Proteus AMM Engine

Shell v2 White Paper, Part 1

October 13, 2021

Abstract

An AMM engine is a system that can render any trading strategy, any bonding curve. We created an AMM engine, Proteus, by constructing a bonding curve algorithm with flexible and precise geometry. The algorithm works by defining an identity curve based on a generalized conic section. We then scale the pool's balances to the identity curve, compute the swap, then rescale the output. Among other features, this algorithm enables concentrated liquidity with fungible LP tokens. Because of its greater flexibility and precision, Proteus is capable of implementing more capital efficient bonding curves than Stableswap or Uniswap v3. Future research will focus on optimizing the parameters for stablecoin markets and adding the capability to continuously transition from one curve to another. The long term vision is for Proteus to be an abstraction layer between financial engineers and the blockchain, an operating system for trading strategies. Alongside this white paper, we also provide a reference implementation in Solidity, which can be accessed here.

1 Introduction

Liquidity pools are the foundation of decentralized finance (DeFi). Over 75% of all trade volume on decentralized exchanges goes through liquidity pools.(5) A liquidity pool is a smart contract where users, called "liquidity providers" (LPs), pool their capital together and delegate control of their capital to an automated market maker (AMM). An AMM is an algorithm that uses the LPs' capital to implement a trading strategy on their behalf.

The goal of this paper is not to design a capital efficient trading strategy for liquidity pools. Rather, our goal is to engineer a generic AMM algorithm that can implement *any* trading strategy. We call this an "AMM engine". An operating system is an abstraction layer between an application and the hardware. When developers write software applications, they do not need to deal with low-level tasks such as memory management. An AMM engine is an abstraction layer between a financial engineer and the smart contracts running on a blockchain. Writing smart contract systems is inherently difficult because computation is expensive and a single mistake can cost millions if not billions of dollars. Financial engineers should not have to directly interface with this medium in order to implement their trading strategy. Instead, they should rely on AMM engines.

We call our AMM engine, "Proteus." The name comes from the ancient Greek god, a shapeshifter who lives in the sea. He could adopt any form, be it fish, lion or man. Anyone strong enough to wrestle Proteus and hold him in place was entitled to ask the deity a question to which he would always give a true answer.

2 Bonding Curves

In a world of unlimited storage and computation, a trading strategy could be represented as a massive lookup table, a spreadsheet. However, we are operating in an environment where logic is expensive and data storage and retrieval is exorbitant. Storing a single uint256 (about 3% of a kilobyte) can cost in excess of $10.00 on Ethereum mainnet at the time of writing.(7) Therefore, we must embed our AMM's trading strategy in as few bytes as possible, keeping the amount of logic to a minimum.

Bancor (4) and Uniswap (1) pioneered the use of "bonding curves" as a means to compress a pool's trading strategy into a manageable form. To understand how a bonding curve works, we first need to mathematically define a liquidity pool's behavior. A liquidity pool starts with a balance of Token x and Token y. We can define these balances as a point on a Cartesian plane, $P_0 = (x_0, y_0)$. A user wants to trade with the pool by giving some amount of x in return for some amount of y. We denote the trading amounts as Δx and Δy. The goal of the AMM is to calculate Δy given P_0 and Δx.

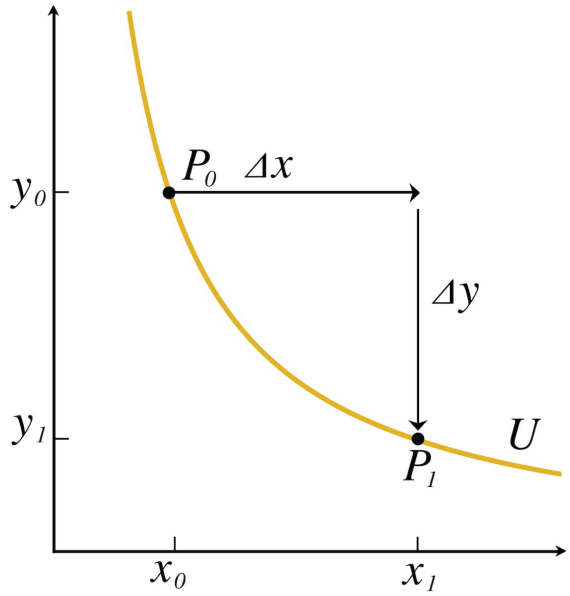

Figure 1: The geometry of swaps and bonding curves

In Figure 1, we have a curve, U, passing through P_0. That is a bonding curve. We can use it to calculate Δy by adding Δx to x_0 to get x_1. Given x_1 and the bonding curve, we can calculate y_1. And finally, $\Delta y = y_0 - y_1$. In that way, we can use the curve to calculate the price of any swap no matter the starting balances or the trade size. As long as the curve has a simple algebraic expression, we can embed the AMM's entire trading strategy in a very compact form. For example, constant product, the algorithm that powers Uniswap v1 and v2, takes the following form: $xy = k$. That simple expression encompasses all possible balances and all possible swap amounts.

