Curve Documentation

Release 1.0.0

Curve.fi

Aug 01, 2021

CONTENTS

Curve is an exchange liquidity pool on Ethereum. Curve is designed for extremely efficient stablecoin trading and low risk, supplemental fee income for liquidity providers, without an opportunity cost.

This documentation outlines the technical implementation of the core Curve protocol and related smart contracts. It may be useful for contributors to the Curve codebase, third party integrators, or technically proficient users of the protocol.

Non-technical users may prefer the Resources section of the main Curve website.

Note: All code starting with `$` is meant to be run on your terminal. Code starting with `>>>` is meant to run inside the Brownie console.

Note: This project relies heavily upon `brownie` and the documentation assumes a basic familiarity with it. You may wish to view the Brownie documentation if you have not used it previously.

CHAPTER

ONE

PROCOTOL OVERVIEW

Curve can be broadly separated into the following categories:

- *StableSwap*: Exchange contracts and core functionality of the protocol
- The *DAO*: Protocol governance and value accrual
- The *Factory*: Permissionless deployment of Curve metapools
- The Registry: Standardized API and on-chain resources to aid 3rd party integrations

CHAPTER

TWO

CURVE STABLESWAP EXCHANGE: OVERVIEW

Curve achieves extremely efficient stablecoin trades by implementing the StableSwap invariant, which has significantly lower slippage for stablecoin trades than many other prominent invariants (e.g., constant-product). Note that in this context *stablecoins* refers to tokens that are stable representations of one another. This includes, for example, USD-pegged stablecoins (like DAI and USDC), but also ETH and sETH (synthetic ETH) or different versions of wrapped BTC. For a detailed overview of the StableSwap invariant design, please read the official StableSwap whitepaper.

A Curve pool is essentially a smart contract that implements the StableSwap invariant and therefore contains the logic for exchanging stable tokens. However, while all Curve pools implement the StableSwap invariant, they may come in different pool flavors.

In its simplest form, a Curve pool is an implementation of the StableSwap invariant with 2 or more tokens, which can be referred to as a *plain pool*. Alternative and more complex pool flavors include pools with lending functionality, so-called *lending pools*, as well as *metapools*, which are pools that allow for the exchange of one or more tokens with the tokens of one or more underlying base pools.

Curve also integrates with Synthetix to offer cross-asset swaps.

All exchange functionality that Curve supports, as well as noteworthy implementation details, are explained in technical depth in this section.

CHAPTER

THREE

CURVE STABLESWAP: POOLS

A Curve pool is a smart contract that implements the StableSwap invariant and thereby allows for the exchange of two or more tokens.

More broadly, Curve pools can be split into three categories:

- `Plain pools`: a pool where two or more stablecoins are paired against one another.
- `Lending pools`: a pool where two or more *wrapped* tokens (e.g., `cDAI`) are paired against one another, while the underlying is lent out on some other protocol.
- `Metapools`: a pool where a stablecoin is paired against the LP token from another pool.

Source code for Curve pools may be viewed on GitHub.

Warning: The API for plain, lending and metapools applies to all pools that are implemented based on pool templates. When interacting with older Curve pools, there may be differences in terms of visibility, gas efficiency and/or variable naming. Furthermore, note that older contracts use `vyper 0.1.x...` and that the getters generated for public arrays changed between `0.1.x` and `0.2.x` to accept `uint256` instead of `int128` in order to handle the lookups.

Please **do not** assume for a Curve pool to implement the API outlined in this section but verify this before interacting with a pool contract.

For information on code style please refer to the official *style guide*.

3.1 Plain Pools

The simplest Curve pool is a plain pool, which is an implementation of the StableSwap invariant for two or more tokens. The key characteristic of a plain pool is that the pool contract holds **all** deposited assets at all times.

An example of a Curve plain pool is 3Pool, which contains the tokens `DAI`, `USDC` and `USDT`.

Note: The API of plain pools is also implemented by lending and metapools.

The following Brownie console interaction examples are using EURS Pool. The template source code for plain pools may be viewed on GitHub.

Note: Every pool has the constant private attribute `N_COINS`, which is the number of coins in the pool. This is referred to by several pool methods in the API.

3.1.1 Getting Pool Info

`StableSwap.`**`coins`**`(`*i: uint256*`)` → address: view
Getter for the array of swappable coins within the pool.

```
>>> pool.coins(0)
'0xdB25f211AB05b1c97D595516F45794528a807ad8'
```

`StableSwap.`**`balances`**`(`*i: uint256*`)` → uint256: view
Getter for the pool balances array.

```
>>> pool.balances(0)
2918187395
```

`StableSwap.`**`owner`**`()` → address: view
Getter for the admin/owner of the pool.

```
>>> pool.owner()
'0xeCb456EA5365865EbAb8a2661B0c503410e9B347'
```

`StableSwap.`**`lp_token`**`()` → address: view
Getter for the *LP token* of the pool.

```
>>> pool.lp_token()
'0x194eBd173F6cDacE046C53eACcE9B953F28411d1'
```

Note: In older Curve pools `lp_token` may **not** be `public` and thus not visible.

`StableSwap.`**`A`**`()` → uint256: view
The *amplification coefficient* for the pool.

```
>>> pool.A()
100
```

`StableSwap.`**`A_precise`**`()` → uint256: view
The *amplification coefficient* for the pool not scaled by `A_PRECISION` (`100`).

```
>>> pool.A_precise()
10000
```

`StableSwap.`**`get_virtual_price`**`()` → uint256: view
The current price of the pool LP token relative to the underlying pool assets. Given as an integer with 1e18 precision.

```
>>> pool.get_virtual_price()
1001692838188850782
```

`StableSwap.`**`fee`**`()` → uint256: view
The pool swap fee, as an integer with 1e10 precision.

```
>>> pool.fee()
4000000
```

`StableSwap.`**`admin_fee`**`()` → uint256: view
The percentage of the swap fee that is taken as an admin fee, as an integer with with 1e10 precision.

Admin fee is set at 50% (5000000000) and is paid out to veCRV holders (see *Fee Collection and Distribution*).

```
>>> pool.admin_fee()
5000000000
```

3.1.2 Making Exchanges

StableSwap.**get_dy**(*i: int128*, *j: int128*, *_dx: uint256*) → uint256: view

Get the amount of coin `j` one would receive for swapping `_dx` of coin `i`.

```
>>> pool.get_dy(0, 1, 100)
996307731416690125
```

Note: In the `EURS Pool`, the decimals for `coins(0)` and `coins(1)` are 2 and 18, respectively.

StableSwap.**exchange**(*i: int128*, *j: int128*, *_dx: uint256*, *_min_dy: uint256*) → uint256

Perform an exchange between two coins.

- `i`: Index value for the coin to send
- `j`: Index value of the coin to receive
- `_dx`: Amount of `i` being exchanged
- `_min_dy`: Minimum amount of `j` to receive

Returns the actual amount of coin `j` received. Index values can be found via the `coins` public getter method.

```
>>> expected = pool.get_dy(0, 1, 10**2) * 0.99
>>> pool.exchange(0, 1, 10**2, expected, {"from": alice})
```

3.1.3 Adding/Removing Liquidity

StableSwap.**calc_token_amount**(*_amounts: uint256[N_COINS]*, *_is_deposit: bool*) → uint256: view

Calculate addition or reduction in token supply from a deposit or withdrawal.

- `_amounts`: Amount of each coin being deposited
- `_is_deposit`: Set True for deposits, False for withdrawals

Returns the expected amount of LP tokens received. This calculation accounts for slippage, but not fees.

```
>>> pool.calc_token_amount([10**2, 10**18], True)
1996887509167925969
```

StableSwap.**add_liquidity**(*_amounts: uint256[N_COINS]*, *_min_mint_amount: uint256*) → uint256

Deposit coins into the pool.

- `_amounts`: List of amounts of coins to deposit
- `_min_mint_amount`: Minimum amount of LP tokens to mint from the deposit

Returns the amount of LP tokens received in exchange for the deposited tokens.

StableSwap.**remove_liquidity**(*_amount: uint256*, *_min_amounts: uint256[N_COINS]*) → uint256[N_COINS]

Withdraw coins from the pool.

- `_amount`: Quantity of LP tokens to burn in the withdrawal

- `_min_amounts`: Minimum amounts of underlying coins to receive

Returns a list of the amounts for each coin that was withdrawn.

`StableSwap.`**`remove_liquidity_imbalance`**`(`*_amounts: uint256[N_COINS], _max_burn_amount: uint256*`)` → uint256

Withdraw coins from the pool in an imbalanced amount.

- `_amounts`: List of amounts of underlying coins to withdraw
- `_max_burn_amount`: Maximum amount of LP token to burn in the withdrawal

Returns actual amount of the LP tokens burned in the withdrawal.

`StableSwap.`**`calc_withdraw_one_coin`**`(`*_token_amount: uint256, i: int128*`)` → uint256

Calculate the amount received when withdrawing a single coin.

- `_token_amount`: Amount of LP tokens to burn in the withdrawal
- `i`: Index value of the coin to withdraw

`StableSwap.`**`remove_liquidity_one_coin`**`(`*_token_amount: uint256, i: int128, _min_amount: uint256*`)` → uint256

Withdraw a single coin from the pool.

- `_token_amount`: Amount of LP tokens to burn in the withdrawal
- `i`: Index value of the coin to withdraw
- `_min_amount`: Minimum amount of coin to receive

Returns the amount of coin `i` received.

3.2 Lending Pools

Curve pools may contain lending functionality, whereby the underlying tokens are lent out on other protocols (e.g., Compound or Yearn). Hence, the main difference to a plain pool is that a lending pool does **not** hold the underlying token itself, but a **wrapped** representation of it.

Currently, Curve supports the following lending pools:

- `aave`: Aave pool, with lending on Aave
- `busd`: BUSD pool, with lending on yearn.finance
- `compound`: Compound pool, with lending on Compound
- `ib`: Iron Bank pool, with lending on Cream
- `pax`: PAX pool, with lending on yearn.finance
- `usdt`: USDT pool, with lending on Compound
- `y`: Y pool, with lending on yearn.finance

An example of a Curve lending pool is Compound Pool, which contains the wrapped tokens `cDAI` and `cUSDC`, while the underlying tokens `DAI` and `USDC` are lent out on Compound. Liquidity providers of the Compound Pool therefore receive interest generated on Compound in addition to fees from token swaps in the pool.

Implementation of lending pools may differ with respect to how wrapped tokens accrue interest. There are two main types of wrapped tokens that are used by lending pools:

- `cToken-style tokens`: These are tokens, such as interest-bearing cTokens on Compound (e.g., `cDAI`) or on yTokens on Yearn, where interest accrues as the rate of the token increases.

- `aToken-style tokens`: These are tokens, such as aTokens on AAVE (e.g., `aDAI`), where interest accrues as the balance of the token increases.

The template source code for lending pools may be viewed on GitHub.

Note: Lending pools also implement the API from *plain pools*.

3.2.1 Getting Pool Info

`StableSwap.`**`underlying_coins`**(*i: uint256*) → address: view
Getter for the array of **underlying** coins within the pool.

```
>>> lending_pool.coins(0)
'0x5d3a536E4D6DbD6114cc1Ead35777bAB948E3643'
>>> lending_pool.coins(1)
'0x39AA39c021dfbaE8faC545936693aC917d5E7563'
```

3.2.2 Making Exchanges

Like plain pools, lending pools have the `exchange` method. However, in the case of lending pools, calling `exchange` performs a swap between two **wrapped** tokens in the pool.

For example, calling `exchange` on the Compound Pool, would result in a swap between the wrapped tokens `cDAI` and `cUSDC`.

`StableSwap.`**`exchange_underlying`**(*i: int128*, *j: int128*, *dx: uint256*, *min_dy: uint256*) → uint256
Perform an exchange between two **underlying** tokens. Index values can be found via the `underlying_coins` public getter method.

- `i`: Index value for the underlying coin to send
- `j`: Index value of the underlying coin to receive
- `_dx`: Amount of *i* being exchanged
- `_min_dy`: Minimum amount of *j* to receive

Returns the actual amount of coin `j` received.

Note: Older Curve lending pools may not implement the same signature for `exchange_underlying`. For instance, Compound pool does not return anything for `exchange_underlying` and therefore costs more in terms of gas.

3.2.3 Adding/Removing Liquidity

The function signatures for adding and removing liquidity to a lending pool are *mostly* the same as for a *plain pool*. However, for lending pools, liquidity is added and removed in the **wrapped** token, not the underlying.

In order to be able to add and remove liquidity in the underlying token (e.g., remove DAI from Compound Pool instead of cDAI) there exists a `Deposit<POOL>.vy` contract (e.g., (DepositCompound.vy).

Warning: Older Curve lending pools (e.g., Compound Pool) **do not** implement all plain pool methods for *adding and removing liquidity*. For instance, `remove_liquidity_one_coin` is not implemented by Compound Pool).

Some newer pools (e.g., IB) have a modified signature for `add_liquidity` and allow the caller to specify whether the deposited liquidity is in the wrapped *or* underlying token.

`StableSwap.`**`add_liquidity`**(*_amounts: uint256[N_COINS]*, *_min_mint_amount: uint256*, *_use_underlying: bool = False*) → uint256

Deposit coins into the pool.

- `_amounts`: List of amounts of coins to deposit
- `_min_mint_amount`: Minimum amount of LP tokens to mint from the deposit
- `_use_underlying` If `True`, deposit underlying assets instead of wrapped assets.

Returns amount of LP tokens received in exchange for the deposited tokens.

3.3 Metapools

A metapool is a pool where a stablecoin is paired against the LP token from another pool, a so-called *base pool*.

For example, a liquidity provider may deposit `DAI` into 3Pool and in exchange receive the pool's LP token `3CRV`. The `3CRV` LP token may then be deposited into the GUSD metapool, which contains the coins `GUSD` and `3CRV`, in exchange for the metapool's LP token `gusd3CRV`. The obtained LP token may then be staked in the metapool's liquidity gauge for `CRV` rewards.

Metapools provide an opportunity for the base pool liquidity providers to earn additional trading fees by depositing their LP tokens into the metapool. Note that the `CRV` rewards received for staking LP tokens into the pool's liquidity gauge may differ for the base pool's liquidity gauge and the metapool's liquidity gauge. For details on liquidity gauges and protocol rewards, please refer to *Liquidity Gauges and Minting CRV*.

Note: Metapools also implement the API from *plain pools*.

3.3.1 Getting Pool Information

`StableSwap.`**`base_coins`**(*i: uint256*) → address: view

Get the coins of the base pool.

```
>>> metapool.base_coins(0)
'0x6B175474E89094C44Da98b954EedeAC495271d0F'
>>> metapool.base_coins(1)
'0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48'
```

(continues on next page)

(continued from previous page)

```
>>> metapool.base_coins(2)
'0xdAC17F958D2ee523a2206206994597C13D831ec7'
```

`StableSwap.`**`coins`**(*i: uint256*) → address: view
Get the coins of the metapool.

```
>>> metapool.coins(0)
'0x056Fd409E1d7A124BD7017459dFEa2F387b6d5Cd'
>>> metapool.coins(1)
'0x6c3F90f043a72FA612cbac8115EE7e52BDe6E490'
```

In this console example, `coins(0)` is the metapool's coin (`GUSD`) and `coins(1)` is the LP token of the base pool (`3CRV`).

`StableSwap.`**`base_pool`**`()` → address: view
Get the address of the base pool.

```
>>> metapool.base_pool()
'0xbEbc44782C7dB0a1A60Cb6fe97d0b483032FF1C7'
```

`StableSwap.`**`base_virtual_price`**`()` → uint256: view
Get the current price of the base pool LP token relative to the underlying base pool assets.

Note that the base pool's virtual price is only fetched from the base pool *if* the cached price has expired. A fetched based pool virtual price is cached for 10 minutes (`BASE_CACHE_EXPIRES: constant(int128) = 10 * 60`).

```
>>> metapool.base_virtual_price()
1014750545929625438
```

`StableSwap.`**`base_cache_update`**`()` → uint256: view
Get the timestamp at which the base pool virtual price was last cached.

```
>>> metapool.base_cache_updated()
1616583340
```

3.3.2 Making Exchanges

Similar to lending pools, on metapools exchanges can be made either between the coins the metapool actually holds (another pool's LP token and some other coin) *or* between the metapool's underlying coins. In the context of a metapool, **underlying** coins refers to the metapool's coin and any of the base pool's coins. The base pool's LP token is **not** included as an underlying coin.

For example, the GUSD metapool would have the following:

- Coins: `GUSD`, `3CRV` (3Pool LP)
- Underlying coins: `GUSD`, `DAI`, `USDC`, `USDT`

Note: While metapools contain public getters for `coins` and `base_coins`, there exists **no** getter for obtaining a list of all underlying coins.

`StableSwap.`**`exchange`**(*i: int128*, *j: int128*, *_dx: uint256*, *_min_dy: uint256*) → uint256
Perform an exchange between two (non-underlying) coins in the metapool. Index values can be found via the `coins` public getter method.

- `i`: Index value for the coin to send
- `j`: Index valie of the coin to receive
- `_dx`: Amount of `i` being exchanged
- `_min_dy`: Minimum amount of `j` to receive

Returns the actual amount of coin `j` received.

`StableSwap.`**`exchange_underlying`**(*i: int128*, *j: int128*, *_dx: uint256*, *_min_dy: uint256*) → uint256
Perform an exchange between two underlying coins. Index values are the `coins` followed by the `base_coins`, where the base pool LP token is **not** included as a value.

- `i`: Index value for the underlying coin to send
- `j`: Index valie of the underlying coin to recieve
- `_dx`: Amount of `i` being exchanged
- `_min_dy`: Minimum amount of underlying coin `j` to receive

Returns the actual amount of underlying coin `j` received.

The template source code for metapools may be viewed on GitHub.

3.4 Admin Pool Settings

The following are methods that may only be called by the pool admin (`owner`).

Additionally, some admin methods require a two-phase transaction process, whereby changes are committed in a first transaction and after a forced delay applied via a second transaction. The minimum delay after which a committed action can be applied is given by the constant pool attribute `admin_actions_delay`, which is set to 3 days.

3.4.1 Pool Ownership

`StableSwap.`**`commit_transfer_ownership`**(*_owner: address*)
Initiate an ownership transfer of pool to `_owner`.

Callable only by the ownership admin. The ownership can not be transferred before `transfer_ownership_deadline`, which is the timestamp of the current block delayed by `admin_actions_delay`.

`StableSwap.`**`apply_transfer_ownership`**()
Transfers ownership of the pool from current owner to the owner previously set via `commit_transfer_ownership`.

> **Warning:** Pool ownership can only be transferred once.

`StableSwap.`**`revert_transfer_ownership`**()
Reverts any previously committed transfer of ownership. This method resets the `transfer_ownership_deadline` to `0`.

3.4.2 Amplification Coefficient

The amplification co-efficient ("A") determines a pool's tolerance for imbalance between the assets within it. A higher value means that trades will incur slippage sooner as the assets within the pool become imbalanced.

Note: Within the pools, `A` is in fact implemented as `1 / A` and therefore a higher value implies that the pool will be **more** tolerant to slippage when imbalanced.

The appropriate value for A is dependent upon the type of coin being used within the pool.

It is possible to modify the amplification coefficient for a pool after it has been deployed. However, it requires a vote within the Curve DAO and must reach a 15% quorum.

`StableSwap.`**`ramp_A`**(*_future_A: uint256*, *_future_time: uint256*)
Ramp `A` up or down by setting a new `A` to take effect at a future point in time.

- `_future_A`: New future value of `A`
- `_future_time`: Timestamp at which new `A` should take effect

`StableSwap.`**`stop_ramp_A`**()
Stop ramping `A` up or down and sets `A` to current `A`.

3.4.3 Trade Fees

Curve pools charge fees on token swaps, where the fee may differ between pools. An admin fee is charged on the pool fee. For an overview of how fees are distributed, please refer to *Fee Collection and Distribution*.

`StableSwap.`**`commit_new_fee`**(*_new_fee: uint256*, *_new_admin_fee: uint256*)
Commit new pool and admin fees for the pool. These fees do not take immediate effect.

- `_new_fee`: New pool fee
- `_new_admin_fee`: New admin fee (expressed as a percentage of the pool fee)

Note: Both the pool `fee` and the `admin_fee` are capped by the constants `MAX_FEE` and `MAX_ADMIN_FEE`, respectively. By default `MAX_FEE` is set at 50% and `MAX_ADMIN_FEE` at 100% (which is charged on the `MAX_FEE` amount).

`StableSwap.`**`apply_new_fee`**()
Apply the previously committed new pool and admin fees for the pool.

Note: Unlike ownership transfers, pool and admin fees may be set more than once.

`StableSwap.`**`revert_new_parameters`**()
Resets any previously committed new fees.

`StableSwap.`**`admin_balances`**(*i: uint256*) → uint256
Get the admin balance for a single coin in the pool.

- `i`: Index of the coin to get admin balance for

Returns the admin balance for coin `i`.

`StableSwap.`**`withdraw_admin_fees`**()
Withdraws and transfers admin fees of the pool to the pool owner.

StableSwap.**donate_admin_fees**()
Donate all admin fees to the pool's liquidity providers.

Note: Older Curve pools do not implement this method.

3.4.4 Kill a Pool

StableSwap.**kill_me**()
Pause a pool by setting the `is_killed` boolean flag to `True`.

This disables the following pool functionality: * `add_liquidity` * `exchange` * `remove_liquidity_imbalance` * `remove_liquidity_one_coin`

Hence, when paused, it is only possible for existing LPs to remove liquidity via `remove_liquidity`.

