Project description

Substrate-based runtime version of Curve automated market maker will be a 7 week long project which will deliver a following functionality:

- Low slippage, high efficiency stablecoin exchange
- High efficiency exchange tool for other homogeneous assets on Polkadot (e.g. wrapped assets)
- Low risk fee income for liquidity providers.
- Liquidity superfluidity with additional rewards from supplying liquidity to lending protocols such as Equilibrium and Acala.

Curve AMM is of paramount importance to the entire Polkadot ecosystem. With the introduction of parachains and interconnection of different Polka-based projects the issue of multiple wrapped assets representing the same underlying assets arises:

Consider ETH, for example. There are multiple bridging solutions who promise to introduce wrapped-ETH and other ERC-20 tokens to Polkadot. There needs to be a way to manage or exchange all of these representations of the same underlying asset inside Polkadot with low cost and low slippage, and that is where the Curve AMM comes into play.

Curve's unique *stableswap* invariant utilizes liquidity much more efficiently compared to all existing DEXes for stablecoins at already several hundred USD TVL (total value locked). Since initial liquidity on Polkadot is hardly going to be very large, proposed efficiency is VERY important for the ecosystem to flourish.

Development Roadmap

Milestone 1 - Initial implementation, pool manipulations, invariant calculation - 3 weeks - 15 working days - $15000, $1000/day

- Assets will be handled using a new assets trait.
- We will implement a pool storage structure for handling different asset pools with different parameters.
- We will implement methods to set up custom asset pools.
- We will implement a method to iteratively calculate Curve's invariant D and points on the bonding curve in non-overflowing integer operations.
- The code will have complete unit-test coverage to ensure functionality and robustness.
- We will publish the code in the Equilibrium's public GitHub repository.

Milestone 2 - actual assets exchange, - 3 weeks - 15 working days - $15000, $1000/day

- We will implement methods to work with asset pools: add liquidity / remove liquidity.

- We will implement methods to exchange assets within pools.
- We will implement liquidity superfluidity: assets locked inside Curve liquidity pools may be further used in various lending protocols across the Polkadot ecosystem.
- We will implement a mechanism to reward liquidity providers with LP tokens.
- The code will have complete unit-test coverage to ensure functionality and robustness.
- We will publish the code in the Equilibrium's public GitHub repository.
- We will publish the code in the Equilibrium's public GitHub repository.
- We will provide detailed technical documentation describing pallet logic, storage, interfaces, subscriptions.

Business Logic

Pools

Each pool will be represented by a pool_id address with following parameters:

account_id: id of the Liquidity Pool.
assets: list of multiassets supported by the pool
pool_token: LP multiasset
amplification: initial amplification coefficient (leverage)
fee: amount of fee pool charges for the exchange.

Anyone willing to create a pool, by depositing a minimum amount of tokens.

User actions

Following extrinsics will be available for communicating with the pool:

1. *add_lqiuidity* - user deposits asset/assets, system recalculates exchange invariant D, calculates user share of the pool, calculates how much pool tokens to mint and mints this amount of tokens to the user.
2. *remove_liquidity* - user withdraws asset/assets by burning corresponding amount of LP tokens. Users have 2 choices, either withdraw assets proportionally to the current pool weights, or withdraw in an imbalanced fashion, which leads to recalculation of the exchange invariant D.
3. *exchange*: - user exchanges i-th asset of the pool for j-th asset of the pool, system calculates the exchange rates (amount of j-th asset to sell given amount of i-th asset it is buying) and transfers j-th asset to the user.

Curve's unique invariant

Detailed technical description of Curve's approach for minimizing the price slippage when swapping "identical" assets is described in [stableswap whitepaper]

When portfolio of virtually identical (e.g. different stablecoins, or different wrapped representations of BTC inside polkadot) is in balance the exchange price between them should be close to 1 (constant-price invariant), but when portfolio moves out of balance, the price should reflect that by switching to the hyperbola-style constant-product invariant.

$\sum_i x_i = D$ - constant price invariant

$\prod_i x_i = \left(\frac{D}{n}\right)^n$ - constant product invariant

The constant D here is the total amount of tokens when they're in balance (equal amounts with equal price).