3 Curve Geometry and Liquidity Concentration

The geometry of the bonding curve dictates the AMM's trading strategy. For example, the slope of the tangent line (the negative derivative, $-\frac{dy}{dx}$) is the "instantaneous price" of the pool. The instantaneous price is the hypothetical ratio between $\frac{\Delta y}{\Delta x}$ if we were to trade an infinitesimally small amount that leaves the balances unchanged. The analogue from order book exchanges is the spot price.

The curvature (how much a line changes direction) at each instantaneous price ($-\frac{dy}{dx}$) determines the slippage. Slippage is how much the instantaneous price changes from before the swap versus after the swap. The higher the curvature, the higher the slippage.(3) If slippage is zero in a region of the curve, then the bonding curve must be a straight line with no curvature in that region. A "concentrated liquidity" curve has low curvature around the price range of interest. Allocating liquidity and curvature is essential for optimizing an AMM's trading strategy for a given market.

Stableswap (6), invented by Curve Finance, constructs a bonding curve that has low curvature when $\frac{dy}{dx} = -1$ and has progressively higher curvature as the instantaneous price deviates from 1:1 (see Figure 2). Stablecoins are pegged to \$1.00 and often trade at or near 1:1. Hence, Stableswap facilitates low-slippage trades for stablecoins. However, because the bonding curve concentrates liquidity at a certain price point, it must necessarily have less liquidity in other price points. Therefore, stableswap does not work well for volatile assets. Constant product evenly spreads liquidity across all price points. Hence, constant product works for volatile assets but is less optimal for stablecoins.

4 Using Conics as an AMM Engine

Because the geometry of the bonding curve determines the AMM's trading strategy, we need to construct a bonding curve algorithm with flexible geometry if we want to create an AMM engine. There are many families of algebraic equations we can choose from. Indeed, any curve can be approximated by a polynomial expression. We have found that generalized conic sections (see Equation 1) are a good candidate because they are simple to compute yet have a flexible

geometry.

$$Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0 \tag{1}$$

Conic sections encompass lines, parabolas, hyperbolas and ellipses. They can replicate many constant function AMMs such as constant product, constant sum and Balancer Lab's constant value (9). We can even replicate stableswap with an ellipse (see Figure 2). This level of geometric flexibility will allow financial engineers to precisely allocate the AMM's liquidity at arbitrary price points.

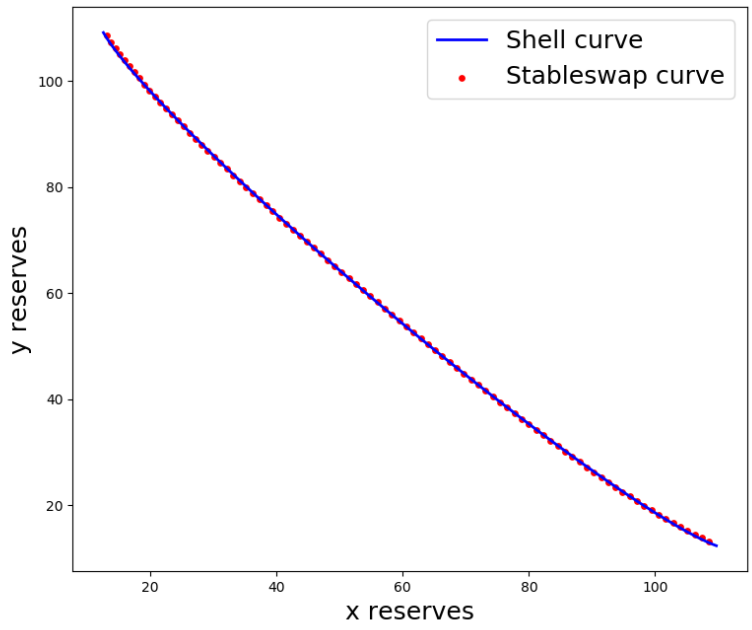

Figure 2: The Shell curve is an ellipse fitted to match the Stableswap curve

However, there is a limitation to using conics or other polynomials to generate flexible bonding curves. As LPs add or remove funds from the pool, the balances change and we must modify the bonding curve. To gain some intuition as to why, consider what would happen if we had two pools with the same bonding curve algorithm. One pool has $100 of capital and the other curve has $1 billion of capital. All things being equal, we would expect to get a better rate swapping $500 through the $1 billion pool versus the $100 pool. Larger pools have less slippage than smaller pools. Ergo, smaller pools will have higher curvature. How do we construct an AMM that can expand and contract arbitrarily shaped bonding curves?