Note: Pools can only be killed within the first 30 days after deployment.

StableSwap.**unkill_me**()
Unpause a pool that was previously paused, re-enabling exchanges.

CHAPTER

FOUR

CURVE STABLESWAP EXCHANGE: LP TOKENS

In exchange for depositing coins into a Curve pool (see *Curve Pools*), liquidity providers receive pool LP tokens. A Curve pool LP token is an ERC20 contract specific to the Curve pool. Hence, LP tokens are transferrable. Holders of pool LP tokens may stake the token into a pool's *liquidity gauge* in order to receive `CRV` token rewards. Alternatively, if the LP token is supported by a metapool, the token may be deposited into the respective metapool in exchange for the metapool's LP token (see *here*).

The following versions of Curve pool LP tokens exist:

- CurveTokenV1: LP token targetting Vyper ^0.1.0-beta.16
- CurveTokenV2: LP token targetting Vyper ^0.2.0
- CurveTokenV3: LP token targetting Vyper ^0.2.0 with gas optimizations

The version of each pool's LP token can be found in the *Deployment Addresses*.

Note: For older Curve pools the `token` attribute is not always `public` and a getter has not been explicitly implemented.

4.1 Curve Token V1

The implementation for a Curve Token V1 may be viewed on GitHub.

`CurveToken.`**`name`**`()` → string[64]: view

Get the name of the token.

```
>>> lp_token.name()
'Curve.fi yDAI/yUSDC/yUSDT/yBUSD'
```

`CurveToken.`**`symbol`**`()` → string[32]: view

Get the token symbol.

```
>>> lp_token.symbol()
'yDAI+yUSDC+yUSDT+yBUSD'
```

`CurveToken.`**`decimals`**`()` → uint256: view

Get the number of decimals for the token.

```
>>> lp_token.decimals()
18
```

CurveToken.**balanceOf**(*account: address*) → uint256: view
Get the token balance for an account.

- account: Address to get the token balance for

```
>>> lp_token.balanceOf("0x69fb7c45726cfe2badee8317005d3f94be838840")
72372801850459006740117197
```

CurveToken.**totalSupply**() → uint256: view
Get the total token supply.

```
>>> lp_token.totalSupply()
73112516629065063732935484
```

CurveToken.**allowance**(*_owner: address*, *_spender: address*) → uint256: view
Get the allowance of an account to spend on behalf of some other account.

- _owner: Account that is paying when _spender spends the allowance
- _spender: Account that can spend up to the allowance

Returns the allowance of _spender for _owner.

CurveToken.**transfer**(*_to: address*, *_value: uint256*) → bool
Transfer tokens to a specified address.

- _to: Receiver of the tokens
- _value: Amount of tokens to transfer

Returns True if the transfer succeeded.

CurveToken.**transferFrom**(*_from: address*, *_to: address*, *_value: uint256*) → bool
Transfer tokens from one address to another. Note that while this function emits a Transfer event, this is not required as per the specification, and other compliant implementations may not emit the event.

- _from: Address which you want to send tokens from
- _to: Address which you want to transfer to
- _value: Amount of tokens to be transferred

Returns True if transfer succeeded.

CurveToken.**approve**(*_spender: address*, *_value: uint256*) → bool
Approve the passed address to spend the specified amount of tokens on behalf of msg.sender.

Beware that changing an allowance with this method brings the risk that someone may use both the old and the new allowance by unfortunate transaction ordering. One possible solution to mitigate this race condition is to first reduce the spender's allowance to 0 and set the desired value afterwards (see this GitHub issue).

- _spender: Address which will spend the funds.
- _value: Amount of tokens to be spent.

Returns True if approval succeeded.

> **Warning:** For Curve LP Tokens V1 and V2, **non-zero to non-zero approvals are prohibited**. Instead, after every non-zero approval, the allowance for the spender **must** be reset to 0.

4.1.1 Minter Methods

The following methods are only callable by the `minter` (private attribute).

Note: For Curve Token V1, the `minter` attribute is not `public`.

`CurveToken.`**`mint`**`(`*_to: address*, *_value: uint256*`)`
Mint an amount of the token and assign it to an account. This encapsulates the modification of balances such that the proper events are emitted.

- `_to`: Address that will receive the created tokens
- `_value`: Amount that will be created

`CurveToken.`**`burn`**`(`*_value: uint256*`)`
Burn an amount of the token of `msg.sender`.

- `_value`: Token amount that will be burned

`CurveToken.`**`burnFrom`**`(`*_to: address*, *_value: uint256*`)`
Burn an amount of the token from a given account.

- `_to`: Account whose tokens will be burned
- `_value`: Amount that will be burned

`CurveToken.`**`set_minter`**`(`*_minter: address*`)`
Set a new minter for the token.

- `_minter`: Address of the new minter

4.2 Curve Token V2

The implementation for a Curve Token V2 may be viewed on GitHub.

Note: Compared to Curve Token v1, the following changes have been made to the API:

- `minter` attribute is `public` and therefore a minter getter has been generated
- `name` and `symbol` attributes can be set via `set_name`
- `mint` method returns `bool`
- `burnFrom` method returns `bool`
- `burn` method has been removed

Warning: For Curve LP Tokens V1 and V2, **non-zero to non-zero approvals are prohibited**. Instead, after every non-zero approval, the allowance for the spender **must** be reset to `0`.

`CurveToken.`**`minter`**`()` → address: view
Getter for the address of the `minter` of the token.

`CurveToken.`**`set_name`**`(`*_name: String[64]*, *_symbol: String[32]*`)`
Set the name and symbol of the token.

- `_name`: New name of token
- `_symbol`: New symbol of token

This method can only be called by `minter`.

`CurveToken.`**`mint`**`(`*_to: address*, *_value: uint256*`)` → bool
Mint an amount of the token and assign it to an account. This encapsulates the modification of balances such that the proper events are emitted.

Returns `True` if not reverted.

`CurveToken.`**`burnFrom`**`(`*_to: address*, *_value: uint256*`)` → bool
Burn an amount of the token from a given account.

- `_to`: Account whose tokens will be burned
- `_value`: Amount that will be burned

Returns `True` if not reverted.

4.3 Curve Token V3

The Curve Token V3 is more gas efficient than versions 1 and 2.

Note: Compared to the Curve Token V2 API, there have been the following changes:

- `increaseAllowance` and `decreaseAllowance` methods added to mitigate race conditions

The implementation for a Curve Token V3 may be viewed on GitHub.

`CurveToken.`**`increaseAllowance`**`(`*_spender: address*, *_added_value: uint256*`)` → bool
Increase the allowance granted to `_spender` by the `msg.sender`.

This is alternative to `approve` that can be used as a mitigation for the potential race condition.

- `_spender`: Address which will transfer the funds
- `_added_value`: Amount of to increase the allowance

Returns `True` if success.

`CurveToken.`**`decreaseAllowance`**`(`*_spender: address*, *_subtracted_value: uint256*`)` → bool
Decrease the allowance granted to `_spender` by the `msg.sender`.

This is alternative to {approve} that can be used as a mitigation for the potential race condition.

- `_spender`: Address which will transfer the funds
- `_subtracted_value`: Amount of to decrease the allowance

Returns `True` if success.

CHAPTER

FIVE

CURVE STABLESWAP EXCHANGE: DEPOSIT CONTRACTS

Curve pools may rely on a different contract, called a *deposit zap* for the addition and removal of underlying coins. This is particularly useful for lending pools, which may only support the addition/removal of wrapped coins. Furthermore, deposit zaps are also useful for metapools, which do not support the addition/removal of base pool coins.

5.1 Lending Pool Deposits

While Curve lending pools support swaps in both the wrapped *and* underlying coins, not all lending pools allow liquidity providers to deposit or withdraw the underlying coin.

For example, the Compound Pool allows swaps between `cDai` and `cUSDC` (wrapped coins), as well as swaps between `DAI` and `USDC` (underlying coins). However, liquidity providers are not able to deposit `DAI` or `USDC` to the pool directly. The main reason for why this is not supported by all Curve lending pools lies in the size limit of contracts. Lending pools may differ in complexity and can end up being very close to the contract byte code size limit. In order to overcome this restriction, liquidity can be added and removed to and from a lending pool in the underlying coins via a different contract, called a *deposit zap*, tailored to lending pools.

For an overview of the Curve lending pool implementation, please refer to the *Lending Pool* section.

The template source code for a lending pool deposit zap may be viewed on GitHub.

Note: Lending pool deposit zaps may differ in their API. Older pools do not implement the newer API template.

5.1.1 Deposit Zap API (OLD)

Older Curve lending pool deposit zaps do not implement the template API. The deposit zaps which employ an older API are:

- `DepositBUSD`: BUSD pool deposit zap
- `DepositCompound`: Compound pool deposit zap
- `DepositPAX`: PAX pool deposit zap
- `DepositUSDT`: USDT pool deposit zap
- `DepositY`: Y pool deposit zap

While not a lending pool, note that the following contract also implements the newer deposit zap API:

- `DepositSUSD`: SUSD pool deposit zap

Get Deposit Zap Information

Note: Getters generated for public arrays changed between Vyper `0.1.x` and `0.2.x` to accept `uint256` instead of `int128` in order to handle the lookups. Older deposit zap contracts (v1) use `vyper 0.1.x...`, while newer zaps (v2) use `vyper 0.2.x...`.

The following Brownie console interaction examples are using the Compound Pool Deposit Zap.

`DepositZap.`**`curve`**`()` → address: view
Getter for the pool associated with this deposit contract.

```
>>> zap.curve()
'0xA2B47E3D5c44877cca798226B7B8118F9BFb7A56'
```

`DepositZap.`**`underlying_coins`**`(`*i: int128*`)` → address: view
Getter for the array of **underlying** coins within the associated pool.

- `i`: Index of the underlying coin for which to get the address

```
>>> zap.underlying_coins(0)
'0x6B175474E89094C44Da98b954EedeAC495271d0F'
>>> zap.underlying_coins(1)
'0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48'
```

`DepositZap.`**`coins`**`(`*i: int128*`)` → address: view
Getter for the array of **wrapped** coins within the associated pool.

- `i`: Index of the coin for which to get the address

```
>>> zap.coins(0)
'0x5d3a536E4D6DbD6114cc1Ead35777bAB948E3643'
>>> zap.coins(1)
'0x39AA39c021dfbaE8faC545936693aC917d5E7563'
```

`DepositZap.`**`token`**`()` → address: view
Getter for the LP token of the associated pool.

```
>>> zap.token()
'0x845838DF265Dcd2c412A1Dc9e959c7d08537f8a2'
```

Adding/Removing Liquidity

`DepositZap.`**`add_liquidity`**`(`*uamounts: uint256[N_COINS]*, *min_mint_amount: uint256*`)`
Wrap underlying coins and deposit them in the pool

- `uamounts`: List of amounts of underlying coins to deposit
- `min_mint_amount`: Minimum amount of LP token to mint from the deposit

`DepositZap.`**`remove_liquidity`**`(`*_amount: uint256*, *min_uamounts: uint256[N_COINS]*`)`
Withdraw and unwrap coins from the pool.

- `_amount`: Quantity of LP tokens to burn in the withdrawal
- `min_uamounts`: Minimum amounts of underlying coins to receive

DepositZap.**remove_liquidity_imbalance**(*uamounts: uint256[N_COINS], max_burn_amount: uint256*)
Withdraw and unwrap coins from the pool in an imbalanced amount.

- `uamounts`: List of amounts of underlying coins to withdraw
- `max_burn_amount`: Maximum amount of LP token to burn in the withdrawal

DepositZap.**remove_liquidity_one_coin**(*_token_amount: uint256, i: int128, min_uamount: uint256, donate_dust: bool = False*)
Withdraw and unwrap a single coin from the pool

- `_token_amount`: Amount of LP tokens to burn in the withdrawal
- `i`: Index value of the coin to withdraw
- `min_uamount`: Minimum amount of underlying coin to receive

DepositZap.**calc_withdraw_one_coin**(*_token_amount: uint256, i: int128*) → uint256
Calculate the amount received when withdrawing a single underlying coin.

- `_token_amount`: Amount of LP tokens to burn in the withdrawal
- `i`: Index value of the coin to withdraw

DepositZap.**withdraw_donated_dust**()
Donates any LP tokens of the associated pool held by this contract to the contract owner.

5.1.2 Deposit Zap API (NEW)

Compared to the older deposit zaps, the newer zaps mainly optimize for gas efficiency. The API is only modified in part, specifically with regards to `return` values and variable naming.

Get Deposit Zap Information

DepositZap.**curve**() → address: view
Getter for the pool associated with this deposit contract.

DepositZap.**underlying_coins**(*i: uint256*) → address: view
Getter for the array of **underlying** coins within the associated pool.

- `i`: Index of the underlying coin for which to get the address

DepositZap.**coins**(*i: uint256*) → address: view
Getter for the array of **wrapped** coins within the associated pool.

- `i`: Index of the coin for which to get the address

DepositZap.**lp_token**() → address: view
Getter for the LP token of the associated pool.

Adding/Removing Liquidity

DepositZap.**add_liquidity**(*_underlying_amounts: uint256[N_COINS], _min_mint_amount: uint256*) → uint256

Wrap underlying coins and deposit them in the pool

- `_underlying_amounts`: List of amounts of underlying coins to deposit
- `_min_mint_amount`: Minimum amount of LP tokens to mint from the deposit

Returns the amount of LP token received in exchange for the deposited amounts.

DepositZap.**remove_liquidity**(*_amount: uint256, _min_underlying_amounts: uint256[N_COINS]*) → uint256[N_COINS]

Withdraw and unwrap coins from the pool.

- `_amount`: Quantity of LP tokens to burn in the withdrawal
- `_min_underlying_amounts`: Minimum amounts of underlying coins to receive

Returns list of amounts of underlying coins that were withdrawn.

DepositZap.**remove_liquidity_imbalance**(*_underlying_amounts: uint256[N_COINS], _max_burn_amount: uint256*) → uint256[N_COINS]

Withdraw and unwrap coins from the pool in an imbalanced amount. Amounts in *_underlying_amounts* correspond to withdrawn amounts before any fees charge for unwrapping.

- `_underlying_amounts`: List of amounts of underlying coins to withdraw
- `_max_burn_amount`: Maximum amount of LP token to burn in the withdrawal

Returns list of amounts of underlying coins that were withdrawn.

DepositZap.**remove_liquidity_one_coin**(*_amount: uint256, i: int128, _min_underlying_amount: uint256*) → uint256

Withdraw and unwrap a single coin from the pool

- `_amount`: Amount of LP tokens to burn in the withdrawal
- `i`: Index value of the coin to withdraw
- `_min_underlying_amount`: Minimum amount of underlying coin to receive

Returns amount of underlying coin received.

5.2 Metapool Deposits

While Curve metapools support swaps between base pool coins, the base pool LP token and metapool coins, they do not allow liquidity providers to deposit and/or withdraw base pool coins.

For example, the GUSD metapool is a pool consisting of `GUSD` and `3CRV` (the LP token of the 3Pool) and allows for swaps between `GUSD`, `DAI`, `USDC`, `USDT` and `3CRV`. However, liquidity providers are not able to deposit `DAI`, `USDC` or `USDT` to the pool directly. The main reason why this is not possible lies in the maximum byte code size of contracts. Metapools are complex and can therefore end up being very close to the contract byte code size limit. In order to overcome this restriction, liquidity can be added and removed to and from a metapool in the base pool's coins through a metapool deposit zap.

For an overview of the Curve metapool implementation, please refer to the *Metapool* section.

The template source code for a metapool deposit "zap" may be viewed on GitHub.

A list of all deployed metapool deposit zaps can be found *here*.

Note: Metapool deposit zaps contain the following private and hardcoded constants:

- `N_COINS`: Number of coins in the metapool (excluding base pool coins)
- `BASE_N_COINS`: Number of coins in the base pool
- `N_ALL_COINS`: All coins in the metapool, excluding the base pool LP token (`N_COINS + BASE_N_COINS - 1`)

5.2.1 Get Deposit Zap Information

`DepositZap.`**`pool`**`()` → address: view
Getter for the metapool associated with this deposit contract.

`DepositZap.`**`base_pool`**`()` → address: view
Getter for the base pool of the metapool associated with this deposit contract.

`DepositZap.`**`base_coins`**`(`*i: uint256*`)` → address: view
Getter for the array of the coins of the metapool's base pool.

- `i`: Index of the underlying coin for which to get the address

`DepositZap.`**`coins`**`(`*i: uint256*`)` → address: view
Getter for the array of metapool's coins.

- `i`: Index of the coin for which to get the address

`DepositZap.`**`token`**`()` → address: view
Getter for the LP token of the associated metapool.

5.2.2 Adding/Removing Liquidity

Note: For methods taking the index argument `i`, a number in the range from `0` to `N_ALL_COINS - 1` is valid. This refers to all coins apart from the base pool LP token.

`DepositZap.`**`add_liquidity`**`(`*_amounts: uint256[N_ALL_COINS]*, *_min_mint_amount: uint256*`)` → uint256
Wrap underlying coins and deposit them in the pool.

- `_amounts`: List of amounts of underlying coins to deposit
- `_min_mint_amount`: Minimum amount of LP tokens to mint from the deposit

Returns the amount of LP token received in exchange for depositing.

`DepositZap.`**`remove_liquidity`**`(`*_amount: uint256*, *_min_amounts: uint256[N_ALL_COINS]*`)` → uint256[N_ALL_COINS]
Withdraw and unwrap coins from the pool.

- `_amount`: Quantity of LP tokens to burn in the withdrawal
- `_min_amounts`: Minimum amounts of underlying coins to receive

Returns a list of amounts of underlying coins that were withdrawn.

`DepositZap.`**`remove_liquidity_one_coin`**`(`*_token_amount: uint256*, *i: int128*, *_min_amount: uint256*`)` → uint256
Withdraw and unwrap a single coin from the metapool.

- `_token_amount`: Amount of LP tokens to burn in the withdrawal
- `i`: Index value of the coin to withdraw
- `_min_amount`: Minimum amount of underlying coin to receive

Returns the amount of the underlying coin received.

`DepositZap.`**`remove_liquidity_imbalance`**(*_amounts: uint256[N_ALL_COINS]*, *_max_burn_amount: uint256*) → uint256

Withdraw coins from the pool in an imbalanced amount

- `_amounts`: List of amounts of underlying coins to withdraw
- `_max_burn_amount`: Maximum amount of LP token to burn in the withdrawal

Returns the actual amount of the LP token burned in the withdrawal.

`DepositZap.`**`calc_withdraw_one_coin`**(*_token_amount: uint256*, *i: int128*) → uint256

Calculate the amount received when withdrawing and unwrapping a single coin

- `_token_amount`: Amount of LP tokens to burn in the withdrawal
- `i`: Index value of the coin to withdraw (`i` should be in the range from `0` to `N_ALL_COINS - 1`, where the LP token of the base pool is removed).

Returns the amount of coin `i` received.

`DepositZap.`**`calc_token_amount`**(*_amounts: uint256[N_ALL_COINS]*, *_is_deposit: bool*) → uint256

Calculate addition or reduction in token supply from a deposit or withdrawal.

- `_amounts`: Amount of each underlying coin being deposited
- `_is_deposit`: Set True for deposits, False for withdrawals

Returns the expected amount of LP tokens received.

CHAPTER

SIX

CROSS ASSET SWAPS

Curve integrates with Synthetix to allow large scale swaps between different asset classes with minimal slippage. Utilizing Synthetix' zero-slippage synth conversions and Curve's deep liquidity and low fees, we can perform fully on-chain cross asset swaps at scale with a 0.38% fee and minimal slippage.

Cross asset swaps are performed using the `SynthSwap` contract, deployed to the mainnet at the following address:

0x58A3c68e2D3aAf316239c003779F71aCb870Ee47

Source code and information on the technical implementation are available on Github.

6.1 How it Works

As an example, suppose we have asset `A` and wish to exchange it for asset `D`. For this swap to be possible, `A` and `D` must meet the following requirements:

- Must be of different asset classes (e.g. USD, EUR, BTC, ETH)
- Must be exchangeable for a Synthetic asset within one of Curve's pools (e.g. sUSD, sBTC)

The swap can be visualized as `A -> B -> C | C -> D`:

1. The initial asset `A` is exchanged on Curve for `B`, a synth of the same asset class.
2. `B` is converted to `C`, a synth of the same asset class as `D`.
3. A settlement period passes to account for sudden price movements between `B` and `C`.
4. Once the settlement period has passed, `C` is exchanged on Curve for the desired asset `D`.

6.1.1 Settler NFTs

Swaps cannot occur atomically due to the Synthetix settlement period. Each unsettled swap is represented by an ERC721 non-fungible token.

- Each NFT has a unique token ID. Token IDs are never re-used. The NFT is minted upon initiating the swap and burned when the swap is completed.
- The NFT, and associated right to claim, is fully transferable. It is not possible to transfer the rights to a partial claim. The approved operator for an NFT also has the right to complete the swap with the underlying asset.
- Token IDs are not sequential. This contract does not support the enumerable ERC721 extension. This decision is based on gas efficiency.

6.1.2 Front-running Considerations

The benefits from these swaps are most apparent when the exchange amount is greater than $1m USD equivalent. As such, the initiation of a swap gives a strong indicator other market participants that a 2nd post-settlement swap will be coming. We attempt to minimize the risks from this in several ways:

- `C -> D` is not declared on-chain when performing the swap from `A -> C`.
- It is possible to perform a partial swap from `C -> D`, and to swap into multiple final assets. The NFT persists until it has no remaining underlying balance of `C`.
- There is no fixed time frame for the second swap. A user can perform it immediately or wait until market conditions are more favorable.
- It is possible to withdraw `C` without performing a second swap.
- It is possible to perform additional `A -> B -> C` swaps to increase the balance of an already existing NFT.