We want to construct some linear combination of the above two invariants. Let's multiply constant price invariant by leverage parameter χ and add it to the constant product invariant, this way we will have an invariant which is constant-product when $\chi = 0$, and constant price when $\chi >> 0$ Further, to make invariant dimensionless we multiply constant price part by $\chi * D^{n-1}$ yielding the following equation:

$$\chi * D^{n-1} * \sum_i x_i + \prod_i x_i = \chi * D^n + \left(\frac{D}{n}\right)^n$$

χ needs to be dynamic: when the portfolio is in balance, it should equal some amplification coefficient A, but when it's out of balance it should fall off to zero.

$$\chi = \frac{A\prod_i x_i}{\left(\frac{D}{n}\right)^n}$$

Substituting this to the equation above and rearranging the terms, we come up at the final invariant equation:

$$An^n \sum_i x_i + D = ADn^n + D^{n+1} / (n^n \prod_i x_i)$$

When we have a portfolio of assets x[1..n], we need to calculate D and hold the equation above true when performing trades (swapping x(i) for x(j)), this is done by numerically solving the equation above either for D of for x(j).

Technical specification

Custom dependencies. We need to operate with custom Assets, so we will create this trait and simple implementation for it. Other projects can add adapters to adapt their realization.

```
Trait Assets<AssetId, Balance, AccountId> {
	fn create_asset() -> Result<AssetId, Error>;
	fn mint(asset: AssetId, dest: AccountId, amount: Balance) -> Result<(), Error>;
	fn burn(asset: AssetId, dest: AccountId, amount: Balance) -> Result<(), Error>;
	fn transfer(asset: AssetId, source: AccountId, dest: AccountId, amount: Balance) ->
Result<(), Error>;

	fn balance(asset: AssetId, who: AccountId) -> Balance // returns available balance for
transfers
	fn total_issuance(asset: AssetId) -> Balance // total issuance of the Asset
}
```

Data Model

Constants
const MODULE_ID: ModuleId // Module Id for sub account creation (each pool should have unique AccountId)

Pallet Trait

Type AssetId // identificator type of Asset
Type Balance // The balance of an account
Type Assets: Assets<AssetId, Balance, AccountId>
Type Currency // standart balances pallet for utility token or adapter
Type CreationFee: Get<Balance> // anti ddos fee for pool creation
Type OnUnbalanced: OnUnbalanced // what to do with fee (burn, transfer to treasury, etc)
Type ModuleId: ModuleId // module account

Structs

PoolInfo {

PoolAsset: AssetId // LP multiasset
Assets: Vec<AssetId> // list of multiassets supported by the pool
Amplification: FixedI128 // initial amplification coefficient (leverage)
Fee: Permill // amount of fee pool charges for the exchange.
}

Storage

PoolCount: u32 // Current pools count
Pool map(u32) -> PoolInfo // all pools infos

Module

CreatePool(assets: Vec<AssetId>) // creates pool, taking creation fee from the caller

AddLiquidity(poolId: u32, asset: AssetId, amount: Balance) // adding liquidity to the pool (transfer tokens from the caller to pool account), minting corresponding pool tokens to the caller

RemoveLiquidity(poolId: u32, asset: AssetId, amount: Balance) // removing liquidity to the pool (transfer tokens from liquidity pool account to the caller), burning corresponding pool tokens from the caller

Exchange(poolId: u32, asset_from: AssetId, asset_to: AssetId, amount: Balance) // transferring “amount” “asset_from” tokens from the caller and then transferring corresponding amount of “asset_to” tokens to the caller

ETH reference implementation

https://github.com/curvefi/curve-contract/blob/master/contracts/pool-templates/base/SwapTemplateBase.vy

Team

There will be 5 team-members working on this project:

Business/Technical Analyst
Senior Developer
Middle Developer
Junior Developer
QA Engineer

FTE table (based on the 40-hour workweek)

Team member	**Full time equivalent**
Business/Technical Analyst	0.5
Senior Developer	0.5
Middle Developer	1
Junior Developer	1
QA Engineer	0.5