5 The Bonding Curve Invariant

We are not specifying a single curve but a family of curves that are isomorphic to each other. In what way are they isomorphic? When the curve expands, it must preserve an invariant relationship between the instantaneous price and the ratio of the balances. For any $\frac{y}{x}$ there must be one and only one $\frac{dy}{dx}$. Figure 3 represents this invariant geometrically. All balances along the line $\frac{y}{x} = m$ have the same ratio. If two bonding curves, U and $U^{'}$, are isomorphic, then both intersections with that line will have the same instantaneous price. Geometrically, that means the tangent lines at intersection will be parallel.

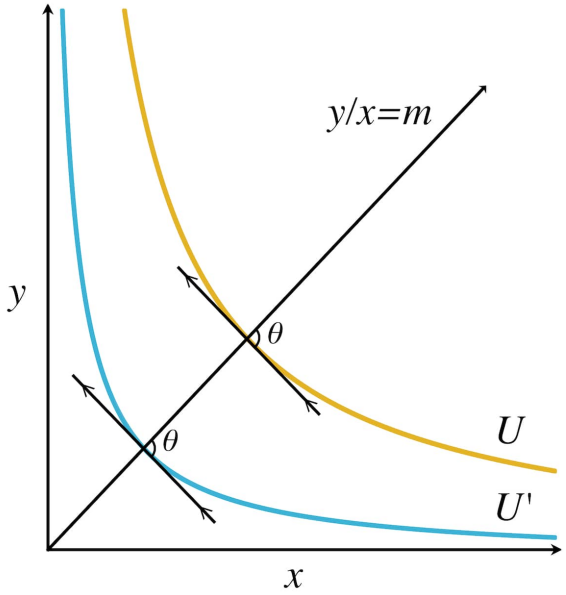

Figure 3: U and $U^{'}$ are isomorphic because their tangent lines are parallel.

If the invariant is violated, then the pool can be flash loan(8) attacked. The attacker takes out a large flash loan and deposits it into the pool, increasing the balances. Because the larger curve is not isomorphic to the smaller curve, the pool now quotes a different price. The attacker then swaps against the pool at the modified price. After the swap, the user withdraws from the pool, extracting value in the process.

How can we achieve flexible geometry while respecting this invariant? Simply fitting an algebraic expression for a single bonding curve is not enough. We need to modify the bonding curve equation as the pool grows. In constant product, $xy = k$, the value for k scales with the size of the pool. While that strategy may work for simple equations, it becomes quite complicated for more flexible geometries. In a conic section, we will have to modify six coefficients (see Equation 1) after each deposit and withdrawal, which is not a trivial problem.

Instead of scaling the curve to the size of the pool, we can instead scale the balances to the

curve. If we can do that, then we can use any algebraic expression we want and not have to worry about modifying coefficients. It turns out that with some basic geometry, we can devise a simple algorithm to do just that.

6 Scaling Balances to the Curve

Figure 4 shows the geometry of two isomorphic curves, U and $U^{'}$, before and after a swap. Both curves start and end with the same balance ratios, m_0 and m_1. For clarity of notation, we labeled seven points of interest: A, B, C, D, E, F, O. These points correspond to the following:

$$A = P_0$$
$$C = P_1$$
$$D = P_0^{'}$$
$$F = P_1^{'}$$
$$O = origin$$
$$AB = \Delta y$$
$$BC = \Delta x$$
$$DE = \Delta y^{'}$$
$$EF = \Delta x^{'}$$