The range of available actions and time frames make it significantly more difficult to predict the outcome of a swap and trade against it.

6.2 Exchange API

6.2.1 Finding Swappable Assets

In general, any asset that is within a Curve pool also containing a Synth may be used in a cross asset swap. You can use the following view methods to confirm whether or not an asset is supported:

`StableSwap.`**`synth_pools`**`(`*_synth: address*`)` → address: view
Get the address of the Curve pool used to swap a synthetic asset.

If this function returns `ZERO_ADDRESS`, the given synth cannot be used within cross-asset swaps.

`StableSwap.`**`swappable_synth`**`(`*_token: address*`)` → address: view
Get the address of the synthetic asset that `_token` may be directly swapped for.

If this function returns `ZERO_ADDRESS`, the given token cannot be used within a cross-asset swap.

```
>>> synth_swap = Contract('0x58A3c68e2D3aAf316239c003779F71aCb870Ee47')
>>> dai = Contract('0x6b175474e89094c44da98b954eedeac495271d0f')

>>> synth_swap.swappable_synth(dai)  # returns sUSD
'0x57Ab1ec28D129707052df4dF418D58a2D46d5f51'

>>> synth_swap.synth_pools('0x57ab1ec28d129707052df4df418d58a2d46d5f51')
↪# returns Curve sUSD pool
'0xA5407eAE9Ba41422680e2e00537571bcC53efBfD'
```

6.2.2 Estimating Swap Amounts

StableSwap.**get_swap_into_synth_amount**(*_from: address*, *_synth: address*, *_amount: uint256*) → uint256: view

Return the amount received when performing a cross-asset swap.

This method is used to calculate `_expected` when calling *swap_into_synth*. You should reduce the value slightly to account for market movement prior to the transaction confirming.

- `_from`: Address of the initial asset being exchanged.
- `_synth`: Address of the synth being swapped into.
- `_amount`: Amount of *_from* to swap.

Returns the expected amount of `_synth` received in the swap.

```
>>> synth_swap = Contract('0x58A3c68e2D3aAf316239c003779F71aCb870Ee47')
>>> dai = Contract('0x6b175474e89094c44da98b954eedeac495271d0f')
>>> sbtc = Contract('0xfe18be6b3bd88a2d2a7f928d00292e7a9963cfc6')

>>> synthswap.get_swap_into_synth_amount(dai, sbtc, 100000 * 1e18)
2720559215249173192
```

StableSwap.**get_swap_from_synth_amount**(*_synth: address*, *_to: address*, *_amount: uint256*) → uint256: view

Return the amount received when swapping out of a settled synth.

This method is used to calculate `_expected` when calling *swap_from_synth*. You should reduce the value slightly to account for market movement prior to the transaction confirming.

- `_synth`: Address of the synth being swapped out of.
- `_to`: Address of the asset to swap into.
- `_amount`: Amount of `_synth` being exchanged.

Returns the expected amount of *_to* received in the swap.

```
>>> synth_swap = Contract('0x58A3c68e2D3aAf316239c003779F71aCb870Ee47')
>>> sbtc = Contract('0xfe18be6b3bd88a2d2a7f928d00292e7a9963cfc6')
>>> wbtc = Contract('0x2260fac5e5542a773aa44fbcfedf7c193bc2c599')

>>> synthswap.get_swap_from_synth_amount(sbtc, wbtc, 2720559215249173192)
273663013
```

StableSwap.**get_estimated_swap_amount**(*_from: address*, *_to: address*, *_amount: uint256*) → uint256: view

Estimate the final amount received when swapping between `_from` and `_to`.

Note that the actual received amount may be different due to rate changes during the settlement period.

- `_from`: Address of the initial asset being exchanged.
- `_to`: Address of the asset to swap into.
- `_amount`: Amount of *_from* being exchanged.

Returns the estimated amount of *_to* received.

```
>>> synth_swap = Contract('0x58A3c68e2D3aAf316239c003779F71aCb870Ee47')
>>> dai = Contract('0x6b175474e89094c44da98b954eedeac495271d0f')
>>> wbtc = Contract('0x2260fac5e5542a773aa44fbcfedf7c193bc2c599')
```

(continues on next page)

(continued from previous page)

```
>>> synthswap.get_estimated_swap_amount(dai, wbtc, 100000 * 1e18)
273663013
```

Note: This method is for estimating the received amount from a complete swap over two transactions. If `_to` is a Synth, you should use *`get_swap_into_synth_amount`* instead.

6.2.3 Initiating a Swap

All cross asset swaps are initiated with the following method:

`StableSwap.`**`swap_into_synth`**(*_from: address*, *_synth: address*, *_amount: uint256*, *_expected: uint256*, *_receiver: address = msg.sender*, *_existing_token_id: uint256 = 0*) → uint256: payable

Perform a cross-asset swap between `_from` and `_synth`.

Synth swaps require a settlement time to complete and so the newly generated synth cannot immediately be transferred onward. Calling this function mints an NFT representing ownership of the unsettled synth.

- `_from`: Address of the initial asset being exchanged. For Ether swaps, use `0xEeeeeEeeeEeEeeEeEeEeeeEEEeeeeEeeeeeeeeEEeE`.
- `_synth`: Address of the synth being swapped into.
- `_amount`: Amount of `_from` to swap. If you are swapping from Ether, you must also send exactly this much Ether with the transaction. If you are swapping any other asset, you must have given approval to the swap contract to transfer at least this amount.
- `_expected`: Minimum amount of `_synth` to receive.
- `_receiver`: Address of the recipient of `_synth`, if not given, defaults to the caller.
- `_existing_token_id`: Token ID to deposit `_synth` into. If not given, a new NFT is minted for the generated synth. When set as non-zero, the token ID must be owned by the caller and must already represent the same synth as is being swapped into.

Returns the `uint256` token ID of the NFT representing the unsettled swap. The token ID is also available from the emitted `TokenUpdate` event.

```
>>> alice = accounts[0]

>>> synth_swap = Contract('0x58A3c68e2D3aAf316239c003779F71aCb870Ee47')
>>> dai = Contract('0x6b175474e89094c44da98b954eedeac495271d0f')
>>> sbtc = Contract('0xfe18be6b3bd88a2d2a7f928d00292e7a9963cfc6')

>>> expected = synth_swap.get_swap_into_synth_amount(dai, sbtc, dai.
↪balanceOf(alice)) * 0.99

>>> tx = synth_swap.swap_into_synth(dai, sbtc, expected, {'from': alice})
Transaction sent:␣
↪0x83b311af19be08b8ec6241c3e834ccdf3b22586971de82a76a641e43bdf2b3ee
  Gas price: 20 gwei   Gas limit: 1200000   Nonce: 5

>>> tx.events['TokenUpdate']['token_id']
242399470789520938623986522716345106047390461906 5
```

6.2.4 Getting Info about an Unsettled Swap

StableSwap.**token_info**(*_token_id: uint256*) → address, address, uint256, uint256: view
Get information about the underlying synth represented by an NFT.

- _token_id: NFT token ID to query info about. Reverts if the token ID does not exist.

Returns the owner of the NFT, the address of the underlying synth, the balance of the underlying synth, and the current maximum number of seconds until the synth may be settled.

```
>>> synth_swap = Contract('0x58A3c68e2D3aAf316239c003779F71aCb870Ee47')
>>> synthswap.token_
↪info(24239947078952093862398652271634510604739046190 65).dict()
{
    'owner': "0xEF422dBBF46120dE627fFb913C9AFaD44c735618",
    'synth': "0x57Ab1ec28D129707052df4dF418D58a2D46d5f51",
    'time_to_settle': 0,
    'underlying_balance': 1155647333395694644849
}
```

6.2.5 Completing a Swap

Once the settlement period on a swap has finished, any of the following methods may be used to complete the swap.

StableSwap.**swap_from_synth**(*_token_id: uint256*, *_to: address*, *_amount: uint256*, *_expected: uint256*, *_receiver: address = msg.sender*) → uint256: nonpayable
Swap the underlying synth represented by an NFT into another asset.

Callable by the owner or operator of _token_id after the synth settlement period has passed. If _amount is equal to the total remaining balance of the synth represented by the NFT, the NFT is burned.

- _token_id: The identifier for an NFT.
- _to: Address of the asset to swap into.
- _amount: Amount of the underlying synth to swap.
- _expected: Minimum amount of _to to receive.
- _receiver: Address to send the final received asset to. If not given, defaults to the caller.

Returns the remaining balance of the underlying synth within the active NFT.

```
>>> wbtc = Contract('0x2260fac5e5542a773aa44fbcfedf7c193bc2c599')

>>> amount = synth_swap.token_info(token_id)['underlying_balance']
>>> expected = swynth_swap.get_swap_from_synth_amount(sbtc, wbtc, amount)␣
↪* 0.99

>>> synth_swap.swap_from_synth(token_id, wbtc, amount, expected, {'from':␣
↪alice})
Transaction sent:␣
↪0x83b311af19be08b8ec6241c3e834ccdf3b22586971de82a76a641e43bdf2b3ee
  Gas price: 20 gwei   Gas limit: 800000   Nonce: 6
```

StableSwap.**withdraw**(*_token_id: uint256*, *_amount: uint256*, *_receiver: address = msg.sender*) → uint256: nonpayable
Withdraw the underlying synth represented by an NFT.

Callable by the owner or operator of _token_id after the synth settlement period has passed. If _amount is equal to the total remaining balance of the synth represented by the NFT, the NFT is burned.

- `_token_id`: The identifier for an NFT.
- `_amount`: Amount of the underlying synth to withdraw.
- `_receiver`: Address of the recipient of the withdrawn synth. If not given, defaults to the caller.

Returns the remaining balance of the underlying synth within the active NFT.

```
>>> amount = synth_swap.token_info(token_id)['underlying_balance']

>>> synth_swap.withdraw(token_id, amount, {'from': alice})
Transaction sent:␣
↪0x83b311af19be08b8ec6241c3e834ccdf3b22586971de82a76a641e43bdf2b3ee
  Gas price: 20 gwei   Gas limit: 800000   Nonce: 6
```

`StableSwap.`**`settle`**`(`*_token_id: uint256*`)` → bool: nonpayable

Settle the synth represented in an NFT. Note that settlement is performed when swapping or withdrawing, there is no requirement to call this function separately.

- `_token_id` The identifier for an NFT.

Returns `True`.

CHAPTER

SEVEN

THE CURVE DAO

Curve DAO consists of multiple smart contracts connected by Aragon. Interaction with Aragon occurs through a modified implementation of the Aragon Voting App. Aragon's standard one token, one vote method is replaced with a weighting system based on locking tokens.

Curve DAO has a token (CRV) which is used for both governance and value accrual.

Source code for the Curve DAO can be found on Github.

CHAPTER

EIGHT

CURVE DAO: VOTE-ESCROWED CRV

Participating in Curve DAO governance requires that an account have a balance of vote-escrowed CRV (veCRV). veCRV is a non-standard ERC20 implementation, used within the Aragon DAO to determine each account's voting power.

veCRV is represented by the `VotingEscrow` contract, deployed to the Ethereum mainnet at:

0x5f3b5DfEb7B28CDbD7FAba78963EE202a494e2A2

veCRV cannot be transferred. The only way to obtain veCRV is by locking CRV. The maximum lock time is four years. One CRV locked for four years provides an initial balance of one veCRV.

A user's veCRV balance decays linearly as the remaining time until the CRV unlock decreases. For example, a balance of 4000 CRV locked for one year provides the same amount of veCRV as 2000 CRV locked for two years, or 1000 CRV locked for four years.

8.1 Implementation Details

User voting power w_i is linearly decreasing since the moment of lock. So does the total voting power W. In order to avoid periodic check-ins, every time the user deposits, or withdraws, or changes the locktime, we record user's slope and bias for the linear function $w_i(t)$ in the public mapping `user_point_history`. We also change slope and bias for the total voting power $W(t)$ and record it in `point_history`. In addition, when a user's lock is scheduled to end, we schedule change of slopes of $W(t)$ in the future in `slope_changes`. Every change involves increasing the `epoch` by 1.

This way we don't have to iterate over all users to figure out, how much should $W(t)$ change by, neither we require users to check in periodically. However, we limit the end of user locks to times rounded off by whole weeks.

Slopes and biases change both when a user deposits and locks governance tokens, and when the locktime expires. All the possible expiration times are rounded to whole weeks to make number of reads from blockchain proportional to number of missed weeks at most, not number of users (which is potentially large).

8.2 Querying Balances, Locks and Supply

`VotingEscrow.`**`balanceOf`**(*addr: address*, *_t: uint256 = block.timestamp*) → uint256

Get the current voting power for an address.

- `addr`: User wallet address

```
>>> vote_escrow = Contract('0x5f3b5DfEb7B28CDbD7FAba78963EE202a494e2A2
↪')
```

(continues on next page)

(continued from previous page)

```
>>> vote_escrow.balanceOf('0xF89501B77b2FA6329F94F5A05FE84cEbb5c8b1a0')
546419132938914450333356 4
```

VotingEscrow.**balanceOfAt**(*addr: address*, *_block: uint256*) → uint256

Measure the voting power of an address at a historic block height.

This function is taken from the MiniMe ERC20 implementation and is required for compatibility with Aragon.

- addr: User wallet address
- _block: Block to calculate the voting power at

```
>>> height = len(chain) - 10000  # ten thousand blocks prior to the␣
↪current block
>>> vote_escrow.balanceOfAt('0xF89501B77b2FA6329F94F5A05FE84cEbb5c8b1a0
↪', height)
54701883110176983106287 52
```

VotingEscrow.**totalSupply**() → uint256

Calculate the current total voting power.

```
>>> vote_escrow.totalSupply()
102535077684041114817306735
```

VotingEscrow.**totalSupplyAt**(*_block: uint256*) → uint256

Calculate the total voting power at a historic block height.

- _block Block to calculate the total voting power at.

```
>>> height = len(chain) - 10000  # ten thousand blocks prior to the␣
↪current block
>>> vote_escrow.totalSupplyAt(height)
101809514082846807874928588
```

VotingEscrow.**locked**(*_user: address*)

Get information about the current CRV lock for an address.

- _user: Address to query.

Returns amount of CRV currently locked, and the epoch time that the lock expires.

```
>>> vote_escrow.locked('0xF89501B77b2FA6329F94F5A05FE84cEbb5c8b1a0').
↪dict()
{
    'amount': 56647166122693923976337 36,
    'end': 1736985600
}
```

8.3 Working with Vote-Locks

VotingEscrow.**create_lock**(*_value: uint256*, *_unlock_time: uint256*)

Deposit CRV into the contract and create a new lock.

Prior to calling this function, the contract must be approved to transfer at least `_value` CRV. A new lock cannot be created when an existing lock already exists.

- `_value`: The amount of CRV to deposit.
- `_unlock_time` Epoch time when tokens unlock. This value is rounded down to the nearest whole week. The maximum duration for a lock is four years.

```
>>> import time
>>> crv = Contract('0xd533a949740bb3306d119cc777fa900ba034cd52')
>>> vote_escrow = Contract('0x5f3b5DfEb7B28CDbD7FAba78963EE202a494e2A2
↪')

>>> crv.approve(vote_escrow, 2**256-1, {'from': alice})
Transaction sent:␣
↪0xa7978a8d7fc185d9194bd3c2fa1801ccc57ad4edcfcaff7b5dab1c9101b78cf9
  Gas price: 20.0 gwei   Gas limit: 56299   Nonce: 23


>>> amount = crv.balanceOf(alice)
>>> unlock_time = int(time.time() + 86400 * 365 * 4)
>>> vote_escrow.create_lock(amount, unlock_time, {'from': alice})
Transaction sent:␣
↪0xa7978a8d7fc185d9194bd3c2fa1801ccc57ad4edcfcaff283958329291b78cf1
  Gas price: 20.0 gwei   Gas limit: 307234   Nonce: 24
```

VotingEscrow.**increase_amount**(*_value: uint256*)

Deposit additional CRV into an existing lock.

- `_value`: The amount of CRV to deposit.

```
>>> amount = crv.balanceOf(alice)
>>> vote_escrow.increase_amount(amount, {'from': alice})
Transaction sent:␣
↪0xa7978a8d7fc185d9194bd3c2fa1801ccc57ad4edcfcaff7b5dab1c9101b78cf9
  Gas price: 20.0 gwei   Gas limit: 156299   Nonce: 24
```

VotingEscrow.**increase_unlock_time**(*_unlock_time: uint256*)

Extend the unlock time on a lock that already exists.

- `_unlock_time` New epoch time for unlocking. This value is rounded down to the nearest whole week. The maximum duration for a lock is four years.

```
>>> unlock_time = int(time.time() + 86400 * 365 * 4)
>>> vote_escrow.increase_unlock_time(unlock_time, {'from': alice})
Transaction sent:␣
↪0xa7978a8d7fc185d9194bd3c2fa1801ccc57ad4edcfcaff7b5dab1c9101b78cf9
  Gas price: 20.0 gwei   Gas limit: 282041   Nonce: 24
```

VotingEscrow.**withdraw**()

Withdraw deposited CRV tokens once a lock has expired.

```
>>> vote_escrow.withdraw({'from': alice})
Transaction sent:␣
↪0xa7978a8d7fc185d9194bd3c2fa1801ccc57ad4edcfcaff7b5dab1c9101b78cf9
  Gas price: 20.0 gwei   Gas limit: 178629   Nonce: 24
```

CHAPTER

NINE

THE CURVE DAO: LIQUIDITY GAUGES AND MINTING CRV

Curve incentivizes liquidity providers with the CRV, the protocol governance token. Allocation, distribution and minting of CRV are managed via several related DAO contracts:

- `LiquidityGauge`: Measures liquidity provided by users over time, in order to distribute CRV and other rewards
- `GaugeController`: Central controller that maintains a list of gauges, weights and type weights, and coordinates the rate of CRV production for each gauge
- `Minter`: CRV minting contract, generates new CRV according to liquidity gauges

9.1 Implementation Details

9.1.1 CRV Inflation

CRV follows a piecewise linear inflation schedule. The inflation is reduced by $2^{1/4}$ each year. Each time the inflation reduces, a new mining epoch starts.

The initial supply of CRV is 1.273 billion tokens, which is 42% of the eventual $t \to \infty$" supply of ≈ 3.03 billion tokens. All of these initial tokens are gradually vested (with every block). The initial inflation rate which supports the above inflation schedule is $r = 22.0\%$ (279.6 millions per year). All of the inflation is distributed to Curve liquidity providers, according to measurements taken by the gauges. During the first year, the approximate inflow into circulating supply is 2 million CRV per day. The initial circulating supply is 0.

9.1.2 Liquidity Gauges

Inflation is directed to users who provide liquidity within the protocol. This usage is measured via "Liquidity Gauge" contracts. Each pool has an individual liquidity gauge. The *Gauge Controller* maintains a list of gauges and their types, with the weights of each gauge and type.

To measure liquidity over time, the user deposits their LP tokens into the liquidity gauge. Coin rates which the gauge is getting depends on current inflation rate, gauge weight, and gauge type weights. Each user receives a share of newly minted CRV proportional to the amount of LP tokens locked. Additionally, rewards may be boosted by up to factor of 2.5 if the user vote-locks tokens for Curve governance in the *Voting Escrow* contract.

Suppose we have the inflation rate r changing with every epoch (1 year), gauge weight w_g and gauge type weight w_t. Then, all the gauge handles the stream of inflation with the rate $r' = w_g w_t r$ which it can update every time w_g, w_t, or mining epoch changes.

To calculate a user's share of r', we must calculate the integral: $\mathrm{I}_u = \int \frac{r'(t)\, b_u(t)}{S(t)}\, dt$, where $b_u(t)$ is the balance supplied by the user (measured in LP tokens) and $S(t)$ is total liquidity supplied by users, depending on the time t; the value I_u gives the amount of tokens which the user has to have minted to them. The user's balance b_u changes every time the user u makes a deposit or withdrawal, and S changes every time _any_ user makes a deposit or withdrawal (so S can change many times in between two events for the user u". In the liquidity gauge contract, the vaule of I_u is recorded per-user in the public `integrate_fraction` mapping.

To avoid requiring that all users to checkpoint periodically, we keep recording values of the following integral (named `integrate_inv_supply` in the contract):

$$\mathrm{I}_{is}(t) = \int_0^t \frac{r'(t)}{S(t)} dt.$$

The value of I_{is} is recorded at any point any user deposits or withdraws, as well as every time the rate r' changes (either due to weight change or change of mining epoch).

When a user deposits or withdraws, the change in I_u can be calculated as the current (before user's action) value of I_{is} multiplied by the pre-action user's balance, and sumed up across the user's balances: $\mathrm{I}_u(t_k) = \sum_k b_u(t_k) \left[I_{is}(t_k) - I_{is}(t_{k-1}) \right]$. The per-user integral is possible to repalce with this sum because $b_u(t)$ changed for all times between t_{k-1} and t_k.

9.1.3 Boosting

In order to incentivize users to participate in governance, and additionally create stickiness for liquidity, we implement the following mechanism. A user's balance, counted in the liquidity gauge, gets boosted by users locking CRV tokens in *Voting Escrow* contract, depending on their vote weight w_i: $b_u^* = \min\left(0.4\, b_u + 0.6\, S\frac{w_i}{W},\, b_u\right)$. The value of w_i is taken at the time the user performs any action (deposit, withdrawal, withdrawal of minted CRV tokens) and is applied until the next action this user performs.

