 1)
```

```
            periodLabel.textColor = UIColor.init(red: red, green: green, blue: blue, alpha: 1)
                rotationsLabel.textColor = UIColor.init(red: red, green: green, blue: blue, alpha: 1)
                drawingDistLabel.textColor = UIColor.init(red: red, green: green, blue: blue, alpha: 1)
                outerCircleRadLabel.textColor = UIColor.init(red: red, green: green, blue: blue, alpha: 1)
                innerCircleRadLable.textColor = UIColor.init(red: red, green: green, blue: blue, alpha: 1)
                scaleTextLabel.textColor = UIColor.init(red: red, green: green, blue: blue, alpha: 1)
            drawingColorLabel.textColor = UIColor.init(red: red, green: green, blue: blue, alpha: 1)
    }

    func drawSample(){
        UIGraphicsBeginImageContext(colorSample.frame.size)
        let context = UIGraphicsGetCurrentContext()
        CGContextSetLineCap(context, CGLineCap.Round)
        CGContextSetLineWidth(context, 20)
        CGContextMoveToPoint(context, 45, 45)
        CGContextAddLineToPoint(context, 45, 45)
        CGContextSetRGBStrokeColor(context, red, green, blue, opacity)
        CGContextStrokePath(context)
        colorSample.image = UIGraphicsGetImageFromCurrentImageContext()
        UIGraphicsEndImageContext()
    }

    @IBAction func hideShowCircles(sender: UIButton){
        if(circlesHidden){
            self.view.addSubview(innerCircle)
            mainView.addSubview(outerCircle)
            mainView.addSubview(pencilPoint)
            circlesHidden = false
            dispatch_async(dispatch_get_main_queue(), {sender.highlighted = false})
        } else if(!circlesHidden) {
            innerCircle.removeFromSuperview()
            outerCircle.removeFromSuperview()
            pencilPoint.removeFromSuperview()
            circlesHidden = true
            dispatch_async(dispatch_get_main_queue(), {sender.highlighted = true})
        }
    }
}

class ShapeView: UIView {
    var size: CGFloat!
    let lineWidth: CGFloat = 2
    var fillColor: UIColor!
    var circleRadius: CGFloat!
    var outsideColor: UIColor

    init(origin: CGPoint, radius: CGFloat, fill: UIColor, outsideColor: UIColor){
        self.circleRadius = radius
        self.fillColor = fill
        self.size = (circleRadius * 40) + 2
        self.outsideColor = outsideColor
        super.init(frame: CGRectMake(0.0, 0.0, size, size))
        self.backgroundColor = UIColor.clearColor()
        self.center = origin
        initGestureRecognizers()
    }

    required init(coder aDecoder:NSCoder){
        fatalError("init(coder:) has not been implemented")
```

```
    }

    override func drawRect(rect: CGRect){
        CGRectInset(rect, lineWidth/2, lineWidth/2)
        let path = UIBezierPath(arcCenter: CGPoint(x: size/2,y: size/2),
                radius: CGFloat(20*circleRadius),
                startAngle: CGFloat(0), endAngle:CGFloat(2 * M_PI), clockwise: false)

        self.fillColor.setFill()
        path.fill()
        path.lineWidth = self.lineWidth
        self.outsideColor.setStroke()
        path.stroke()
    }

    func initGestureRecognizers(){
        let rotationGR = UIRotationGestureRecognizer(target: self,
                action: #selector(ShapeView.didRotate(_:)))

        addGestureRecognizer(rotationGR)
    }

    func didRotate(rotationGR: UIRotationGestureRecognizer){
        self.superview!.bringSubviewToFront(self)
        self.transform = CGAffineTransformRotate(self.transform, rotationGR.rotation)
        rotationGR.rotation = 0
    }

    func reposition(moveToThisPoint: CGPoint){
        self.center = moveToThisPoint
    }
}
```