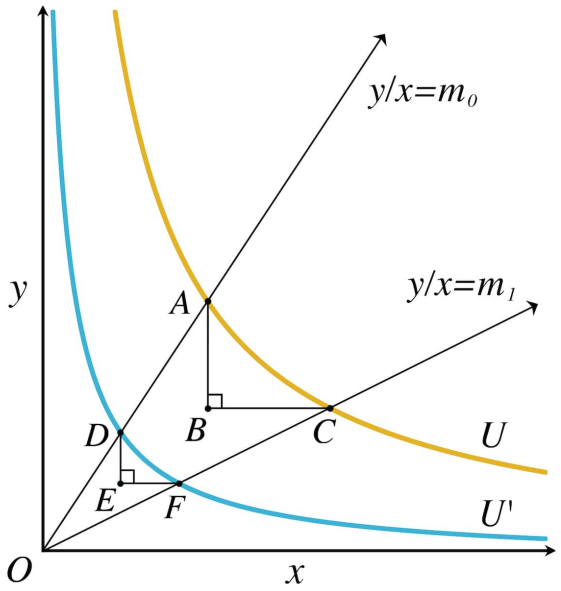

Figure 4: The geometry of swaps and isomorphic bonding curves

Because U and $U^{'}$ are isomorphic, $\triangle ABC \cong \triangle DEF$ and $\triangle OAC \cong \triangle ODF$. That gives us the following equalities:

$$\frac{DE}{AB} = \frac{EF}{BC} = \frac{OD}{OA} = \frac{OF}{OC} = \lambda \tag{2}$$

Where λ is a linear scale factor. From there we can infer:

$$\lambda = \frac{x^{'}}{x} = \frac{\Delta x^{'}}{\Delta x} = \frac{\Delta y^{'}}{\Delta y} = \frac{y^{'}}{y} \tag{3}$$

We can leverage these relationships to scale the pool's balances to a bonding curve. First, we explicitly define the identity or reference curve. If we are using a conic section, we explicitly define the six coefficients (see Equation 1). The identity curve corresponds to $U^{'}$. The second curve, U, is implicit and never specified. Given P_0 and $U^{'}$, we can calculate λ solving a system of equations. Once we know λ, we can calculate $\Delta x^{'}$, $y^{'}$ and finally y_1. This process allows us to use complicated equations as identity curves because we can circumvent curve rescaling. Rather than continually update the curve equation (a hard problem), we keep the curve equation constant and just rescale the balances (an easy problem). The following section will explain how the algorithm works by computing an example swap using the curve specified in Figure 2. We will then give a more general description.

7 Example of Calculating Swaps

We start off knowing three pieces of information (see Figure 5):

1. The starting balances, $P_0 = (25000, 30000)$
2. The swap amount, $\Delta x = 10000$
3. The identity curve equation, $U^{'}$

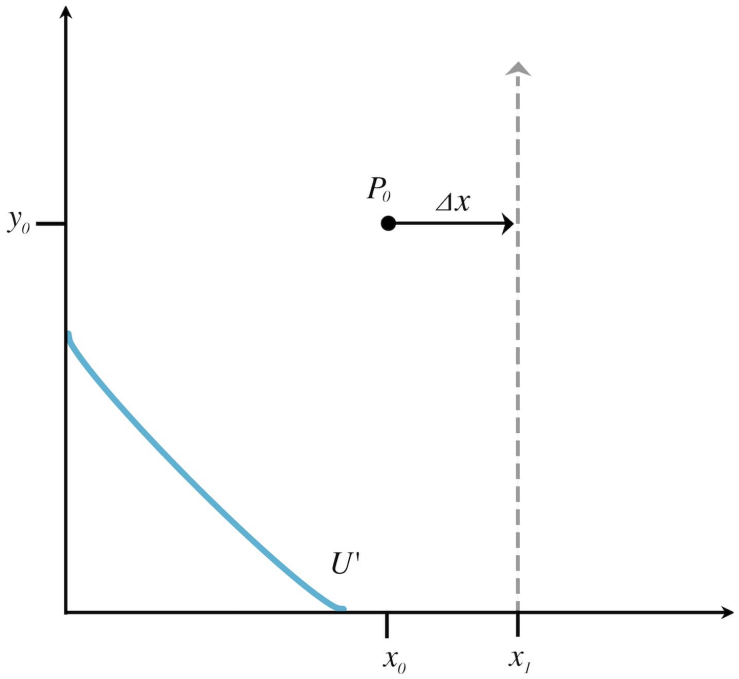

Figure 5: Using only the information in this diagram, we have to calculate the swap amount, Δy.