If no users vote-lock any CRV (or simply don't have any), the inflation will simply be distributed proportionally to the liquidity b_u each one of them provided. However, if a user stakes enough CRV, they are able to boost their stream of CRV by up to factor of 2.5 (reducing it slightly for all users who are not doing that).

Implementation details are such that a user gets the boost at the time of the last action or checkpoint. Since the voting power decreases with time, it is favorable for users to apply a boost and do no further actions until they vote-lock more tokens. However, once the vote-lock expires, everyone can "kick" the user by creating a checkpoint for that user and, essentially, resetting the user to no boost if they have no voting power at that point already.

Finally, the gauge is supposed to not miss a full year of inflation (e.g. if there were no interactions with the guage for the full year). If that ever happens, the abandoned gauge gets less CRV.

9.1.4 Gauge Weight Voting

Users can allocate their veCRV towards one or more liquidity gauges. Gauges receive a fraction of newly minted CRV tokens proportional to how much veCRV the gauge is allocated. Each user with a veCRV balance can change their preference at any time.

When a user applies a new weight vote, it gets applied at the start of the next epoch week. The weight vote for any one gauge cannot be changed more often than once in 10 days.

9.1.5 The Gauge Controller

The "Gauge Controller" maintains a list of gauges and their types, with the weights of each gauge and type. In order to implement weight voting, `GaugeController` has to include parameters handling linear character of voting power each user has.

`GaugeController` records points (bias + slope) per gauge in `vote_points`, and _scheduled_ changes in biases and slopes for those points in `vote_bias_changes` and `vote_slope_changes`. New changes are applied at the start of each epoch week.

Per-user, per-gauge slopes are stored in `vote_user_slopes`, along with the power the user has used and the time their vote-lock ends.

The totals for slopes and biases for vote weight per gauge, and sums of those per type, are scheduled / recorded for the next week, as well as the points when voting power gets to 0 at lock expiration for some of users.

When a user changes their gauge weight vote, the change is scheduled for the next epoch week, not immediately. This reduces the number of reads from storage which must to be performed by each user: it is proportional to the number of weeks since the last change rather than the number of interactions from other users.

9.2 Gauge Types

Each liquidity gauge is assigned a type within the gauge controller. Grouping gauges by type allows the DAO to adjust the emissions according to type, making it possible to e.g. end all emissions for a single type.

Currently active gauge types are as follows:

- Ethereum: `0`
- Fantom: `1`
- Polygon (Matic): `2`
- xDai: `4`
- Crypto Pools: `5`

Type `3` has been deprecated.

9.3 LiquidityGauge

Each pool has a unique liquidity gauge. Deployment addresses can be found in the *addresses reference* section of the documentation.

There are several versions of liquidity gauge contracts in use. Source code for these contracts is available on Github.

9.3.1 Querying Gauge Information

`LiquidityGauge.`**`lp_token`**`()` → address: view
: The address of the LP token that may be deposited into the gauge.

`LiquidityGauge.totalSupply -> uint256: view`
: The total amount of LP tokens that are currently deposited into the gauge.

`LiquidityGauge.`**`working_supply`**`()` → uint256: view
: The "working supply" of the gauge - the effective total LP token amount after all deposits have been *boosted*.

9.3.2 Querying User Information

`LiquidityGauge.`**`balanceOf`**(*addr: address*) → uint256: view
The current amount of LP tokens that `addr` has deposited into the gauge.

`LiquidityGauge.`**`working_balances`**(*addr: address*) → uint256: view
The "working balance" of a user - their effective balance after *boost* has been applied.

`LiquidityGauge.`**`claimable_tokens`**(*addr: address*) → uint256: nonpayable
The amount of currently mintable CRV for `addr` from this gauge.

Note: Calling this function modifies the state. Off-chain integrators can call it as though it were a `view` function, however on-chain integrators **must** use it as `nonpayable` or the call will revert.

```
>>> gauge.claimable_tokens.call(alice)
3849184923983248t5273
```

`LiquidityGauge.`**`integrate_fraction`**(*addr: address*) → uint256: view
The total amount of CRV, both mintable and already minted, that has been allocated to `addr` from this gauge.

9.3.3 Checkpoints

`LiquidityGauge.`**`user_checkpoint`**(*addr: address*) → bool: nonpayable
Record a checkpoint for `addr`, updating their boost.

Only callable by `addr` or `Minter` - you cannot trigger a checkpoint for another user.

`LiquidityGauge.kick(addr: address): nonpayable`
Trigger a checkpoint for `addr`. Only callable when the current boost for `addr` is greater than it should be, due to an expired veCRV lock.

9.3.4 Deposits and Withdrawals

`LiquidityGauge.deposit(amount: uint256, receiver: address = msg.sender): nonpayable`
Deposit LP tokens into the gauge.

Prior to depositing, ensure that the gauge has been approved to transfer `amount` LP tokens on behalf of the caller.

- `amount`: Amount of tokens to deposit
- `receiver`: Address to deposit for. If not given, defaults to the caller. If specified, the caller must have been previous approved via *`approved_to_deposit`*

```
>>> lp_token = Contract(gauge.lp_token())
>>> balance = lp_token.balanceOf(alice)

>>> lp_token.approve(gauge, balance, {'from': alice})
Transaction sent:␣
↪0xa791801ccc57ad4edcfcaff7b5dab1c9101b78cf978a8d7fc185d9194bd3c2fa
  Gas price: 20.0 gwei   Gas limit: 56299   Nonce: 23

>>> gauge.deposit(balance, {'from': alice})
```

(continues on next page)

(continued from previous page)

```
Transaction sent:␣
↪0xd4edcfcaff7b5dab1c9101b78cf978a8d7fc185d9194bd3c2faa791801ccc57a
  Gas price: 20.0 gwei   Gas limit: 187495   Nonce: 24
```

LiquidityGauge.withdraw(amount: uint256): nonpayable

Withdraw LP tokens from the gauge.

- `amount`: Amount of tokens to withdraw

```
>>> balance = gauge.balanceOf(alice)
>>> gauge.withdraw(balance, {'from': alice})
Transaction sent:␣
↪0x1b78cf978a8d7fc185d9194bd3c2faa791801ccc57ad4edcfcaff7b5dab1c910
  Gas price: 20.0 gwei   Gas limit: 217442   Nonce: 25
```

`LiquidityGauge.`**`approved_to_deposit`**(*caller: address*, *receiver: address*) → bool: view

Return the approval status for `caller` to deposit LP tokens into the gauge on behalf of `receiver`.

LiquidityGauge.set_approve_deposit(depositor: address, can_deposit: bool): nonpayable

Approval or revoke approval for another address to deposit into the gauge on behalf of the caller.

- `depositor`: Address to set approval for
- `can_deposit`: Boolean - can this address deposit on behalf of the caller?

```
>>> gauge.approved_to_deposit(bob, alice)
False

>>> gauge.set_approve_deposit(bob, True, {'from': alice})
Transaction sent:␣
↪0xc185d9194bd3c2faa791801ccc57ad4edcfcaff7b5dab1c9101b78cf978a8d7f
  Gas price: 20.0 gwei   Gas limit: 47442   Nonce: 26

>>> gauge.approved_to_deposit(bob, alice)
True
```

9.3.5 Killing the Gauge

LiquidityGauge.kill_me(): nonpayable

Toggle the killed status of the gauge.

This function may only be called by the *ownership or emergency admins* within the DAO.

A gauge that has been killed is unable to mint CRV. Any gauge weight given to a killed gauge effectively burns CRV. This should only be done in a case where a pool had to be killed due to a security risk, but the gauge was already voted in.

`LiquidityGauge.`**`is_killed`**`()` → bool: view

The current killed status of the gauge.

9.4 LiquidityGaugeReward

Along with measuring liquidity for CRV distribution, `LiquidityGaugeReward` stakes LP tokens into an SNX staking rewards contract and handles distribution of an the additional rewards token. Rewards gauges include the full API of *LiquidityGauge*, with the following additional methods:

9.4.1 Querying Reward Information

`LiquidityGaugeReward.`**`reward_contract`**`()` → address: view
The address of the staking rewards contract that LP tokens are staked into.

`LiquidityGaugeReward.`**`rewarded_token`**`()` → address: view
The address of the reward token being received from *`reward_contract`*.

`LiquidityGaugeReward.`**`is_claiming_rewards`**`()` → bool: view
Boolean indicating if rewards are currently being claimed by this gauge.

9.4.2 Calculating Claimable Rewards

Note: There is no single function that returns the currently claimable reward amount. To calculate:

```
>>> gauge.claimable_reward(alice) - gauge.claimed_rewards_for(alice)
97924174626247611803
```

`LiquidityGaugeReward.`**`claimable_reward`**`(`*addr: address*`)` → uint256: view
The total earned reward tokens, both claimed and unclaimed, for `addr`.

`LiquidityGaugeReward.`**`claimed_rewards_for`**`(`*addr: address*`)` → uint256: view
The number of reward tokens already claimed for `addr`.

9.4.3 Claiming Rewards

`LiquidityGaugeReward.claim_rewards(addr: address = msg.sender): nonpayable`
Claim reward tokens for an address. If `addr` is not specified, defaults to the caller.

9.5 LiquidityGaugeV2

The v2 liquidity gauge adds a full ERC20 interface to the gauge, tokenizing deposits so they can be directly transferred between accounts without having to withdraw and redeposit. It also improves flexibility for onward staking, allowing staking to be enabled or disabled at any time and handling up to eight reward tokens at once.

9.5.1 Querying Reward Information

`LiquidityGaugeV2.`**`reward_contract`**`()` → address: view
The address of the staking rewards contract that LP tokens are staked into.

`LiquidityGaugeV2.`**`rewarded_tokens`**`(`*idx: uint256*`)` → address: view
Getter for an array of rewarded tokens currently being received by *`reward_contract`*.

The contract is capable of handling up to eight reward tokens at once - if there are less than eight currently active, some values will return as `ZERO_ADDRESS`.

9.5.2 Approvals and Transfers

`LiquidityGaugeV2.`**`transfer`**`(`*_to: address*, *_value: uint256*`)` → bool:
Transfers gauge deposit from the caller to `_to`.

This is the equivalent of calling `withdraw(_value)` followed by `deposit(_value, _to)`. Pending reward tokens for both the sender and receiver are also claimed during the transfer.

Returns `True` on success. Reverts on failure.

`LiquidityGaugeV2.`**`transferFrom`**`(`*_from: address*, *_to: address*, *_value: uint256*`)` → bool:
Transfers a gauge deposit between `_from` and `_to`.

The caller must have previously been approved to transfer at least `_value` tokens on behalf of `_from`. Pending reward tokens for both the sender and receiver are also claimed during the transfer.

Returns `True` on success. Reverts on failure.

`LiquidityGaugeV2.`**`approve`**`(`*_spender: address*, *_value: uint256*`)` → bool:
Approve the passed address to transfer the specified amount of tokens on behalf of the caller.

Returns `True` on success. Reverts on failure.

9.5.3 Checking and Claiming Rewards

Note: Rewards are claimed automatically each time a user deposits or withdraws from the gauge, and on gauge token transfers.

`LiquidityGaugeV2.`**`claimable_reward`**`(`*_addr: address*, *_token: address*`)` → uint256: nonpayable
Get the number of claimable reward tokens for a user.

Note: This function determines the claimable reward by actually claiming and then returning the received amount. As such, it is state changing and only of use to off-chain integrators. The mutability should be manually changed to `view` within the ABI.

- `_addr` Account to get reward amount for
- `_token` Token to get reward amount for

Returns the number of tokens currently claimable for the given address.

`LiquidityGaugeV2.claim_rewards(_addr: address = msg.sender): nonpayable`
Claim all available reward tokens for `_addr`. If no address is given, defaults to the caller.

LiquidityGaugeV2.claim_historic_rewards(_reward_tokens: address[8], _addr: address = msg.se

Claim reward tokens available from a previously-set staking contract.

- `_reward_tokens`: Array of reward token addresses to claim
- `_addr`: Address to claim for. If none is given, defaults to the caller.

9.5.4 Setting the Rewards Contract

LiquidityGaugeV2.set_rewards(_reward_contract: address, _sigs: bytes32, _reward_tokens: ad

Set the active reward contract.

- `_reward_contract`: Address of the staking contract. Set to `ZERO_ADDRESS` if staking rewards are being removed.
- `_sigs`: A concatenation of three four-byte function signatures: `stake`, `withdraw` and `getReward`. The signatures are then right padded with empty bytes. See the example below for more information on how to prepare this data.
- `_reward_tokens`: Array of rewards tokens received from the staking contract.

This action is only possible via the contract admin. It cannot be called when the gauge has no deposits. As a safety precaution, this call validates all the signatures with the following sequence of actions:

1. LP tokens are deposited into the new staking contract, verifying that the deposit signature is correct.
2. `balanceOf` is called on the LP token to confirm that the gauge's token balance is now zero.
3. The LP tokens are withdrawn, verifying that the withdraw function signature is correct.
4. `balanceOf` is called on the LP token again, to confirm that the gauge has successfully withdrawn it's entire balance.
5. A call to claim rewards is made to confirm that it does not revert.

These checks are required to protect against an incorrectly designed staking contract or incorrectly structured input arguments.

It is also possible to claim from a reward contract that does not require onward staking. In this case, use `00000000` for the function selectors for both staking and withdrawing.

An example of generating the signatures input and enabling a vanilla SNX rewards contract:

```
>>> Rewards = Contract("0x99ac10631f69c753ddb595d074422a0922d9056b")

# first, we get the signatures for depositing, withdrawing and claiming
>>> sigs = [rewards.stake.signature, rewards.withdraw.signature, rewards.
↪getReward.signature]
>>> sigs
["0xa694fc3a", "0x2e1a7d4d", "0x3d18b912"]

# now we remove the leading 0x and concatentate them
>>> sigs = "".join(i[2:] for i in sigs)
>>> sigs
"a694fc3a2e1a7d4d3d18b912"

# finally, we add the leading 0x and trailing 00 bytes
>>> sigs = "0x" + sigs + ("00" * 20)
>>> sigs
"0xa694fc3a2e1a7d4d3d18b912000000000000000000000000000000000000000000000000"
```

(continues on next page)

(continued from previous page)

```
# now we are ready to set the rewards contract
>>> gauge.set_rewards(rewards, sigs, [reward_token] + [ZERO_ADDRESS] * 7,
↪{'from': alice})
```

9.6 LiquidityGaugeV3

`LiquidityGaugeV3` is the current iteration of liquidity gauge used for curve pools on Ethereum mainnet. It retains a majority of `LiquidityGaugeV2`'s functionality such as tokenized deposits, and flexible onward staking with up to 8 reward tokens with some modifications.

Outline of modified functionality:

1. Ability to redirect claimed rewards to an alternative account.
2. Opt-in claiming of rewards on interactions with the gauge, instead of auto-claiming.
3. Retrieving rewards from the reward contract happens at a minimum of once an hour, for reduced gas costs.
4. Expose the amount of claimed and claimable rewards for users.
5. Removal of `claim_historic_rewards` function.
6. Modify `claimable_reward` to be a slightly less accurate view function.
7. Reward tokens can no longer be removed once set, adding more tokens requires providing the array of reward_tokens with any new tokens appended.
8. `deposit(_value, _to)` and `withdraw(_value, _to)` functions have an additional optional argument `_claim_rewards`, which when set to `True` will claim any pending rewards.

As this gauge maintains a similar API to `LiquidityGaugeV2`, the documentation only covers functions that were added or modified since the previous version.

9.6.1 Querying Reward Information

`LiquidityGaugeV3.`**`rewards_receiver`**(*addr: address*) → address: view
: This gauge implementation allows for the redirection of claimed rewards to alternative accounts. If an account has enabled a default rewards receiver this function will return that default account, otherwise it'll return `ZERO_ADDRESS`.

`LiquidityGaugeV3.`**`last_claim`**`()` → uint256: view
: The epoch timestamp of the last call to claim from `reward_contract`.

9.6.2 Checking and Claiming Rewards

Note: Unlike `LiquidityGaugeV2`, rewards are **not** automatically claimed each time a user performs an action on the gauge.

`LiquidityGaugeV3.claim_rewards(_addr: address = msg.sender, _receiver: address = ZERO_ADDR`
: Claim all available reward tokens for `_addr`. If no address is given, defaults to the caller. If the `_receiver` argument is provided rewards will be distributed to the address specified (caller must be `_addr` in this case). If the `_receiver` argument is not provided, rewards are sent to the default receiver for the account if one is set.

`LiquidityGaugeV3.`**`claimed_reward`**(*_addr: address*, *_token: address*) → uint256: view
Get the number of already claimed reward tokens for a user.

`LiquidityGaugeV3.`**`claimable_reward`**(*_addr: address*, *_token: address*) → uint256: view
Get the number of claimable reward tokens for a user

Note: This call does not consider pending claimable amount in `reward_contract`. Off-chain callers should instead use *`claimable_reward_write`* as a view method.

`LiquidityGaugeV3.`**`claimable_reward_write`**(*_addr: address*, *_token: address*) → uint256: non-payable
Get the number of claimable reward tokens for a user. This function should be manually changed to "view" in the ABI. Calling it via a transaction will checkpoint a user's rewards updating the value of *`claimable_reward`*. This function does not claim/distribute pending rewards for a user.

9.7 GaugeController

`GaugeController` is deployed to the Ethereum mainnet at:

0x2F50D538606Fa9EDD2B11E2446BEb18C9D5846bB.

This is a fixed address, the contract cannot be swapped out or upgraded.

Source code for this contract is available on Github.

9.7.1 Querying Gauge and Type Weights

`GaugeController.`**`gauge_types`**(*gauge_addr: address*) → int128: view
The gauge type for a given address, as an integer.

Reverts if `gauge_addr` is not a gauge.

`GaugeController.`**`get_gauge_weight`**(*gauge_addr: address*) → uint256: view
The current gauge weight for `gauge_addr`.

`GaugeController.`**`get_type_weight`**(*type_id: int128*) → uint256: view
The current type weight for `type_id` as an integer normalized to 1e18.

`GaugeController.`**`get_total_weight`**() → uint256: view
The current total (type-weighted) weight for all gauges.

`GaugeController.`**`get_weights_sum_per_type`**(*type_id: int128*) → uint256: view
The sum of all gauge weights for `type_id`.

9.7.2 Vote-Weighting

Vote weight power is expressed as an integer in bps (units of 0.01%). `10000` is equivalent to a 100% vote weight.

`GaugeController.`**`vote_user_power`**(*user: address*) → uint256: view
The total vote weight power allocated by `user`.

`GaugeController.`**`last_user_vote`**(*user: address*, *gauge: address*) → uint256: view
Epoch time of the last vote by `user` for `gauge`. A gauge weight vote may only be modified once every 10 days.

GaugeController.**vote_user_slopes**(*user: address*, *gauge: address*)
Information about `user`'s current vote weight for `gauge`.

Returns the current slope, allocated voting power, and the veCRV locktime end.

```
>>> slope = gauge_controller.vote_user_slopes(alice, gauge)

>>> slope['power']  # the current vote weight for this gauge
4200
```

GaugeController.vote_for_gauge_weights(_gauge_addr: address, _user_weight: uint256): nonpay
Allocate voting power for changing pool weights.

- _gauge_addr Gauge which *msg.sender* votes for
- _user_weight Weight for a gauge in bps (units of 0.01%). Minimal is 0.01%. Ignored if 0

```
>>> gauge_controller = Contract(
↪"0x2F50D538606Fa9EDD2B11E2446BEb18C9D5846bB")

>>> gauge_controller.vote_for_gauge_weights(my_favorite_gauge, 10000, {
↪'from': alice})
Transaction sent:␣
↪0xc185d9194bd3c2faa791801ccc57ad4edcfcaff7b5dab1c9101b78cf978a8d7f
  Gas price: 20.0 gwei   Gas limit: 47442   Nonce: 26
```

9.7.3 Adding New Gauges and Types

All of the following methods are only be callable by the DAO *ownership admin* as the result of a successful *vote*.

GaugeController.add_gauge(addr: address, gauge_type: int128): nonpayable
Add a new gauge.

- `addr`: Address of the new gauge being added
- `gauge_type`: Gauge type

Note: Once a gauge has been added it cannot be removed. New gauges should be very carefully verified prior to adding them to the gauge controller.

GaugeController.**gauge_relative_weight**(*addr: address*, *time: uint256 = block.timestamp*) → uint256: view
Get the relative the weight of a gauge normalized to 1e18 (e.g. 1.0 == 1e18).

Inflation which will be received by this gauge is calculated as `inflation_rate * relative_weight / 1e18`. * `addr`: Gauge address * `time`: Epoch time to return a gauge weight for. If not given, defaults to the current block time.

GaugeController.add_type(_name: String[64], weight: uint256 = 0): nonpayable
Add a new gauge type.

- `_name`: Name of gauge type
- `weight`: Weight of gauge type

GaugeController.**change_type_weight**(*type_id: int128*, *weight: uint256*)
Change the weight for a given gauge type.

Only callable by the DAO *ownership admin*.

- `type_id` Gauge type id
- `weight` New Gauge weight

9.8 Minter

`Minter` is deployed to the Ethereum mainnet at:

0xd061D61a4d941c39E5453435B6345Dc261C2fcE0.

This is a fixed address, the contract cannot be swapped out or upgraded.

Source code for this contract is available on Github.

9.8.1 Minting CRV

`Minter.mint(gauge_addr: address): nonpayable`
Mint allocated tokens for the caller based on a single gauge.

- `gauge_addr`: `LiquidityGauge` address to get mintable amount from

`Minter.mint_many(gauge_addrs: address[8]): nonpayable`
Mint CRV for the caller from several gauges.

- `gauge_addr`: A list of `LiquidityGauge` addresses to mint from. If you wish to mint from less than eight gauges, leave the remaining array entries as `ZERO_ADDRESS`.

`Minter.mint_for(gauge_addr: address, for: address): nonpayable`
Mint tokens for a different address.

In order to call this function, the caller must have been previously approved by `for` using `toggle_approve_mint`.

- `gauge_addr`: `LiquidityGauge` address to get mintable amount from
- `for`: address to mint for. The minted tokens are sent to this address, not the caller.

`Minter.toggle_approve_mint(minting_user: address): nonpayable`
Toggle approval for `minting_user` to mint CRV on behalf of the caller.

`Minter`.**`allowed_to_mint_for`**(*minter: address*, *for: address*) → bool: view
Getter method to check if `minter` has been approved to call mint_for on behalf of `for`.

CHAPTER

TEN

THE CURVE DAO: GAUGES FOR EVM SIDECHAINS

In addition to Ethereum, Curve is active on several sidechains.

The Curve DAO is sufficiently complex that it cannot be easily bridged outside of Ethereum, however aspects of functionality (including CRV emissions) are capable on the various sidechains where pools are active.

Source code for the smart contracts used in sidechain emissions are available on Github.

Note: Each sidechain comes with it's own set of tradeoffs between security, scalability and cost of use. The technical specifications and security considerations of each sidechain is outside the scope of this documentation, however we encourage all users to do their own research prior to transferring funds off of Ethereum and onto a sidechain.

10.1 Implementation Details

At a high level, the process of CRV distribution on sidechain gauges is as follows:

1. On Ethereum, a `RootChainGauge` contract mints allocated CRV each week and transfers it over the bridge.

   At the beginning of each epoch week, a call is made to the `checkpoint` function within each gauge. This function mints all of the allocated CRV for the previous week, and transfers them over the bridge to another contract deployed at the same address on the related sidechain. Emissions are delayed by one week in order to avoid exceeding the max allowable supply of CRV.

   Checkpointing may be performed by anyone. However, for chains that use the AnySwap bridge the checkpoint must happen via the `CheckpointProxy` contract.

2. On the sidechain, CRV is received into a `ChildChainStreamer` contract and then streamed out to a `RewardsOnlyGauge`.

   The bridge automatically transfers CRV into a streamer contract, deployed at the same address on the sidechain as the gauge is on Ethereum. Once the CRV has arrived, a call is made to `notify_reward_amount`. This call updates the local accounting and streams the balance out linearly over the next seven days.

3. Liquidity providers who have staked their LP tokens in the `RewardsOnlyGauge` may claim their CRV.

   The sidechain gauge is a simplified version of the gauges used on Ethereum. It handles CRV as though it were any other 3rd-party reward token, evenly distributing between stakers based on the deposited balances as the time the token is received.

10.2 RootChainGauge

`RootChainGauge` is a simplified liquidity gauge contract used for bridging CRV from Ethereum to a sidechain. Each root gauge is added to the gauge controller and receives gauge weight votes to determine emissions for a sidechain pool.

The gauge cannot be directly staked into. There is one important external function:

`RootChainGauge.checkpoint(): nonpayable`

Mints all allocated CRV emissions for the gauge, and transfers across the bridge.

This function should be called once per week, immediately after the start of the epoch week. Subsequent calls within the same epoch week have no effect.

For gauges that use the AnySwap bridge, this function is guarded and can only be called indirectly via `CheckpointProxy.checkpoint_many`.

10.3 ChildChainStreamer

`ChildChainStreamer` is a simple reward streaming contract. The logic is similar to that of the Synthetix staking rewards contract.

For each `RootChainGauge` deployed on Ethereum, a `ChildChainStreamer` is deployed at the same address on the related sidechain. CRV tokens that are sent over the bridge are transferred into the streamer. From there they are released linearly over seven days, to the gauge where LPs ultimately stake and claim them.

`ChildChainStreamer.notify_reward_amount(token: address):`

Notify the contract of a newly received reward. This updates the local accounting and streams the reward over a preset period (typically seven days).

If the previous reward period has already expired, this function is callable by anyone. When there is an active reward period it may only be called by the designated reward distributor account. Without this check, it would be possible to exploit the system by repeatedly calling to extend an active reward period and thus dragging out the duration over which the rewards are released.

Reverts if `token` is not registered as a reward within the contract, or if no extra balance of `token` was added prior to the call.

10.4 RewardsOnlyGauge

`RewardsOnlyGauge` is a simplified version of the same gauge contract used on Ethereum. The logic around CRV emissions and minting has been removed - it only deals with distribution of externally received rewards.

The API for this contract is similar to that of `LiquidityGaugeV3`.

10.5 RewardClaimer

`RewardClaimer` is a minimal passthrough contract that allows claiming from multiple reward streamers. For example the am3CRV pool on Polygon utilizes this contract to receive both CRV emissions bridged across from Ethereum, as well as WMATIC rewards supplied via a `RewardStreamer` contract. The `RewardsOnlyGauge` calls the `RewardClaimer` as a way to retrieve both the CRV and WMATIC rewards.

CHAPTER

ELEVEN

CURVE DAO: FEE COLLECTION AND DISTRIBUTION

Curve exchange contracts have the capability to charge an "admin fee", claimable by the contract owner. The admin fee is represented as a percentage of the total fee collected on a swap.

For exchanges the fee is taken in the output currency and calculated against the final amount received. For example, if swapping from USDT to USDC, the fee is taken in USDC.

Liquidity providers also incur fees when adding or removing liquidity. The fee is applied such that, for example, a swap between USDC and USDT would pay roughly the same amount of fees as depositing USDC into the pool and then withdrawing USDT. The only case where a fee is not applied on withdrawal is when removing liquidity via *remove_liquidity*, as this method does not change the imbalance of the pool in any way.

Exchange contracts are indirectly owned by the Curve DAO via a *proxy ownership contract*. This contract includes functionality to withdraw the fees, convert them to 3CRV, and forward them into the fee distributor contract. Collectively, this process is referred to as "burning".

Note: The burn process involves multiple transactions and is very gas intensive. Anyone can execute any step of the burn process at any time and there is no hard requirement that it happens in the correct order. However, running the steps out of order can be highly inefficient. If you wish to burn, it is recommended that you review all of the following information so you understand exactly what is happening.

11.1 Withdrawing Admin Fees

Admin fees are stored within each exchange contract and viewable via the *admin_balances* public getter method. The contract owner may call to claim the fees at any time using *withdraw_admin_fees*. Most pools also include a function to donate pending fees to liquidity providers via *donate_admin_fees*.

Fees are initially claimed via `PoolProxy.withdraw_many`. This withdraws fees from many pools at once, pulling them into the `PoolProxy` contract.

11.2 The Burn Process

Burning is handled on a per-coin basis. The process is initiated by calling the `PoolProxy.burn` or `PoolProxy.burn_many` functions. Calling to burn a coin transfers that coin into the burner and then calls the `burn` function on the burner.

Each `burn` action typically performs one conversion into another asset; either 3CRV itself, or something that is a step closer to reaching 3CRV. As an example, here is the sequence of conversions required to burn HBTC:

```
HBTC -> WBTC -> sBTC -> sUSD -> USDC -> 3CRV
```

Efficiency within the intermediate conversions is the reason it is important to run the burn process in a specific order. If you burn sBTC prior to burning HBTC, you will have to burn sBTC a second time!

There are a total of **nine** burner contracts, each of which handles a different category of fee coin. The following list also outlines the rough sequence in which burners should be executed:

- `LPBurner`: LP tokens in non-3CRV denominated metapools
- `SynthBurner`: non-USD denominated assets that are synths or can be swapped into synths
- `ABurner`: Aave lending tokens
- `CBurner`: Compound lending tokens
- `YBurner`: Yearn lending tokens
- `MetaBurner`: USD denominated assets that are directly swappable for 3CRV
- `USDNBurner`: USDN
- `UniswapBurner`: Assets that must be swapped on Uniswap/Sushiswap
- `UnderlyingBurner`: Assets that can be directly deposited into 3pool, or swapped for an asset that is deposited into 3pool

Source code for burners is available on Github.

11.2.1 LPBurner

The LP Burner handles non-3CRV LP tokens, collected from metapools. The most common token burned via the LP burner is sbtcCRV from BTC metapools.

LP burner calls to `StableSwap.remove_liquidity_one_coin` to unwrap the LP token into a single asset. The new asset is then transferred on to another burner.

The burner is configurable via the following functions:

`LPBurner.`**`set_swap_data`**(*lp_token: address*, *coin: address*, *burner: address*) → bool: nonpayable

Set conversion and transfer data for `lp_token`

- `lp_token`: LP token address
- `coin`: Address of the underlying coin to remove liquidity in
- `burner`: Burner to transfer `coin` to

This function is callable by the ownership admin and so requires a successful DAO vote.

Returns `True`.

11.2.2 SynthBurner

The synth burner is used to convert non-USD denominated assets into sUSD. This is accomplished via synth conversion, the same mechanism used in *cross-asset swaps*.

When the synth burner is called to burn a non-synthetic asset, it uses *`RegistrySwap.exchange_with_best_rate`* to swap into a related synth. If no direct path to a synth is avaialble, a swap is made into an intermediate asset.

For synths, the burner first transfers to the *underlying burner*. Then it calls `UnderlyingBurner.convert_synth`, performing the cross-asset swap within the underlying burner. This is done to avoid requiring another transfer call after the settlement period has passed.

The optimal sequence when burning assets using the synth burner is thus:

1. Coins that cannot directly swap to synths
2. Coins that can directly swap to synths
3. Synthetic assets

The burner is configurable via the following functions:

`SynthBurner.`**`set_swap_for`**`(`*_coins: address[10]*, *_targets: address[10]*`)` → bool:
Set target coins that the burner will swap into.

- `coins`: Array of coin addresses that will be burnt. If you wish to set less than 10, fill the remaining array slots with `ZERO_ADDRESS`.
- `targets`: Array of coin addresses to be swapped into. The address as index `n` within this list corresponds to the address at index `n` within `coins`.

For assets that can be directly swapped for a synth, the target should be set as that synth. For assets that cannot be directly swapped, the target must be an asset that has already had it's own target registered (e.g. can be swapped for a synth).

This function is unguarded. All targets are validated using the registry.

Returns `True`.

`SynthBurner.`**`add_synths`**`(`*_synths: address[10]*`)` → bool:
Register synthetic assets within the burner.

- `synths`: List of synths to register

This function is unguarded. For each synth to be added, a call is made to Synth.currencyKey to validate the addresss and obtain the synth currency key.

Returns `True`.

11.2.3 ABurner, CBurner, YBurner

`ABurner`, `CBurner` and `YBurner` are collectively known as "lending burners". They unwrap lending tokens into the underlying asset and transfer those assets onward into the *underlying burner*.

There is no configuration required for these burners.

11.2.4 MetaBurner

The meta-burner is used for assets within metapools that can be directly swapped for 3CRV. It uses the registry's `exchange_with_best_rate` and transfers 3CRV directly to the *fee distributor*.

There is no configuration required for this burner.

11.2.5 USDNBurner

The USDN burner is a special case that handles only USDN. Due to incompatibilities between the USDN pool and how USDN accrues interest, this burner is required to ensure the LPs recieve a fair share of that interest.

The burn process consists of:

1. 50% of the USDN to be burned is transferred back into the pool.
2. The burner calls to `donate_admin_fees`, creditting the returned USDN to LPs
3. The remaining USDN is swapped for 3CRV and transferred directly to the *fee distributor*.

There is no configuration required for this burner.

11.2.6 UniswapBurner

`UniswapBurner` is used for burning assets that are not supported by Curve, such as SNX recieved by the DAO via the Synthetix trading incentives program.

The burner works by querying swap rates on both Uniswap and Sushiswap using a path of `initial asset -> wETH -> USDC`. It then performs the swap on whichever exchange offers a better rate. The received USDC is sent into the *underlying burner*.

There is no configuration required for this burner.

11.2.7 UnderlyingBurner

The underlying burner handles assets that can be directly swapped to USDC, and deposits DAI/USDC/USDT into 3pool to obtain 3CRV. This is the final step of the burn process for many assets that require multiple intermediate swaps.

Note: Prior to burning any assets with the underlying burner, you shoudl have completed the entire process with `SynthBurner`, `UniswapBurner` and all of the lending burners.

The burn process consists of:

- For sUSD, first call settle to complete any pending synth conversions. Then, swap into USDC.
- for all other assets that are not DAI/USDC/USDT, swap into USDC.
- For DAI/USDC/USDT, only transfer the asset into the burner.

Once the entire burn process has been completed you must call `execute` as the final action:

`UnderlyingBurner.`**`execute`**`()` → bool:

Adds liquidity to 3pool and transfers the received 3CRV to the fee distributor.

This is the final function to be called in the burn process, after all other steps are completed. Calling this funciton does nothing if the burner has a zero balance of any of DAI, USDC and USDT.

There is no configuration required for this burner.

11.3 Fee Distribution

Fees are distributed to veCRV holders via the `FeeDistributor` contract. The contract is deployed to the Ethereum mainnet at:

0xA464e6DCda8AC41e03616F95f4BC98a13b8922Dc

Source code for this contract is available on Github.

Fees are distributed weekly. The porportional amount of fees that each user is to receive is calculated based on their veCRV balance relative to the total veCRV supply. This amount is calculated at the *start* of the week. The actual distribution occurs at the *end* of the week based on the fees that were collected. As such, a user that creates a new vote-lock should expect to receive their first fee payout at the end of the following epoch week.

The available 3CRV balance to distribute is tracked via the "token checkpoint". This is updated at minimum every 24 hours. Fees that are received between the last checkpoint of the previous week and first checkpoint of the new week will be split evenly between the weeks.

FeeDistributor.checkpoint_token(): nonpayable

Updates the token checkpoint.

The token checkpoint tracks the balance of 3CRV within the distributor, to determine the amount of fees to distribute in the given week. The checkpoint can be updated at most once every 24 hours. Fees that are received between the last checkpoint of the previous week and first checkpoint of the new week will be split evenly between the weeks.

To ensure full distribution of fees in the current week, the burn process must be completed prior to the last checkpoint within the week.

A token checkpoint is automatically taken during any `claim` action, if the last checkpoint is more than 24 hours old.

`FeeDistributor.`**`claim`**(*addr: address = msg.sender*) → uint256: nonpayable

Claims fees for an account.

- `addr`: The address to claim for. If none is given, defaults to the caller.

Returns the amount of 3CRV received in the claim. For off-chain integrators, this function can be called as though it were a view method in order to check the claimable amount.

Note: Every veCRV related action (locking, extending a lock, increasing the locktime) increments a user's veCRV epoch. A call to claim will consider at most 50 user epochs. For accounts that performed many veCRV actions, it may be required to call claim more than once to receive the fees. In such cases it can be more efficient to use *claim_many*.

```
>>> distro = Contract("0xA464e6DCda8AC41e03616F95f4BC98a13b8922Dc")
>>> distro.claim.call({'from': alice})
1323125068357710082803

>>> distro.claim({'from': alice})
Transaction sent:␣
↪0xa7978a8d7fb185d9194bd3c2fa1801ddd57ad4edcfcaff7b5dab1c9101b78cf9
  Gas price: 92.0 gwei   Gas limit: 256299   Nonce: 42
```

`FeeDistributor.`**`claim_many`**(*receivers: address[20]*) → bool: nonpayable

Perform multiple claims in a single call.

- `receivers`: An array of address to claim for. Claiming terminates at the first `ZERO_ADDRESS`.

This is useful to claim for multiple accounts at once, or for making many claims against the same account if that account has performed more than 50 veCRV related actions.

Returns `True`.

CHAPTER

TWELVE

THE CURVE DAO: GOVERNANCE AND VOTING

Curve uses Aragon for governance and control of the protocol admin functionality. Interaction with Aragon occurs through a modified implementation of the Aragon Voting App.

Much of the following functionality is possible via the DAO section of the Curve website. The following section outlines DAO interactions via the CLI using the Brownie console.

Deployment addresses can be found in the *addresses reference* section of the documentation.

12.1 Creating a Vote

A single vote can perform multiple actions. The new_vote.py script in the DAO repo is used to create new votes.

1. Modify the `TARGET`, `ACTIONS` and `DESCRPTION` variables at the beginning of the script. The comments within the script explain how each of these variables work.
2. Simulate the vote in a forked mainnet:

   ```
   brownie run voting/new_vote simulate --network mainnet-fork
   ```

   The simulation creates the vote, votes for it until quorum is reached, and then executes. The vote was successful if none of the transactions within the simulation fail. You can optionally include the `-I` flag to inspect the result of the vote once the simulation completes.
3. Create the vote:
   1. Modify the `SENDER` variable to use an account that is permitted to make a vote for the DAO you are targetting.
   2. Create the vote with the following command:

      ```
      brownie run voting/new_vote make_vote --network mainnet
      ```

   3. The vote should automatically appear within the site UX shortly.

12.2 Inspecting Votes

The decode_vote.py script in the DAO repo is used to decode a vote in order to see which action(s) it will perform.

To use the script, start by modifying the `VOTE_ID` and `VOTING_ADDRESS` variables at the start of the script. Then run the following:

```
brownie run voting/decode_vote --network mainnet
```

The script will output a list of transactions to be performed by the vote.

12.3 Voting

To place a vote via the CLI, first open a Brownie console connected to mainnet. Then use the following commands:

```
>>> aragon = Contract(VOTING_ADDRESS)
>>> aragon.vote(VOTE_ID, MY_VOTE, False, {'from': acct})
Transaction sent:␣
↪0xa791801ccc57ad4edcfcaff7b5dab1c9101b78cf978a8d7fc185d9194bd3c2fa
  Gas price: 20.0 gwei   Gas limit: 156299   Nonce: 23
```

- `VOTING_ADDRESS` is one of the voting addresses given above
- `VOTE_ID` is the numeric ID of the vote
- `MY_VOTE` is a boolean

12.4 Executing a Vote

To execute a vote via the CLI, first open a Brownie console connected to mainnet. Then use the following commands:

```
>>> aragon = Contract(VOTING_ADDRESS)
>>> aragon.executeVote({'from': acct})
Transaction sent:␣
↪0x85d9194bd3c2fa1801ccc57ad4edcfa7978a8d7fc1caff7b5dab1c9101b78cf9
  Gas price: 20.0 gwei   Gas limit: 424912   Nonce: 24
```

- `VOTING_ADDRESS` is one of the voting addresses given above

CHAPTER

THIRTEEN

CURVE DAO: PROTOCOL OWNERSHIP

The Curve DAO controls admin functionality throughout the protocol. Performing calls to to owner/admin-level functions is only possible via a successful DAO vote.

Ownership is handled via a series of proxy contracts. At a high level, the flow of ownership is:

```
DAO -> Aragon Agent -> Ownership Proxy -> Contracts
```

At the ownership proxy level there are two main contracts:

- `PoolProxy`: Admin functionality for *exchange contracts*
- `GaugeProxy`: Admin functionality for *liquidity gauges*

The DAO is capable of replacing the ownership proxies via a vote. Deployment addresses for the current contracts can be found in the *addresses reference* section of the documentation.

13.1 Agents

The Curve DAO has a total of three Aragon Agent ownership addresses, which are governed by two independent DAOs:

1. The **Community DAO** (or just "the DAO") governs the day-to-day operation of the protocol.

   Voting is based on a user's holdings of "Vote Escrowed CRV" (veCRV). veCRV is obtained by locking CRV for up to 4 years, with 1 veCRV equal to 1 CRV locked for 4 years. As the lock time decreases, An account's veCRV balance decreases linearly as the time remaining until unlock decreases. veCRV is non-transferrable.

   An account must have a minimum balance of 2500 veCRV to make a DAO vote. Each vote lasts for one week. Votes cannot be executed until the entire week has passed.

   The DAO has ownership of two admin accounts:

   - The **ownership admin** controls most functionality within the protocol. Performing an action via the ownership admin requires a 30% quorum with 51% support.
   - The **parameter admin** has authority to modify parameters on pools, such as adjusting the amplification co-efficient. Performing an action via the paramater admin requries a 15% quorum with 51% support.

2. The **Emergency DAO** has limited authority to kill pools and gauges during extraordinary circumstances.

   The emergency DAO consists of nine members, comprised of a mix of the Curve team and prominent figures within the DeFi community. Each member has one vote. Any member may propose a vote.

   All members of the emergency DAO may propose new votes. A vote lasts for 24 hours and can be executed immediately once it receives 66% support.

13.2 PoolProxy

`PoolProxy` is used for indirect ownership of *exchange contracts*.

Source code for this contract is available on Github.

13.2.1 Configuring Fee Burners

`PoolProxy`.**burners**(*coin: address*) → address: view
Getter for the burner contract address for `coin`.

PoolProxy.set_burner(coin: address, burner: address): nonpayable
Set burner of `coin` to `burner` address.

Callable only by the ownership admin.

PoolProxy.set_many_burners(coins: address[20], burners: address[20]): nonpayable
Set burner contracts for many coins at once.

- `coins`: Array of coin addresses. If you wish to set less than 20 burners, fill the remaining array slots with `ZERO_ADDRESS`.
- `burners`: Array of burner addresses. The address as index `n` within this list corresponds to the address at index `n` within `coins`.

Callable only by the ownership admin.

PoolProxy.set_donate_approval(pool: address, caller: address, is_approved: bool): nonpayabl
Set approval for an address to call `donate_admin_fees` on a specific pool.