Our identity curve in this example is a conic section of the same form as Equation 1. The coefficients are:

$$A = 0.7129785111362054$$

$$B = 1.4023717661989632$$

$$C = 0.7129785111362054$$

$$D = -30408.265249329583$$

$$E = -30408.265249329583$$

$$F = 324200000$$

These are the same coefficients as the ellipse fitted to the stableswap curve in Figure 2. Implicitly, there is a second curve, U, scaled out from the identity curve to match the actual balances. Although we represent U graphically in Figure 6, we never explicitly calculate the coefficients of this curve.

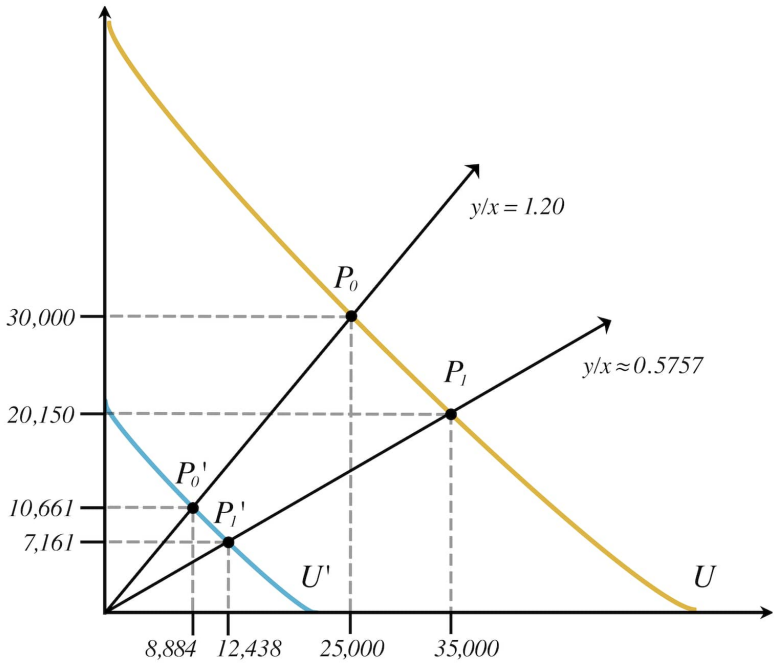

Figure 6: We scale the starting balances at P_0 to $P_0^{'}$. That way, we can use $U^{'}$ and never have to calculate the equation for U.

Our first step is to scale P_0 to the corresponding point on the identity curve, $P_0^{'}$, by finding the intersection between $U^{'}$ and the line $\frac{y}{x} = 1.2$, where 1.2 is the ratio between the y and x balances. Because both P_0 and $P_0^{'}$ have the same ratio between x and y balances, they both have the same instantaneous price (see Figure 3). We have two equations and two unknowns, so finding $P_0^{'}$ is a matter of basic algebra. We should note that $U^{'}$ is an ellipse; thus, there will be two points of intersections, two solutions. Because we are only interested in the lower left portion of the ellipse, we keep the smallest positive real root of $x_0^{'}$. With that in mind, we find:

$$P_0^{'} = (10661, 8884)$$

Now that we know the value for both P_0 and $P_0^{'}$, we can calculate the scale factor, λ, and use it to find the scaled swap amount, $\Delta x^{'}$, and the ending x balance of the identity curve, $x_1^{'}$:

$$\lambda = \frac{x_0^{'}}{x_0} = 0.3554$$

$$\Delta x^{'} = \lambda \Delta x = 3554$$

$$x_1^{'} = x_0^{'} + \Delta x^{'} = 12438$$

By plugging in $x_1^{'}$ to the equation for $U^{'}$, we can find $y_1^{'}$ and hence $P_1^{'}$:

$$P_1^{'} = (12438, 7161)$$

Lastly, we use the scale factor to calculate y_1 and from there, Δy, the amount given to the user:

$$y_1 = \frac{y_1^{'}}{\lambda} = 20150$$

$$\Delta y = y_1 - y_0 = 9850$$

The entire process is represented graphically in Figure 6. We can decompose the general algorithm into the following steps:

1. Start with initial balances, $P_0 = (x_0, y_0)$, and an identity curve, $U^{'}$
2. Compute the corresponding point on the identity curve, $P_0^{'}$, by finding the intersection of the line $y_0/x_0 = m_0$ and $U^{'}$
3. Calculate the scale factor, $\lambda = \frac{x_0^{'}}{x_0}$
4. Scale the swap amount: $\Delta x^{'} = \lambda \Delta x$
5. Calculate the ending point on the identity curve, $P_1^{'}$
6. Rescale $P_1^{'}$ using λ to calculate P_1
7. The swap amount given to the user is: $\Delta y = y_1 - y_0$

8 How to Compare Proteus to Other AMMs

As Angeris et al(3) demonstrated, a bonding curve's geometry is the primary determinate of an AMM's capital efficiency. The way to compare Proteus to other AMMs is in terms of geometric flexibility and precision. Flexibility refers to the range of shapes a bonding curve can take. Precision refers to how closely the actual shape matches the desired shape.

Uniswap v3 (2) requires LPs to allocate liquidity in discrete price bands. If they want to distribute their liquidity in a gradient, where some bands have more liquidity than others, that will require minting separate non-fungible LP tokens. For example, in a stablecoin pool LPs may want to concentrate most of their liquidity around $1.00. But they would also want to distribute some liquidity in the less frequent, but still probable regions: above $1.01 and below $0.99. To create such a position in Uniswap, they would need to mint at least three separate non-fungible tokens (NFTs). And the more precision LPs want to add, the more NFTs they need to mint. Moreover, each NFT represents a discrete price range. A continuous gradient across all prices is not possible.

In contrast, Shell's v2 algorithm has the precision and flexibility to continuously allocate liquidity across a range of prices. And pools use fungible tokens to track LP contributions (see Appendix A). Figure 7 illustrates the difference. An AMM engine with higher precision and flexibility is capable of generating more capital efficient bonding curves because it can allocate capital exactly where it will generate the most liquidity.

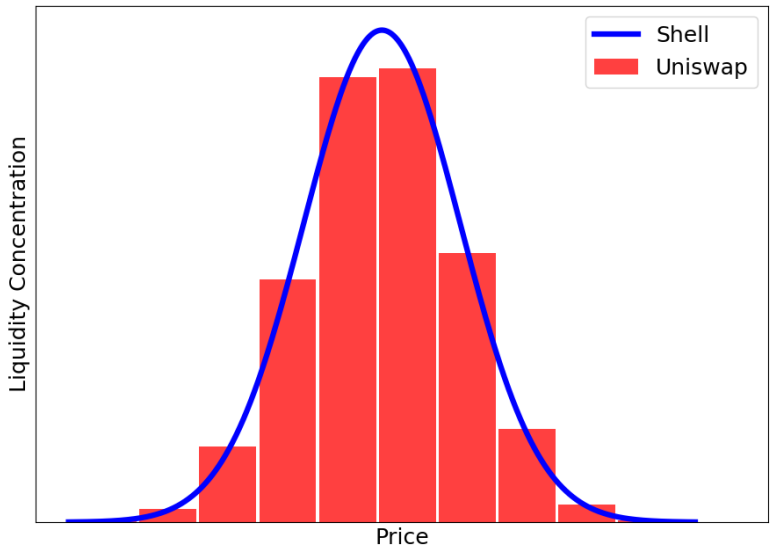

Figure 7: Allocating liquidity across a normal distribution.

9 Potential Use Cases

How might we apply this new algorithm? DeFi protocols often subsidize liquidity pools for their native token by offering additional incentives. This practice is referred to as "liquidity mining." It is advantageous to concentrate liquidity around certain price points to improve the capital efficiency of the pool, and hence the efficiency of the liquidity mining program. Shell's v2 algorithm makes it relatively easy to customize a curve for the liquidity mining pool and because LP tokens are fungible, it is easy to stake the tokens and distribute rewards.

As demonstrated previously in Figure 2, we can use Proteus to fit an ellipse to a Stableswap curve. Is it possible to design a bonding curve that is even more capital efficient for stablecoin trades? Although Stableswap performs well, there is no reason to assume it is at the global maximum of capital efficiency. Prior research done on behalf of Curve Finance (10) showed that market simulations and back testing can be used to find the optimal value of Stableswap's one free parameter, the "amplification coefficient," which determines the level of liquidity concentration. Conic sections have six free parameters, which is both a challenge and an opportunity. On the

one hand, the search space is much larger. On the other hand, there is greater flexibility. Our preliminary results suggest that an ellipse with the right parameters can significantly improve upon Stableswap. The results of this analysis will be the subject of a subsequent paper.

As AMM engines become more sophisticated, average users will lack the specialization to take advantage of the flexibility to design their own bonding curves. A "managed pool," such as Visor Finance (12), designs and executes strategies on behalf LPs. With its flexibility and precision, Proteus will be an excellent substrate for managed pools. The current implementation of our algorithm cannot seamlessly transition from one position to another, one curve to another. Instead, the transition is a discontinuous process. Future iterations of Proteus will enable a continuous evolution from one curve to another. This can be done by defining two identity curves and taking a time-weighted interpolation between them. Not only would this allow a pool manager to adjust to the market in real time, financial engineers can also design curves for time-dependent assets such as bonds and options.