- `pool`: Pool address
- `caller`: Adddress to set approval for
- `is_approved`: Approval status

Callable only by the ownership admin.

PoolProxy.set_burner_kill(_is_killed: bool): nonpayable
Disable or enable the process of fee burning.

Callable by the emergency and ownership admins.

13.2.2 Withdraing and Burning Fees

PoolProxy.withdraw_admin_fees(pool: address): nonpayable
Withdraw admin fees from `pool` into this contract.

This is the first step in fee burning. This function is unguarded - it may be called by any address.

PoolProxy.withdraw_many(pools: address[20]): nonpayable
Withdraw fees from multiple pools in a single call.

This function is unguarded.

PoolProxy.burn(coin: address): nonpayable
Transfer the contract's balance of `coin` into the preset burner and execute the burn process.

Only callable via an externally owned account; a check that `tx.origin == msg.sender` is performed to prevent potential flashloan exploits.

PoolProxy.burn_many(coins: address[20]): nonpayable
Execute the burn process on many coins at once.

Note that burning can be very gas intensive. In some cases burning 20 coins at once is not possible due to the block gas limit.

PoolProxy.donate_admin_fees(_pool: address): nonpayable
Donate a pool's current admin fees to the pool LPs.

Callable by the ownership admin, or any address given explicit permission to do so via `set_donate_approval`

13.2.3 Killing Pools

PoolProxy.kill_me(_pool: address): nonpayable
Pauses the pool.

When paused, it is only possible for existing LPs to remove liquidity via `remove_liquidity`. Exchanges and adding or removing liquidity in other ways are blocked.

Callable only by the emergency admin.

PoolProxy.unkill_me(_pool: address): nonpayable
Unpause a pool that was previously paused, re-enabling exchanges.

Callable by the emergency and ownership admins.

13.2.4 Pool Ownership

PoolProxy.commit_transfer_ownership(pool: address, new_owner: address): nonpayable
Initiate an ownership transfer of `pool` to `new_owner`.

Callable only by the ownership admin.

PoolProxy.accept_transfer_ownership(pool: address): nonpayable
Accept ending ownership transfer for `pool`.

This function is unguarded.

PoolProxy.revert_transfer_ownership(pool: address): nonpayable
Cancel a pending ownership transfer for `pool`.

Callable by the emergency and ownership admins.

13.2.5 Modifying Pool Parameters

PoolProxy.commit_new_parameters(pool: address, amplification: uint256, new_fee: uint256, ne
Initiate a change of parameters for a pool.

- `pool`: Pool address
- `amplification` New Amplification coefficient
- `new_fee` New fee
- `new_admin_fee` New admin fee
- `min_asymmetry` Minimal asymmetry factor allowed.

Asymmetry factor is: `Prod(balances) / (Sum(balances) / N) ** N`

Callable only by the parameter admin.

PoolProxy.apply_new_parameters(_pool: address): nonpayable
Apply a parameter change on a pool.

This function is unguarded, however it can only be called via an EOA to minimize the likelihood of a flashloan exploit.

PoolProxy.revert_new_parameters(_pool: address): nonpayable
Revert comitted new parameters for `pool`

Callable by the emergency and ownership admins.

PoolProxy.ramp_A(_pool: address, _future_A: uint256, _future_time: uint256): nonpayable
Start a gradual increase of the amplification coefficient for a pool.

- `_pool`: Pool address
- `future_A`: New amplification coefficient to ramp to
- `future_time`: Epoch time to complete the ramping at

Callable only by the parameter admin.

PoolProxy.stop_ramp_A(pool: address): nonpayable
Stop the gradual ramping of pool's amplification coefficient.

Callable by the emergency and parameter admins.

PoolProxy.commit_new_fee(pool: address, new_fee: uint256, new_admin_fee: uint256):
Initiate change in the fees for a pool.

- ``pool`:` Pool address
- `new_fee`: New fee
- `new_admin_fee`: New admin fee

Callable only by the parameter admin.

PoolProxy.apply_new_fee(_pool: address): nonpayable
Apply a fee change to a pool.

This function is unguarded.

13.3 GaugeProxy

`GaugeProxy` is used for indirect ownership of *liquidity gauges*.

Source code for this contract is available on Github.

GaugeProxy.set_rewards(gauge: address, reward_contract: address, sigs: bytes32, reward_toke
Set the active reward contract for a `LiquidityGaugeV2` deployment.

See the gauge documentation for details on how this function works.

- `gauge` Gauge address
- `reward_contract`: Address of the staking contract. Set to `ZERO_ADDRESS` if staking rewards are being removed.

- `sigs`: A concatenation of three four-byte function signatures: `stake`, `withdraw` and `getReward`. The signatures are then right padded with empty bytes. See the example below for more information on how to prepare this data.
- `reward_tokens`: Array of rewards tokens received from the staking contract.

Callable by the ownership admin.

GaugeProxy.set_killed(gauge: address, is_killed: bool): nonpayable

Set the killed status for a gauge.

- `gauge` Gauge address
- `is_killed` Killed status to set

Once killed, a gauge always yields a rate of 0 and so cannot mint CRV. Any vote-weight given to a killed gauge effectively burns CRV.

Callable by the ownership admin or the emergency admin.

GaugeProxy.commit_transfer_ownership(gauge: address, new_owner: address): nonpayable

Initiate the transfer of ownership of a gauge.

- `gauge`: Address of the gauge to transfer ownership of
- `new_owner`: New owner address

Callable only by the ownership admin.

GaugeProxy.accept_transfer_ownership(gauge: address): nonpayable

Apply ownership transfer of a gauge.

This function is unguarded. After `commit_transfer_ownership` has been called by the current owner, anyone can call into `GaugeProxy` to trigger the acceptance.

CHAPTER

FOURTEEN

REGISTRY

The Curve registry contracts are open source and may be found on Github.

The registry is comprised of the following contracts:

- `AddressProvider`: Address provider for registry contracts. This contract is immutable and it's address will never change. On-chain integrators should always use this contract to fetch the current address of other registry components.
- `Registry`: The main registry contract. Used to locate pools and query information about them as well as registered coins.
- `PoolInfo`: Aggregate getter methods for querying large data sets about a single pool. Designed for off-chain use.
- `Swaps`: The registry exchange contract. Used to find pools and query exchange rates for token swaps. Also provides a unified exchange API that can be useful for on-chain integrators.

CHAPTER

FIFTEEN

REGISTRY: ADDRESS PROVIDER

`AddressProvider` is an address provider for registry contracts.

Source code for this contract is available on Github.

15.1 How it Works

The address provider is deployed to the Ethereum mainnet at:

0x0000000022D53366457F9d5E68Ec105046FC4383

This contract is immutable. The address will never change.

The address provider is the point-of-entry for on-chain integrators. All other contracts within the registry are assigned an ID within the address provider. IDs start from zero and increment as new components are added. The address associated with an ID may change, but the API of the associated contract will not.

Integrators requiring an aspect of the registry should always start by querying the address provider for the current address of the desired component. An up-to-date list of registered IDs is available *here*.

To interact with the address provider using the Brownie console:

```
$ brownie console --network mainnet
Brownie v1.11.10 - Python development framework for Ethereum

Brownie environment is ready.
>>> provider = Contract.from_explorer('0x0000000022D53366457F9d5E68Ec105046FC4383')
Fetching source of 0x0000000022D53366457F9d5E68Ec105046FC4383 from api.etherscan.io...

>>> provider
<AddressProvider Contract '0x0000000022D53366457F9d5E68Ec105046FC4383'>
```

15.2 View Functions

`AddressProvider.`**`get_registry`**`()` → address: view

Get the address of the main registry contract.

This is a more gas-efficient equivalent to calling *`get_address(0)`*.

```
>>> provider.get_registry()
'0x90E00ACe148ca3b23Ac1bC8C240C2a7Dd9c2d7f5'
```

AddressProvider.**get_address**(*id: uint256*) → address: view
Fetch the address associated with `id`.

```
>>> provider.get_address(1)
'0xe64608E223433E8a03a1DaaeFD8Cb638C14B552C'
```

AddressProvider.**get_id_info**(*id: uint256*) → address, bool, uint256, uint256, string: view
Fetch information about the given `id`.

Returns a tuple of the following:

- `address`: Address associated to the ID.
- `bool`: Is the address at this ID currently set?
- `uint256`: Version of the current ID. Each time the address is modified, this number increments.
- `uint256`: Epoch timestamp this ID was last modified.
- `string`: Human-readable description of the ID.

```
>>> provider.get_id_info(1).dict()
{
    'addr': "0xe64608E223433E8a03a1DaaeFD8Cb638C14B552C",
    'description': "PoolInfo Getters",
    'is_active': True,
    'last_modified': 1604019085,
    'version': 1
}
```

AddressProvider.**max_id**() → uint256: view
Get the highest ID set within the address provider.

```
>>> provider.max_id()
1
```

15.3 Address IDs

- `0`: The main *registry contract*. Used to locate pools and query information about them.
- `1`: Aggregate getter methods for querying large data sets about a single pool. Designed for off-chain use.
- `2`: Generalized swap contract. Used for finding rates and performing exchanges.
- `3`: The *metapool factory*.
- `4`: The *fee distributor*. Used to distribute collected fees to veCRV holders.

CHAPTER

SIXTEEN

REGISTRY

`Registry` is the main registry contract. It is used to locate pools and query information about them.

Source code for this contract is available on Github.

16.1 Deployment Address

Use the *`get_registry`* method to get the address of the main registry from the address provider:

```
>>> provider = Contract('0x0000000022D53366457F9d5E68Ec105046FC4383')
>>> provider.get_registry()
'0x90E00ACe148ca3b23Ac1bC8C240C2a7Dd9c2d7f5'
```

16.2 View Functions

Because Vyper does not support dynamic-length arrays, all arrays have a fixed length. Excess fields contain zero values.

16.2.1 Finding Pools

`Registry.`**`pool_count`**`()` → uint256: view
Return the number of pools currently registered within the contract.

```
>>> registry.pool_count()
18
```

`Registry.`**`pool_list`**`(`*i: uint256*`)` → address: view
Master list of registered pool addresses.

Note that the ordering of this list is not fixed. Index values of addresses may change as pools are added or removed.

Querying values greater than *Registry.pool_count* returns `0x00`.

```
>>> registry.pool_list(7)
'0x8474DdbE98F5aA3179B3B3F5942D724aFcdec9f6'
```

`Registry.`**`get_pool_from_lp_token`**`(`*lp_token: address*`)` → address: view
Get the pool address for a given Curve LP token.

```
>>> registry.get_pool_from_lp_token(
↪'0x1AEf73d49Dedc4b1778d0706583995958Dc862e6')
'0x8474DdbE98F5aA3179B3B3F5942D724aFcdec9f6'
```

Registry.**get_lp_token**(*pool: address*) → address: view
Get the LP token address for a given Curve pool.

```
>>> registry.get_lp_token('0x8474DdbE98F5aA3179B3B3F5942D724aFcdec9f6')
'0x1AEf73d49Dedc4b1778d0706583995958Dc862e6'
```

Registry.**find_pool_for_coins**(*_from: address*, *_to: address*, *i: uint256 = 0*) → address: view
Finds a pool that allows for swaps between `_from` and `_to`. You can optionally include `i` to get the n-th pool, when multiple pools exist for the given pairing.

The order of `_from` and `_to` does not affect the result.

Returns `0x00` when swaps are not possible for the pair or `i` exceeds the number of available pools.

```
>>> registry.find_pool_for_coins(
↪'0x6b175474e89094c44da98b954eedeac495271d0f',
↪'0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48')
'0xbEbc44782C7dB0a1A60Cb6fe97d0b483032FF1C7'

>>> registry.find_pool_for_coins(
↪'0x6b175474e89094c44da98b954eedeac495271d0f',
↪'0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48', 1)
'0x79a8C46DeA5aDa233ABaFFD40F3A0A2B1e5A4F27'
```

16.2.2 Getting Info About a Pool

Coins and Coin Info

Registry.**get_n_coins**(*pool: address*) → uint256[2]: view
Get the number of coins and underlying coins within a pool.

```
>>> registry.get_n_coins('0x8474DdbE98F5aA3179B3B3F5942D724aFcdec9f6')
(2, 4)
```

Registry.**get_coins**(*pool: address*) → address[8]: view
Get a list of the swappable coins within a pool.

```
>>> registry.get_coins('0x8474DdbE98F5aA3179B3B3F5942D724aFcdec9f6')
("0xe2f2a5C287993345a840Db3B0845fbC70f5935a5",
↪"0x6c3F90f043a72FA612cbac8115EE7e52BDe6E490",
↪"0x0000000000000000000000000000000000000000",
↪"0x0000000000000000000000000000000000000000",
↪"0x0000000000000000000000000000000000000000",
↪"0x0000000000000000000000000000000000000000",
↪"0x0000000000000000000000000000000000000000",
↪"0x0000000000000000000000000000000000000000")
```

Registry.**get_underlying_coins**(*pool: address*) → address[8]: view
Get a list of the swappable underlying coins within a pool.

For pools that do not involve lending, the return value is identical to *Registry.get_coins*.

```
>>> registry.get_underlying_coins(
↪'0x8474DdbE98F5aA3179B3B3F5942D724aFcdec9f6')
("0xe2f2a5C287993345a840Db3B0845fbC70f5935a5",
↪"0x6B175474E89094C44Da98b954EedeAC495271d0F",
↪"0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48",
↪"0xdAC17F958D2ee523a2206206994597C13D831ec7",
↪"0x0000000000000000000000000000000000000000",
↪"0x0000000000000000000000000000000000000000",
↪"0x0000000000000000000000000000000000000000",
↪"0x0000000000000000000000000000000000000000")
```

`Registry.`**`get_decimals`**(*pool: address*) → uint256[8]: view

Get a list of decimal places for each coin within a pool.

```
>>> registry.get_decimals('0x8474DdbE98F5aA3179B3B3F5942D724aFcdec9f6')
(18, 18, 0, 0, 0, 0, 0, 0)
```

`Registry.`**`get_underlying_decimals`**(*pool: address*) → uint256[8]: view

Get a list of decimal places for each underlying coin within a pool.

For pools that do not involve lending, the return value is identical to *`Registry.get_decimals`*. Non-lending coins that still involve querying a rate (e.g. renBTC) are marked as having `0` decimals.

```
>>> registry.get_underlying_decimals(
↪'0x8474DdbE98F5aA3179B3B3F5942D724aFcdec9f6')
(18, 18, 6, 6, 0, 0, 0, 0)
```

`Registry.`**`get_coin_indices`**(*pool: address*, *_from: address*, *_to: address*) → (int128, int128, bool): view

Convert coin addresses into indices for use with pool methods.

Returns the index of `_from`, index of `_to`, and a boolean indicating if the coins are considered underlying in the given pool.

```
>>> registry.get_coin_indices('0x79a8C46DeA5aDa233ABaFFD40F3A0A2B1e5A4F27
↪', '0xdac17f958d2ee523a2206206994597c13d831ec7',
↪'0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48')
(2, 1, True)
```

Based on the above call, we know:

- the index of the coin we are swapping out of is `2`
- the index of the coin we are swapping into is `1`
- the coins are considred underlying, so we must call `exchange_underlying`

From this information we can perform a token swap:

```
>>> swap = Contract('0x79a8C46DeA5aDa233ABaFFD40F3A0A2B1e5A4F27')
>>> swap.exchange_underlying(2, 1, 1e18, 0, {'from': alice})
```

Balances and Rates

`Registry.`**`get_balances`**(*pool: address*) → uint256[8]: view
Get available balances for each coin within a pool.

These values are not necessarily the same as calling `Token.balanceOf(pool)` as the total balance also includes unclaimed admin fees.

```
>>> registry.get_balances('0x79a8C46DeA5aDa233ABaFFD40F3A0A2B1e5A4F27')
(114281613944286898232752 27, 47831326741306, 45418708932136,␣
↪48777578907442492245548483, 0, 0, 0, 0)
```

`Registry.`**`get_underlying_balances`**(*pool: address*) → uint256[8]: view
Get balances for each underlying coin within a pool.

For pools that do not involve lending, the return value is identical to *Registry.get_balances*.

```
>>> registry.get_underlying_balances(
↪'0x79a8C46DeA5aDa233ABaFFD40F3A0A2B1e5A4F27')
(11876658145799734093379928, 48715210997790596223520238,␣
↪46553896776332824101242804, 49543896565857325657915396, 0, 0, 0, 0)
```

`Registry.`**`get_admin_balances`**(*pool: address*) → uint256[8]: view
Get the current admin balances (uncollected fees) for a pool.

```
>>> registry.get_admin_balances(
↪'0x79a8C46DeA5aDa233ABaFFD40F3A0A2B1e5A4F27')
(10800690926373756722358, 30891687335, 22800662409,␣
↪16642955874751891466129, 0, 0, 0, 0)
```

`Registry.`**`get_rates`**(*pool: address*) → uint256[8]: view
Get the exchange rates between coins and underlying coins within a pool, normalized to a `1e18` precision.

For non-lending pools or non-lending coins within a lending pool, the rate is `1e18`.

```
>>> registry.get_rates('0x79a8C46DeA5aDa233ABaFFD40F3A0A2B1e5A4F27')
(1039244956550111510, 1018479293504725874, 1024993895758183341,␣
↪1015710454247817308, 0, 0, 0, 0)
```

`Registry.`**`get_virtual_price_from_lp_token`**(*lp_token: address*) → uint256: view
Get the virtual price of a pool LP token.

```
>>> registry.get_virtual_price_from_lp_token(
↪'0x3B3Ac5386837Dc563660FB6a0937DFAa5924333B')
1060673685385893596
```

Pool Parameters

`Registry.`**`get_A`**(*pool: address*) → uint256: view
Get the current amplification coefficient for a pool.

```
>>> registry.get_A('0x79a8C46DeA5aDa233ABaFFD40F3A0A2B1e5A4F27')
500
```

`Registry.`**`get_fees`**(*pool: address*) → uint256[2]: view
Get the fees for a pool.

Fees are expressed as integers with a `1e10` precision. The first value is the total fee, the second is the percentage of the fee taken as an admin fee.

A typical return value here is `(4000000, 5000000000)` - a 0.04% fee, 50% of which is taken as an admin fee.

```
>>> registry.get_fees('0x79a8C46DeA5aDa233ABaFFD40F3A0A2B1e5A4F27')
(4000000, 5000000000)
```

`Registry.`**`get_parameters`**(*pool: address*) → PoolParams: view

Get all parameters for a given pool.

The return value is a struct, organized as follows:

```
struct PoolParams:
    A: uint256
    future_A: uint256
    fee: uint256
    admin_fee: uint256
    future_fee: uint256
    future_admin_fee: uint256
    future_owner: address
    initial_A: uint256
    initial_A_time: uint256
    future_A_time: uint256
```

Note that for older pools where `initial_A` is not public, this value is set to `0`.

```
>>> registry.get_parameters('0x79a8C46DeA5aDa233ABaFFD40F3A0A2B1e5A4F27').
↪dict()
{
    'A': 500,
    'admin_fee': 5000000000,
    'fee': 4000000,
    'future_A': 500,
    'future_A_time': 0,
    'future_admin_fee': 5000000000,
    'future_fee': 4000000,
    'future_owner': "0x56295b752e632f74a6526988eaCE33C25c52c623",
    'initial_A': 0,
    'initial_A_time': 0
}
```

Gas Estimates

`Registry.`**`estimate_gas_used`**(*pool: address*, *_from: address*, *_to: address*) → uint256: view
Get an estimate on the upper bound for gas used in an exchange.

Pool Metadata

`Registry.`**`is_meta`**(*pool: address*) → bool: view
Get a boolean identifying whether `pool` is a metapool.

```
>>> registry.is_meta('0x4f062658EaAF2C1ccf8C8e36D6824CDf41167956')
True
```

`Registry.`**`get_pool_name`**(*pool: address*) → String[64]: view
Get the name given to a pool upon registration.

```
>>> registry.get_pool_name('0x4f062658EaAF2C1ccf8C8e36D6824CDf41167956')
'gusd'
```

`Registry.`**`get_pool_asset_type`**(*pool: address*) → uint256: view
Get the asset type of specific `pool` as an integer.

Note: The asset type of a pool is subject to modification, and is primarily of use to off-chain integrators.

```
>>> registry.get_pool_asset_type('0x4f062658EaAF2C1ccf8C8e36D6824CDf41167956')
0
```

Asset types are as follows:

- `0`: USD
- `1`: BTC
- `2`: ETH
- `3`: Other StableSwap
- `4`: CryptoSwap

16.2.3 Gauges

`Registry.`**`gauge_controller`**() → address: view
Get the address of the Curve DAO GaugeController contract.

```
>>> registry.gauge_controller()
'0x2F50D538606Fa9EDD2B11E2446BEb18C9D5846bB'
```

`Registry.`**`get_gauges`**(*pool: address*) → (address[10], int128[10]): view
Get a list of LiquidityGauge contracts associated with a pool, and their gauge types.

```
>>> registry.get_gauges('0x79a8C46DeA5aDa233ABaFFD40F3A0A2B1e5A4F27')
(('0x8474DdbE98F5aA3179B3B3F5942D724aFcdec9f6',
↪'0x0000000000000000000000000000000000000000',
↪'0x0000000000000000000000000000000000000000',
↪'0x0000000000000000000000000000000000000000',
↪'0x0000000000000000000000000000000000000000',
↪'0x0000000000000000000000000000000000000000',
↪'0x0000000000000000000000000000000000000000',
↪'0x0000000000000000000000000000000000000000',
↪'0x0000000000000000000000000000000000000000',
↪'0x0000000000000000000000000000000000000000'), (0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0))
```

(continues on next page)

(continued from previous page)

16.2.4 Getting Coins and Coin Swap Complements

`Registry.`**`coin_count`**`()` → uint256: view
Get the total number of unique coins throughout all registered curve pools.

```
>>> registry.coin_count()
42
```

`Registry.`**`get_coin`**`(`*i: uint256*`)` → address: view
Get the *ith* unique coin throughout all registered curve pools.