10 Conclusion

An AMM engine is a system that can render any trading strategy, any bonding curve. We created an AMM engine, Proteus, by constructing a bonding curve algorithm with flexible and precise geometry. The algorithm works by defining an identity curve based on a generalized conic section. We then scale the pool's balances to the identity curve, compute the swap, then rescale the output. Because of its greater flexibility and precision, Proteus is capable of implementing more capital efficient bonding curves than Stableswap of Uniswap v3. Future research will focus on optimizing the parameters for stablecoin markets and adding the capability to continuously transition from one curve to another. The long term vision is for Proteus to be an abstraction layer between financial engineers and the blockchain, an operating system for trading strategies.

Creating a fully realized bonding curve engine will require many iterations and a diverse group of contributors. To that end, we want our work to not only be open source, but also as accessible as possible. We have created a reference implementation of our algorithm in Solidity. It can be accessed here. We welcome your feedback.

References

[1] Adams, Hayden (2019). "Uniswap White Paper." https://hackmd.io/@HaydenAdams/HJ9jLsfTz

[2] Adams et al (March, 2021). "Uniswap v3 Core." https://uniswap.org/whitepaper-v3.pdf

[3] Angeris et al (2020, December). "When does the tail wag the dog? Curvature and market making." https://arxiv.org/abs/2012.08040

[4] Bancor Network (2021, October 5). "Introduction to Bancor." https://docs.bancor.network/

[5] DeFi Pulse (2021, October 5). "DEX Tracker - Decentralized Exchanges Trading Volume." https://defiprime.com/dex-volume

[6] Egorov, Michael (2019, November 10). "StableSwap - efficient mechanism for Stablecoin liquidity." https://curve.fi/files/stableswap-paper.pdf

[7] ETH Gas Station (2021, October 4). "ETH Gas Station: Tx Tracker." https://www.ethgasstation.info/calculatorTxV.php

[8] Hertig, Alyssa (2021, February 17). "What Is a Flash Loan? A guide to one of DeFi's most innovative and controversial features." https://www.coindesk.com/learn/2021/02/17/what-is-a-flash-loan/

[9] Martinelli, Fernando and Nikolai Mushegian (2019, September 19). https://balancer.fi/whitepaper.pdf

[10] Nagaking (2021, April 12). "A Parameter Research [Demo and Code]." https://gov.curve.fi/t/a-parameter-research-demo-and-code/1622

[11] Shell Protocol (2020, October 1). "Shell Protocol White Paper." https://github.com/cowri/shell-solidity-v1/blob/master/Shell_White_Paper_v1.0.pdf

[12] Visor Finance (2021, October 12). "What is the Visor Protocol?" https://www.visor.finance/about

A Deposits and Withdrawals

When an LP deposits funds to the pool, they receive another token in return to keep track of their contribution. We call these LP tokens "shells" because they are containers for the value held in the pool, much like a living shell is a container for the organism inside. When an LP wants to withdraw their funds from the pool, they must redeem their shells. The purpose of this section is to describe how the issuance and redemption of shells is calculated using our algorithm.

In order to calculate deposits and withdrawals, we need to introduce a new concept, "utility". We discuss utility at length in our Shell Protocol v1 white paper.(11) Utility is the pool's internal measurement of value. A utility function takes the pool's balances, (x, y), and maps them to a scalar value, U. I.e. $u(x, y) \to U$. When a user makes a deposit, the utility increases because the pool now holds more tokens. When a user makes a withdrawal, the utility decreases because the pool holds fewer tokens. The amount of shells issued or redeemed is directly proportional to the change in the pool's utility. If a deposit causes the pool's utility to increase by 10%, then the pool will issue new shells to the user such that the total supply increases 10%. If a withdrawal will cause utility to decrease by 5%, then the user must return to the pool 5% of the total supply. We can represent this relationship mathematically:

$$\Delta s = \frac{U_1 - U_0}{U_0} s_0 \tag{4}$$

Where s is the shell supply, u_0 is the utility before the transaction and u_1 is the utility after the transaction.