Returns `0x0000000000000000000000000000000000000000` for values of *i* greater than the return value of *Registry.coin_count*.

```
>>> registry.get_coin(0)
'0x6B175474E89094C44Da98b954EedeAC495271d0F'
```

`Registry.`**`get_coin_swap_count`**`(`*coin: address*`)` → uint256: view
Get the total number of unique swaps available for `coin`.

```
>>> registry.get_coin_swap_count(
↪'0x6B175474E89094C44Da98b954EedeAC495271d0F')
12
```

`Registry.`**`get_coin_swap_complement`**`(`*coin: address*, *i: uint256*`)` → address: view
Get the *ith* unique coin available for swapping against `coin` across all registered curve pools.

```
>>> registry.get_coin_swap_complement(
↪'0x6B175474E89094C44Da98b954EedeAC495271d0F', 0)
'0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48'
```

16.2.5 Registry Metadata

`Registry.`**`last_updated`**`()` → uint256:: view
Get the epoch time of the last registry update.

Only successful state modifying functions (`add_pool`, `add_metapool`, `set_pool_gas_estimates`, etc.) will update this return value.

```
>>> registry.last_updated()
1617850905
```

CHAPTER

SEVENTEEN

REGISTRY: POOL INFO

`PoolInfo` contains aggregate getter methods for querying large data sets about a single pool. It is designed for off-chain use (not optimized for gas efficiency).

Source code for this contract is available on Github.

17.1 Deployment Address

The pool info contract is registered in the address provider with ID `1`. To get the current address:

```
>>> provider = Contract('0x0000000022D53366457F9d5E68Ec105046FC4383')
>>> provider.get_address(1)
'0xe64608E223433E8a03a1DaaeFD8Cb638C14B552C'
```

17.2 View Functions

`PoolInfo.`**`get_pool_coins`**(*pool: address*) → address[8], address[8], uint256[8], uint256[8]: view

Get information about the coins in a pool.

The return data is structured as follows:

- `address[8]`: Coin addresses
- `address[8]`: Underlying coin addresses
- `uint256[8]`: Coin decimal places
- `uint256[8]`: Coin underlying decimal places

```
>>> pool_info.get_pool_coins('0x79a8C46DeA5aDa233ABaFFD40F3A0A2B1e5A4F27').dict()
{
    'coins': (
        "0xC2cB1040220768554cf699b0d863A3cd4324ce32",
        "0x26EA744E5B887E5205727f55dFBE8685e3b21951",
        "0xE6354ed5bC4b393a5Aad09f21c46E101e692d447",
        "0x04bC0Ab673d88aE9dbC9DA2380cB6B79C4BCa9aE",
        "0x0000000000000000000000000000000000000000",
        "0x0000000000000000000000000000000000000000",
        "0x0000000000000000000000000000000000000000",
        "0x0000000000000000000000000000000000000000"
    ),
    'decimals': (18, 6, 6, 18, 0, 0, 0, 0),
```

(continues on next page)

(continued from previous page)

```
    'underlying_coins': (
        "0x6B175474E89094C44Da98b954EedeAC495271d0F",
        "0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48",
        "0xdAC17F958D2ee523a2206206994597C13D831ec7",
        "0x4Fabb145d64652a948d72533023f6E7A623C7C53",
        "0x0000000000000000000000000000000000000000",
        "0x0000000000000000000000000000000000000000",
        "0x0000000000000000000000000000000000000000",
        "0x0000000000000000000000000000000000000000"
    ),
    'underlying_decimals': (18, 6, 6, 18, 0, 0, 0, 0)
}
```

`PoolInfo.`**`get_pool_info`**(*pool: address*) → PoolInfo: view

Query information about a pool.

The return data is formatted using the following structs:

```
struct PoolInfo:
    balances: uint256[MAX_COINS]
    underlying_balances: uint256[MAX_COINS]
    decimals: uint256[MAX_COINS]
    underlying_decimals: uint256[MAX_COINS]
    rates: uint256[MAX_COINS]
    lp_token: address
    params: PoolParams
    is_meta: bool
    name: String[64]

# this struct is nested inside `PoolInfo`
struct PoolParams:
    A: uint256
    future_A: uint256
    fee: uint256
    admin_fee: uint256
    future_fee: uint256
    future_admin_fee: uint256
    future_owner: address
    initial_A: uint256
    initial_A_time: uint256
    future_A_time: uint256
```

An example query:

```
>>> pool_info.get_pool_info('0x79a8C46DeA5aDa233ABaFFD40F3A0A2B1e5A4F27').dict()
{
    'balances': (114281613944286898232752 27, 47831326741306, 45418708932136,␣
↪4877757890744249224554 8483, 0, 0, 0, 0),
    'decimals': (18, 6, 6, 18, 0, 0, 0, 0),
    'lp_token': "0x3B3Ac5386837Dc563660FB6a0937DFAa5924333B",
    'params': (500, 500, 4000000, 5000000000, 4000000, 5000000000,
↪"0x56295b752e632f74a6526988eaCE33C25c52c623", 0, 0, 0),
    'rates': (1039246194444517276, 1018480818866816704, 1024994762508449404,␣
↪1015710534981182027, 0, 0, 0, 0),
    'underlying_balances': (11876673238657763875985115,␣
↪4871528882697160226215 3927, 4655393877533512895 8626025,␣
↪4954390076716523411757 3778, 0, 0, 0, 0),
```

(continues on next page)

(continued from previous page)

```
    'underlying_decimals': (18, 6, 6, 18, 0, 0, 0, 0),
    'is_meta': False,
    'name': 'busd'
}
```

CHAPTER

EIGHTEEN

REGISTRY: EXCHANGES

The registry exchange contract is used to find pools and query exchange rates for token swaps. It also provides a unified exchange API that can be useful for on-chain integrators.

Source code for this contract is available on Github.

18.1 Deployment Address

The exchange contract is registered in the address provider with ID `2`. To get the current address:

```
>>> provider = Contract('0x0000000022D53366457F9d5E68Ec105046FC4383')
>>> provider.get_address(2)
'0xD1602F68CC7C4c7B59D686243EA35a9C73B0c6a2'
```

18.2 Finding Pools and Swap Rates

`Swaps.`**`get_best_rate`**`(`*_from: address*, *_to: address*, *_amount: uint256*, *_exclude_pools: address[8]*`)` → (address, uint256): view

Find the pool offering the best rate for a given swap.

- `_from`: Address of coin being sent.
- `_to`: Address of coin being received.
- `_amount`: Quantity of *_from* being sent.
- `_exclude_pools`: [optional] A list of up to 8 addresses which should be excluded from the query.

Returns the address of the pool offering the best rate, and the expected amount received in the swap.

`Swaps.`**`get_exchange_amount`**`(`*_pool: address*, *_from: address*, *_to: address*, *_amount: uint256*`)` → uint256: view

Get the current number of coins received in an exchange.

- `_pool`: Pool address.
- `_from`: Address of coin to be sent.
- `_to`: Address of coin to be received.
- `_amount`: Quantity of `_from` to be sent.

Returns the quantity of `_to` to be received in the exchange.

18.3 Swapping Tokens

Swaps.**exchange**(*_pool: address*, *_from: address*, *_to: address*, *_amount: uint256*, *_expected: uint256*, *_receiver: address = msg.sender*) → uint256: payable

Perform an token exchange using a specific pool.

- `_pool`: Address of the pool to use for the swap.
- `_from`: Address of coin being sent.
- `_to`: Address of coin being received.
- `_amount`: Quantity of `_from` being sent.
- `_expected`: Minimum quantity of `_to` received in order for the transaction to succeed.
- `_receiver`: Optional address to transfer the received tokens to. If not specified, defaults to the caller.

Returns the amount of `_to` received in the exchange.

Swaps.**exchange_with_best_rate**(*_from: address*, *_to: address*, *_amount: uint256*, *_expected: uint256*, *_receiver: address = msg.sender*) → uint256: payable

Perform an exchange using the pool that offers the best rate.

- `_from`: Address of coin being sent.
- `_to`: Address of coin being received.
- `_amount`: Quantity of *_from* being sent.
- `_expected`: Minimum quantity of `_to` received in order for the transaction to succeed.
- `_receiver`: Optional address to transfer the received tokens to. If not specified, defaults to the caller.

Returns the amount of `_to` received in the exchange.

> **Warning:** This function queries the exchange rate for every pool where a swap between `_to` and `_from` is possible. For pairs that can be swapped in many pools this will result in very significant gas costs!

CHAPTER

NINETEEN

METAPOOL FACTORY

The metapool factory allows for permissionless deployment of Curve metapools.

Source code for factory contracts may be viewed on Github.

19.1 Organization

The metapool factory has several core components:

- The *factory* is the main contract used to deploy new metapools. It also acts a registry for finding the deployed pools and querying information about them.
- *Pools* are deployed via a proxy contract. The implementation contract targetted by the proxy is determined according to the base pool. This is the same technique used to create pools in Uniswap V1.
- *Deposit contracts* ("zaps") are used for wrapping and unwrapping underlying assets when depositing into or withdrawing from pools.

CHAPTER

TWENTY

METAPOOL FACTORY: DEPLOYER AND REGISTRY

The `Factory` contract is used to deploy new Curve pools and to find existing ones. It is deployed to the mainnet at the following address:

0x0959158b6040D32d04c301A72CBFD6b39E21c9AE

Source code for this contract is may be viewed on Github.

Warning: Please carefully review the *limitations* of the factory prior to deploying a new pool. Deploying a pool using an incompatible token could result in permanent losses to liquidity providers and/or traders. Factory pools cannot be killed and tokens cannot be rescued from them!

20.1 Deploying a Pool

`Factory.`**`deploy_metapool`**(*_base_pool: address*, *_name: String[32]*, *_symbol: String[10]*, *_coin: address*, *_A: uint256*, *_fee: uint256*) → address: nonpayable

Deploys a new metapool.

- `_base_pool`: Address of the *base pool* to use within the new metapool.
- `_name`: Name of the new metapool.
- `_symbol`: Symbol for the new metapool's LP token. This value will be concatenated with the base pool symbol.
- `_coin`: Address of the coin being used in the metapool
- `_A`: *Amplification coefficient*
- `_fee`: *Trade fee*, given as an integer with 1e10 precision.

Returns the address of the newly deployed pool.

```
>>> factory = Contract('0x0959158b6040D32d04c301A72CBFD6b39E21c9AE')
>>> esd = Contract('0x36F3FD68E7325a35EB768F1AedaAe9EA0689d723')
>>> threepool = Contract('0xbEbc44782C7dB0a1A60Cb6fe97d0b483032FF1C7')

>>> tx = factory.deploy_metapool(threepool, "Empty Set Dollar", "ESD",␣
↪esd, 10, 4000000, {'from': alice})
Transaction sent:␣
↪0x2702cfc4b96be1877f853c246be567cbe8f80ef7a56348ace1d17c026bc31b68
  Gas price: 20 gwei   Gas limit: 1100000   Nonce: 9
```

(continues on next page)

(continued from previous page)

```
>>> tx.return_value
"0xFD9f9784ac00432794c8D370d4910D2a3782324C"
```

Note: After deploying a pool, you must also *add initial liquidity* before the pool can be used.

20.1.1 Limitations

- The token within the new pool must expose a `decimals` method and use a maximum of 18 decimal places.
- The token's `transfer` and `transferFrom` methods must revert upon failure.
- Successful token transfers must move exactly the specified number of tokens between the sender and receiver. Tokens that take a fee upon a successful transfer may cause the pool to break or act in unexpected ways.
- Token balances must not change without a transfer. Rebasing tokens are not supported!
- Pools deployed by the factory cannot be paused or killed.
- Pools deployed by the factory are not eligible for CRV rewards.

20.1.2 Base Pools

A metapool pairs a coin against the LP token of another pool. This other pool is referred to as the "base pool". By using LP tokens, metapools allow swaps against any asset within their base pool, without diluting the base pool's liquidity.

The factory allows deployment of metapools that use the following base pools:

- 3pool (USD denominated assets): 0xbEbc44782C7dB0a1A60Cb6fe97d0b483032FF1C7
- sBTC (BTC denominated assets): 0x7fC77b5c7614E1533320Ea6DDc2Eb61fa00A9714

It is possible to enable additional base pools through a DAO vote.

20.1.3 Choosing an Amplification Coefficient

The amplification co-efficient ("A") determines a pool's tolerance for imbalance between the assets within it. A higher value means that trades will incure slippage sooner as the assets within the pool become imbalanced.

The appropriate value for A is dependent upon the type of coin being used within the pool. We recommend the following values:

- Uncollateralized algorithmic stablecoins: 5-10
- Non-redeemable, collateralized assets: 100
- Redeemable assets: 200-400

It is possible to modify the amplification coefficient for a pool after it has been deployed. However, it requires a vote within the Curve DAO and must reach a 15% quorum.

20.1.4 Trade fees

Curve pools charge a fee for token exchanges and when adding or removing liquidity in an imbalanced manner. 50% of the fees are given to liquidity providers, 50% are distributed to veCRV holders.

For factory pools, the size of the fee is set at deployment. The minimum fee is 0.04% (represented as `4000000`). The maximum fee is 1% (`100000000`). The fee cannot be changed after a pool has been deployed.

20.2 Finding Pools

The following getter methods are available for finding pools that were deployed via the factory:

`Factory.`**`pool_count`**`()` → uint256: view
Returns the total number of pools that have been deployed by the factory.

`Factory.`**`pool_list`**`(`*i: uint256*`)` → address: view
Returns the n'th entry in a zero-indexed array of deployed pools. Returns `ZERO_ADDRESS` when `i` is greater than the number of deployed pools.

Note that because factory-deployed pools are not killable, they also cannot be removed from the registry. For this reason the ordering of pools within this array will never change.

`Factory.`**`find_pool_for_coins`**`(`*_from: address*, *_to: address*, *i: uint256 = 0*`)` → address: view
Finds a pool that allows for swaps between `_from` and `_to`. You can optionally include `i` to get the i-th pool, when multiple pools exist for the given pairing.

The order of `_from` and `_to` does not affect the result.

Returns `ZERO_ADDRESS` when swaps are not possible for the pair or `i` exceeds the number of available pools.

```
>>> esd = Contract('0x36F3FD68E7325a35EB768F1AedaAe9EA0689d723')
>>> usdc = Contract('0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48')

>>> factory.find_pool_for_coins(esd, usdc)
'0xFD9f9784ac00432794c8D370d4910D2a3782324C'
```

20.3 Getting Pool Info

The factory has a similar API to that of the main Registry, which can be used to query information about existing pools.

20.3.1 Coins and Coin Info

`Factory.`**`get_n_coins`**`(`*pool: address*`)` → uint256[2]: view
Get the number of coins and underlying coins within a pool.

```
>>> factory.get_n_coins('0xFD9f9784ac00432794c8D370d4910D2a3782324C')
(2, 4)
```

`Factory.`**`get_coins`**`(`*pool: address*`)` → address[2]: view
Get a list of the swappable coins within a pool.

```
>>> factory.get_coins('0xFD9f9784ac00432794c8D370d4910D2a3782324C')
("0x36F3FD68E7325a35EB768F1AedaAe9EA0689d723",
↪"0x6c3F90f043a72FA612cbac8115EE7e52BDe6E490")
```

`Factory.`**`get_underlying_coins`**(*pool: address*) → address[8]: view

Get a list of the swappable underlying coins within a pool.

```
>>> factory.get_underlying_coins(
↪'0xFD9f9784ac00432794c8D370d4910D2a3782324C')
("0x36F3FD68E7325a35EB768F1AedaAe9EA0689d723",
↪"0x6B175474E89094C44Da98b954EedeAC495271d0F",
↪"0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48",
↪"0xdAC17F958D2ee523a2206206994597C13D831ec7",
↪"0x0000000000000000000000000000000000000000",
↪"0x0000000000000000000000000000000000000000",
↪"0x0000000000000000000000000000000000000000",
↪"0x0000000000000000000000000000000000000000")
```

`Factory.`**`get_decimals`**(*pool: address*) → uint256[8]: view

Get a list of decimal places for each coin within a pool.

```
>>> factory.get_decimals('0xFD9f9784ac00432794c8D370d4910D2a3782324C')
(18, 18, 0, 0, 0, 0, 0, 0)
```

`Factory.`**`get_underlying_decimals`**(*pool: address*) → uint256[8]: view

Get a list of decimal places for each underlying coin within a pool.

For pools that do not involve lending, the return value is identical to *`Registry.get_decimals`*. Non-lending coins that still involve querying a rate (e.g. renBTC) are marked as having `0` decimals.

```
>>> factory.get_underlying_decimals(
↪'0xFD9f9784ac00432794c8D370d4910D2a3782324C')
(18, 18, 6, 6, 0, 0, 0, 0)
```

`Factory.`**`get_coin_indices`**(*pool: address*, *_from: address*, *_to: address*) → (int128, int128, bool): view

Convert coin addresses into indices for use with pool methods.

Returns the index of `_from`, index of `_to`, and a boolean indicating if the coins are considered underlying in the given pool.

```
>>> factory.get_coin_indices('0xFD9f9784ac00432794c8D370d4910D2a3782324C',
↪ '0xdac17f958d2ee523a2206206994597c13d831ec7',
↪'0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48')
(2, 1, True)
```

Based on the above call, we know:

- the index of the coin we are swapping out of is `2`
- the index of the coin we are swapping into is `1`
- the coins are considred underlying, so we must call `exchange_underlying`

From this information we can perform a token swap:

```
>>> swap = Contract('0xFD9f9784ac00432794c8D370d4910D2a3782324C')
>>> swap.exchange_underlying(2, 1, 1e18, 0, {'from': alice})
```

Balances and Rates

`Factory.`**`get_balances`**(*pool: address*) → uint256[2]: view
Get available balances for each coin within a pool.

These values are not necessarily the same as calling `Token.balanceOf(pool)` as the total balance also includes unclaimed admin fees.

```
>>> factory.get_balances('0xFD9f9784ac00432794c8D370d4910D2a3782324C')
(114281613944286898232752 27, 47831326741306)
```

`Factory.`**`get_underlying_balances`**(*pool: address*) → uint256[8]: view
Get balances for each underlying coin within a pool.

```
>>> factory.get_underlying_balances(
↪'0xFD9f9784ac00432794c8D370d4910D2a3782324C')
(11876658145799734093379928, 48715210997790596223520238,␣
↪46553896776332824101242804, 49543896565857325657915396, 0, 0, 0, 0)
```

`Factory.`**`get_admin_balances`**(*pool: address*) → uint256[2]: view
Get the current admin balances (uncollected fees) for a pool.

```
>>> factory.get_admin_balances('0xFD9f9784ac00432794c8D370d4910D2a3782324C
↪')
(10800690926373756722358, 30891687335)
```

`Factory.`**`get_rates`**(*pool: address*) → uint256[2]: view
Get the exchange rates between coins and underlying coins within a pool, normalized to a `1e18` precision.

```
>>> factory.get_rates('0xFD9f9784ac00432794c8D370d4910D2a3782324C')
(1000000000000000000, 1018479293504725874)
```

CHAPTER

TWENTYONE

METAPOOL FACTORY: POOLS

Factory pools are permissionless metapools that can be deployed by anyone. New pools are deployed using *Factory.deploy_metapool*.

Source code for the implementation contracts may be viewed on Github.

21.1 Implementation Contracts

Each pool deployed by the factory is a thin proxy contract created with Vyper's `create_forwarder_to`. The implementation contract targetted by the proxy is determined based on the base pool used. This is the same technique that was used to create pools in Uniswap V1.

The implementation contracts used for pools are deployed to the mainnet at the following addresses:

- 3pool: 0x5F890841f657d90E081bAbdB532A05996Af79Fe6
- sBTC: 0x2f956eee002b0debd468cf2e0490d1aec65e027f

When interacting with a factory pool you should use the ABI at the corresponding implementation address:

```
>>> implementation = Contract("0x5F890841f657d90E081bAbdB532A05996Af79Fe6")
>>> abi = implementation.abi
>>> pool = Contract.from_abi("ESD Pool",
↪"0xFD9f9784ac00432794c8D370d4910D2a3782324C", abi)
```

21.2 Getting Pool Info

`StableSwap.`**`coins`**`(`*i: uint256*`)` → address: view

Getter for the array of swappable coins within the pool. The last coin will always be the LP token of the base pool.

```
>>> pool.coins(0)
'0x36F3FD68E7325a35EB768F1AedaAe9EA0689d723'
```

`StableSwap.`**`balances`**`(`*i: uint256*`)` → uint256: view

Getter for the pool balances array.

```
>>> pool.balances(0)
489897529780862216 8122
```

`StableSwap.`**`A`**`()` → uint256: view

The *amplification coefficient* for the pool.

```
>>> pool.A()
10
```

StableSwap.**get_virtual_price**() → uint256: view
The current price of the pool LP token relative to the underlying pool assets. Given as an integer with 1e18 precision.

```
>>> pool.get_virtual_price()
1006391979770742306
```

StableSwap.**fee**() → uint256: view
The pool swap fee, as an integer with 1e10 precision.

```
>>> pool.fee()
4000000
```

StableSwap.**admin_fee**() → uint256: view
The percentage of the swap fee that is taken as an admin fee, as an integer with with 1e10 precision.