There are many methods to construct a utility function, but there are some fundamental constraints. First, utility must be monotonic:

$$u(x) < u(x + a), \forall a > 0$$

That means when we add tokens, the utility only ever increases. We cannot decrease utility by depositing tokens. Conversely, we cannot increase utility by withdrawing tokens. Second, utility must be a linear function:

$$u(bx, by) = bu(x, y), \forall b > 0$$

If we increase all the balances by a constant factor, b, then the pool's utility must also increase by the same factor. So if the pool balances double from $(x, y) = (100, 150) \to (x, y) = (200, 300)$, then the utility must also double. Lastly, swaps are a special case of deposits and withdrawals where the utility does not change. Any swap can be deconstructed as a deposit followed by a withdrawal. Therefore, the utility function and the bonding curve used to calculate swaps are part of the same underlying structure. Swaps must respect the utility function, and deposits/withdrawals must respect the bonding curve. If we obey these constraints, then shells will be fungible LP tokens that can be used as a currency in their own right.

In our algorithm, we define utility as the Euclidean distance between the point where the line $\frac{y}{x} = 1$ intersects the implicit bonding curve, U (see Sections 6 and 7), and the origin. We refer

to this intersection point as $P^* = (x^*, y^*)$. Figure 8 below demonstrates graphically how we can calculate the utility of the pool when balances are equal to P_0.

Because we do not explicitly calculate U, we must use the identity curve, $U^{'}$. We can calculate the utility of the identity curve, $d^{'}$ using basic algebra. Using the scale factor, λ, we can scale the identity utility to the actual utility:

$$d = \frac{d^{'}}{\lambda}$$

With a viable utility function, we can apply Equation 4 to calculate how many shells to issue or redeem based on the percent change in utility, i.e. the percent change in d.

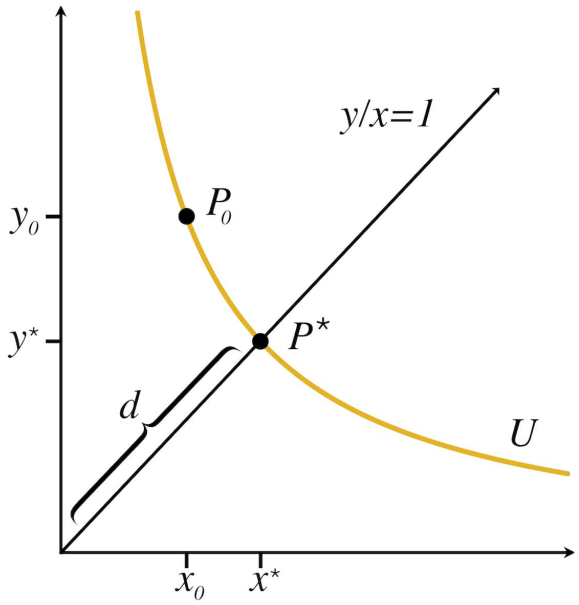

Figure 8: The utility of the pool at P_0 is the Euclidean distance, d, from the origin to P^*.

B Notes on Solidity Implementation

In this paper, we tried to present the algorithm in the most intuitive way possible. The actual implementation of the algorithm in Solidity, while adhering to the same concept, has some modifications worth explaining in more detail. The Solidity implementation is geared toward efficiency and auditability.

Let us define the following three functions:

$$f(x, y) = x^*$$

$$g(x, x^*) = y$$

$$h(y, x^*) = x$$

The function, f, is equivalent to the utility function presented in Appendix A. The only difference is that f does not return the Euclidean distance from the origin, but the balance of x^*. Regardless of whether we use d or x^* to calculate utility, we will end up with the same value for Δs in Equation 4. Computing d requires extra calculations whereas x^* does not.

The function, g, takes the pool's x balance and current utility as inputs and calculates the corresponding y balance. The function, h, takes the pool's y balance and current utility and calculates the corresponding x balance.

To compute deposits, we only need to use f. However, we need g and h to compute swaps and withdrawals. If a user is swapping x in return for y, then we use the following nested functions:

$$\Delta y = g(x_0 + \Delta x, f(x_0, y_0))) - y_0$$

For swaps going the other direction, with the user giving y in return for x:

$$\Delta x = h(y_0 + \Delta y, f(x_0, y_0)) - x_0$$

If a user is making a withdrawal, they provide the number of shells they would like to burn along with the token they would like to receive. The formula for calculating the withdrawal amount of Δx and Δy are given as:

$$\Delta y = (y_0 - g(x_0, f(x_0, y_0)\frac{\Delta s}{s_0}))$$

$$\Delta x = (x_0 - h(y_0, f(x_0, y_0)\frac{\Delta s}{s_0}))$$