For factory pools this is hardcoded at 50% (5000000000).

```
>>> pool.admin_fee()
5000000000
```

21.3 Making Exchanges

StableSwap.**get_dy**(*i: int128*, *j: int128*, *dx: uint256*) → uint256: view
Get the amount received ("dy") when performing a swap between two assets within the pool.

Index values can be found using the *coins* public getter method, or *get_coins* within the factory contract.

- i: Index value of the coin to send.
- j: Index value of the coin to receive.
- dx: The amount of i being exchanged.

Returns the amount of j received.

```
>>> pool.get_dy(0, 1, 10**18)
460306318211728896
```

StableSwap.**get_dy_underlying**(*i: int128*, *j: int128*, *dx: uint256*) → uint256: view
Get the amount received ("dy") when swapping between two underlying assets within the pool.

Index values can be found using *get_underlying_coins* within the factory contract.

- i: Index value of the token to send.
- j: Index value of the token to receive.
- dx: The amount of i being exchanged.

Returns the amount of j received.

```
>>> pool.get_dy_underlying(0, 1, 10**18)
463415003137589177
```

StableSwap.**exchange**(*i: int128, j: int128, dx: uint256, min_dy: uint256, _receiver: address = msg.sender*) → uint256: nonpayable

Performs an exchange between two tokens.

Index values can be found using the *coins* public getter method, or *get_coins* within the factory contract.

- `i`: Index value of the token to send.
- `j`: Index value of the token to receive.
- `dx`: The amount of `i` being exchanged.
- `min_dy`: The minimum amount of `j` to receive. If the swap would result in less, the transaction will revert.
- `_receiver`: An optional address that will receive `j`. If not given, defaults to the caller.

Returns the amount of `j` received in the exchange.

```
>>> expected = pool.get_dy(0, 1, 10**18) * 0.99
>>> pool.exchange(0, 1, 10**18, expected, {'from': alice})
```

StableSwap.**exchange_underlying**(*i: int128, j: int128, dx: uint256, min_dy: uint256, _receiver: address = msg.sender*) → uint256: nonpayable

Perform an exchange between two underlying coins.

Index values can be found using *get_underlying_coins* within the factory contract.

- `i`: Index value of the underlying token to send.
- `j`: Index value of the underlying token to receive.
- `dx`: The amount of `i` being exchanged.
- `min_dy`: The minimum amount of `j` to receive. If the swap would result in less, the transaction will revert.
- `_receiver`: An optional address that will receive `j`. If not given, defaults to the caller.

Returns the amount of `j` received in the exchange.

```
>>> expected = pool.get_dy_underlying(0, 3, 10**18) * 0.99
>>> pool.exchange_underlying(0, 3, 10**18, expected, {'from': alice})
```

21.4 Adding and Removing Liquidity

Note that if you wish to add or remove liqudity using the underlying assets within the base pool, you must use a *depositor contract*.

StableSwap.**calc_token_amount**(*_amounts: uint256[2], _is_deposit: bool*) → uint256: view

Estimate the amount of LP tokens minted or burned based on a deposit or withdrawal.

This calculation accounts for slippage, but not fees. It should be used as a basis for determining expected amounts when calling *add_liquidity* or *remove_liquidity_imbalance*, but should not be considered to be precise!

- `_amounts`: Amount of each coin being deposited. Amounts correspond to the tokens at the same index locations within *coins*.
- `_is_deposit`: set `True` for deposits, `False` for withdrawals.

Returns the expected amount of LP tokens minted or burned.

StableSwap.**calc_withdraw_one_coin**(*_burn_amount: uint256*, *i: int128*) → uint256: view

Calculate the amount received when withdrawing and unwrapping in a single coin. Useful for setting `_max_burn_amount` when calling *remove_liquidity_one_coin*.

- `_pool`: Address of the pool to deposit into.
- `_token_amount`: Amount of LP tokens to burn in the withdrawal.
- `i`: Index value of the underlying coin to withdraw. Can be found using the *coins* getter method.

Returns the expected amount of coin received.

StableSwap.**add_liquidity**(*_deposit_amounts: uint256[2]*, *_min_mint_amount: uint256*, *_receiver: address = msg.sender*) → uint256: nonpayable

Deposits coins into to the pool and mints new LP tokens.

- `_deposit_amounts`: List of amounts of underlying coins to deposit. Amounts correspond to the tokens at the same index locations within *coins*.
- `_min_mint_amount`: Minimum amount of LP tokens to mint from the deposit.
- `_receiver`: Optional address that receives the LP tokens. If not specified, they are sent to the caller.

Returns the amount of LP tokens that were minted in the deposit.

```
>>> amounts = [1e18, 1e18]
>>> expected = pool.calc_token_amount(amounts, True) * 0.99
>>> pool.add_liquidity(amounts, expected, {'from': alice})
```

StableSwap.**remove_liquidity**(*_burn_amount: uint256*, *_min_amounts: uint256[2]*, *_receiver: address = msg.sender*) → uint256[2]: nonpayable

Withdraws coins from the pool and burns LP tokens.

Withdrawal amounts are based on current deposit ratios. Withdrawals using this method do not incur a fee.

- `_burn_amount`: Quantity of LP tokens to burn in the withdrawal. Amounts correspond to the tokens at the same index locations within *coins*.
- `_min_amounts`: Minimum amounts of coins to receive.
- `_receiver`: Optional address that receives the withdrawn coins. If not specified, the coins are sent to the caller.

Returns a list of the amounts of coins that were withdrawn.

```
>>> amount = pool.balanceOf(alice)
>>> pool.remove_liquidity(pool, amount, 0, {'from': alice})
```

StableSwap.**remove_liquidity_imbalance**(*_amounts: uint256[2]*, *_max_burn_amount: uint256*, *_receiver: address = msg.sender*) → uint256: nonpayable

Withdraw coins in an imbalanced amount.

- `_amounts`: List of amounts of underlying coins to withdraw. Amounts correspond to the tokens at the same index locations within *coins*.
- `_max_burn_amount`: Maximum number of LP token to burn in the withdrawal.
- `_receiver`: Optional address that receives the withdrawn coins. If not specified, the coins are sent to the caller.

Returns the amount of the LP tokens burned in the withdrawal.

```
>>> amounts = [1e18, 1e18]
>>> expected = pool.calc_token_amount(amounts, False) * 1.01
>>> pool.remove_liquidity_imbalance(pool, amounts, expected, {'from':␣
↪alice})
```

StableSwap.**remove_liquidity_one_coin**(*_burn_amount: uint256*, *i: int128*, *_min_received: uint256*, *_receiver: address = msg.sender*) → uint256: nonpayable

Withdraw a single asset from the pool.

- `_burn_amount`: Amount of LP tokens to burn in the withdrawal.
- `i`: Index value of the coin to withdraw. Can be found using the *`coins`* getter method.
- `_min_amount`: Minimum amount of the coin to receive
- `_receiver`: Optional address that receives the withdrawn coin. If not specified, the coin is sent to the caller.

Returns the amount of the coin received in the withdrawal.

```
>>> amount = pool.balanceOf(alice)
>>> expected = pool.calc_withdraw_one_coin(pool, amount, 0) * 1.01
>>> pool.remove_liquidity_one_coin(amount, expected, 0, {'from': alice})
```

21.5 Claiming Admin Fees

StableSwap.withdraw_admin_fees(): nonpayable

Transfer admin fees to the fee distributor, allowing the fees to be claimed by veCRV holders.

Anyone can call this method. The destination address for the fees is hardcoded. To simplify fee distribution, this method swaps the admin balance of the non-base pool LP token into the base pool LP token.

21.6 LP Tokens

Factory pools differ from traditional Curve pools in that the pool contract is also the LP token. This improves gas efficiency and simplifies the factory *deployment process*.

Pool contracts adhere to the ERC-20 standard. As such, the following methods are available:

21.6.1 Token Info

StableSwap.**name**() → String[64]: view

The name of the pool / LP token.

StableSwap.**symbol**() → String[32]: view

The token symbol.

StableSwap.**decimals**() → uint256: view

The number of decimals for the token. Curve pool tokens always use 18 decimals.

StableSwap.**totalSupply**() → uint256: view

21.6.2 Balances and Allowances

`StableSwap.`**`balanceOf`**`(`*_addr: address*`)` → uint256: view
Getter for the current balance of an account.

`StableSwap.`**`allowance`**`(`*_owner: address*, *_spender: address*`)` → uint256: view
Getter for the number of tokens `_owner` has approve `_spender` to transfer on their behalf.

`2**256-1` it is considered infinite approval. The approval amount will not decrease when tokens are transferred.

21.6.3 Transfers and Approvals

`StableSwap.`**`approve`**`(`*_spender: address*, *_value: uint256*`)` → bool: nonpayable
Approve `_spender` to transfer up to `_value` tokens on behalf of the caller.

If an approval is given for `2**256-1` it is considered infinite. The approval amount will not decrease when tokens are transferred, reducing gas costs.

- `_spender` Address to set the approval for
- `_value` Amount of the caller's tokens that `_spender` is permitted to transfer

Returns `True` on success. Reverts on failure.

`StableSwap.`**`transfer`**`(`*_to: address*, *_value: uint256*`)` → bool: nonpayable
Transfer tokens from the caller to the given address.

- `_to`: Address receiving the tokens.
- `_value`: Amount of tokens to be transferred.

Returns `True` on a successful call. Reverts on failure.

`StableSwap.`**`transferFrom`**`(`*_from: address*, *_to: address*, *_value: uint256*`)` → bool: nonpayable
Transfer tokens between two addresses. The caller must have been given approval to transfer tokens on behalf of `_from` or the call will revert.

- `_from`: The address to transfer the tokens from.
- `_to`: Address receiving the tokens.
- `_value`: mount of tokens to be transferred.

Returns `True` on a successful call. Reverts on failure.

CHAPTER

TWENTYTWO

METAPOOL FACTORY: ORACLES

Factory contracts include Time-Weighted Average Price oracles. To understand these a bit better, you need to understand how Curve calculates price.

A curve pool is an array of `balances` of the tokens it holds. To provide a price, it calculates how much of `x` you can receive given amount `y`.

22.1 Time-Weighted Average Price oracles

`MetaPool.`**`get_price_cumulative_last`**`()` → uint256[N_COINS]:
Returns the current time-weighted average price (TWAP). This will represent the underlying balances of the pool.

The value returned is the cumulative pool shifting balances over time

`MetaPool.`**`block_timestamp_last`**`()` → uint256:
Returns the last timestamp that a TWAP reading was taken in unix time.

`MetaPool.`**`get_twap_balances`**`(`*_first_balances: uint256[N_COINS]*, *_last_balances: uint256[N_COINS]*, *_time_elapsed: uint256*`)` → uint256[N_COINS]:
Calculate the current effective TWAP balances given two snapshots over time, and the time elapsed between the two snapshots.

- `_first_balances`: First `price_cumulative_last` array that was snapshot via `get_price_cumulative_last`
- `_last_balances`: Second `price_cumulative_last` array that was snapshot via `get_price_cumulative_last`
- `_time_elapsed`: The elapsed time in seconds between `_first_balances` and `_last_balances`

Returns the `balances` of the TWAP value.

`MetaPool.`**`get_dy`**`(`*i: int128*, *j: int128*, *dx: uint256*, *_balances: uint256[N_COINS] = [0, 0]*`)` → uint256:
Calculate the price for exchanging a token with index `i` to token with index `j` and amount `dx` given the `_balances` provided.

- `i`: The index of the coin being sent to the pool, as it related to the metapool
- `j`: The index of the coin being received from the pool, as it relates to the metapool
- `dx`: The amount of `i` being sent to the pool
- `_balances`: The array of balances to be used for purposes of calculating the output amount / exchange rate, this is the value returned in *get_twap_balances*

Returns the quote / price as `dy` given `dx`.

22.2 Security

The Curve TWAP is greatly inspired by Uniswap TWAP architecture, in that the price is a cumulative value over time, which reduces balance shifts due to flash loans, but also records the balances based on the previous block, to avoid recording flashloan data.

CHAPTER

TWENTYTHREE

METAPOOL FACTORY: DEPOSIT CONTRACTS

Deposit contracts (also known as "zaps") allow users to add and remove liquidity from a pool using the pool's underlying tokens.

23.1 Deployment Addresses

A single zap is used for all factory metapools targetting one base pool. The zaps are deployed to mainnet at the following addresses:

- 3pool: 0xA79828DF1850E8a3A3064576f380D90aECDD3359
- sBTC: 0x7AbDBAf29929e7F8621B757D2a7c04d78d633834

23.2 Calculating Expected Amounts

`DepositZap.`**`calc_withdraw_one_coin`**`(`*_pool: address*, *_token_amount: uint256*, *i: int128*`)` → uint256: view

Calculate the amount received when withdrawing and unwrapping in a single coin. Useful for setting `_max_burn_amount` when calling *`remove_liquidity_one_coin`*.

- `_pool`: Address of the pool to deposit into.
- `_token_amount`: Amount of LP tokens to burn in the withdrawal.
- `i`: Index value of the underlying coin to withdraw.

Returns the expected amount of coin received.

`DepositZap.`**`calc_token_amount`**`(`*_pool: address*, *_amounts: uint256[4]*, *_is_deposit: bool*`)` → uint256: view

Calculate addition or reduction in token supply from a deposit or withdrawal.

This calculation accounts for slippage, but not fees. It should be used as a basis for determining expected amounts when calling *`add_liquidity`* or *`remove_liquidity_imbalance`*, but should not be considered to be precise!

- `_pool`: Address of the pool to deposit into.
- `_amounts`: Amount of each underlying coin being deposited or withdrawn. Amounts correspond to the tokens at the same index locations within *`Factory.get_underlying_coins`*.
- `_is_deposit`: set `True` for deposits, `False` for withdrawals.

Returns the expected amount of LP tokens received.

23.3 Adding Liquidity

DepositZap.**add_liquidity**(*_pool: address*, *_deposit_amounts: uint256[4]*, *_min_mint_amount: uint256*, *_receiver: address = msg.sender*) → uint256: nonpayable

Wraps underlying coins and deposit them into `_pool`.

- `_pool`: Address of the pool to deposit into.
- `_deposit_amounts`: List of amounts of underlying coins to deposit. Amounts correspond to the tokens at the same index locations within *Factory.get_underlying_coins*.
- `_min_mint_amount`: Minimum amount of LP tokens to mint from the deposit.
- `_receiver`: Optional address that receives the LP tokens. If not specified, they are sent to the caller.

Returns the amount of LP tokens that were minted in the deposit.

```
>>> zap = Contract('0x7AbDBAf29929e7F8621B757D2a7c04d78d633834')
>>> pool = Contract('0xFD9f9784ac00432794c8D370d4910D2a3782324C')

>>> amounts = [1e18, 1e18, 1e6, 1e6]
>>> expected = zap.calc_token_amount(pool, amounts, True) * 0.99
>>> zap.add_liquidity(pool, amounts, expected, {'from': alice})
```

23.4 Removing Liquidity

DepositZap.**remove_liquidity**(*_pool: address*, *_burn_amount: uint256*, *_min_amounts: uint256[4]*, *_receiver: address = msg.sender*) → uint256[4]: nonpayable

Withdraw underlying coins from `_pool`.

Withdrawal amounts are based on current deposit ratios. Withdrawals using this method do not incur a fee.

- `_pool`: Address of the pool to withdraw from.
- `_burn_amount`: Quantity of LP tokens to burn in the withdrawal. Amounts correspond to the tokens at the same index locations within *Factory.get_underlying_coins*.
- `_min_amounts`: Minimum amounts of underlying coins to receive.
- `_receiver`: Optional address that receives the withdrawn coins. If not specified, the coins are sent to the caller.

Returns a list of the amounts of underlying coins that were withdrawn.

```
>>> zap = Contract('0x7AbDBAf29929e7F8621B757D2a7c04d78d633834')
>>> pool = Contract('0xFD9f9784ac00432794c8D370d4910D2a3782324C')

>>> amount = pool.balanceOf(alice)
>>> zap.remove_liquidity(pool, amount, 0, {'from': alice})
```

DepositZap.**remove_liquidity_one_coin**(*_pool: address*, *_burn_amount: uint256*, *i: int128*, *_min_amount: uint256*, *_receiver: address = msg.sender*) → uint256: nonpayable

Withdraw from `_pool` in a single coin.

- `_pool`: Address of the pool to withdraw from.
- `_burn_amount`: Amount of LP tokens to burn in the withdrawal
- `i`: Index value of the coin to withdraw. Can be found using *Factory.get_underlying_coins*.

- `_min_amount`: Minimum amount of underlying coin to receive
- `_receiver`: Optional address that receives the withdrawn coin. If not specified, the coin is sent to the caller.

Returns the amount of the underlying coin received in the withdrawal.

```
>>> zap = Contract('0x7AbDBAf29929e7F8621B757D2a7c04d78d633834')
>>> pool = Contract('0xFD9f9784ac00432794c8D370d4910D2a3782324C')

>>> amount = pool.balanceOf(alice)
>>> expected = zap.calc_withdraw_one_coin(pool, amount, 2) * 1.01
>>> zap.remove_liquidity_one_coin(pool, amount, expected, 2, {'from':␣
↪alice})
```

DepositZap.**remove_liquidity_imbalance**(*_pool: address*, *_amounts: uint256[N_ALL_COINS]*, *_max_burn_amount: uint256*, *_receiver: address = msg.sender*) → uint256: nonpayable

Withdraw coins from `_pool` in an imbalanced amount.

- `_pool`: Address of the pool to withdraw from.
- `_amounts`: List of amounts of underlying coins to withdraw. Amounts correspond to the tokens at the same index locations within *`Factory.get_underlying_coins`*.
- `_max_burn_amount`: Maximum number of LP token to burn in the withdrawal.
- `_receiver`: Optional address that receives the withdrawn coins. If not specified, the coins are sent to the caller.

Returns the amount of the LP tokens burned in the withdrawal.

```
>>> zap = Contract('0x7AbDBAf29929e7F8621B757D2a7c04d78d633834')
>>> pool = Contract('0xFD9f9784ac00432794c8D370d4910D2a3782324C')

>>> amounts = [1e18, 1e18, 1e6, 1e6]
>>> expected = zap.calc_token_amount(pool, amounts, False) * 1.01
>>> zap.remove_liquidity_imbalance(pool, amounts, expected, {'from':␣
↪alice})
```

Note: The deposit contract must be approved to transfer `_max_burn_amount` LP tokens from the caller or the transaction will fail.

CHAPTER

TWENTYFOUR

METAPOOL FACTORY: LIQUIDITY MIGRATOR

The `PoolMigrator` contract is used for migrating liquidity between Curve factory pools. It is deployed to the mainnet at the following address:

0xd6930b7f661257DA36F93160149b031735237594

Source code for this contract is may be viewed on Github.

24.1 Migrating Liquidity between Pools

`Factory.`**`migrate_to_new_pool`**(*_old_pool: address*, *_new_pool: address*, *_amount: uint256*) → uint256:

Migrate liquidity between two pools.

Each pool must be deployed by the curve factory (v1 or v2) and contain identical assets. Depending on the imbalance of each pool, the migration may incur slippage or provide a bonus.

Prior to calling this method, the caller must have given approval for the migrator to transfer up to `_amount` LP tokens from `_old_pool`.

- `_old_pool`: Address of the pool to migrate from
- `_new_pool`: Address of the pool to migrate into
- `_amount`: Number of `_old_pool` LP tokens to migrate

Returns the number of `_new_pool` LP tokens received in the migration.

```
>>> migrator = Contract('0xd6930b7f661257DA36F93160149b031735237594')
>>> old_pool = Contract('0x36F3FD68E7325a35EB768F1AedaAe9EA0689d723')
>>> new_pool = Contract('0x83D2944d5fC10A064451Dc5852f4F47759F249B6')

>>> balance = old_pool.balanceOf(alice)

>>> old_pool.approve(migrator, balance, {'from': alice})
Transaction sent:␣
↪0x8fc0dc0844ccbbed63d9cb7f2820087db5f70b320efea7ef4ce6b4a678e3cd45
  Gas price: 20 gwei   Gas limit: 1100000   Nonce: 9

>>> migrator.migrate_to_new_pool(old_pool, new_pool, balance, {'from':␣
↪alice})
Transaction sent:␣
↪0xd65182491c13b2620f84fe2d501ace5c8ab1cda1b9ea54d40f4f2351cccd52b6
  Gas price: 20 gwei   Gas limit: 1100000   Nonce: 10
```

CHAPTER

TWENTYFIVE

CONTRIBUTOR GUIDE

Curve is open-source and consists of a number of repositories on Github. Contributions are welcome!

On-chain contracts are spread across different Curve repositories as such:

- curve-contract: contracts for Curve stable swap pools
- curve-dao-contracts: contracts owned by DAO (e.g., liquidity gauges, burners and fee distributor)
- curve-pool-registry: contracts for on-chain information on pools and swaps

In order to make contributing to Curve as seamless as possible, please read thoroughly through these contribution guidelines.

CHAPTER

TWENTYSIX

GENERAL

26.1 Commit Messages

Contributors **should** adhere to the following standards and best practices when making commits to be merged into the Curve codebase.

26.1.1 Conventional Commits

Commit messages **should** adhere to the Conventional Commits standard. A conventional commit message is structured as follows:

```
<type>[optional scope]: <description>

[optional body]

[optional footer]
```

The commit contains the following elements, to communicate intent to the consumers of your library:

- **fix**: a commit of the *type* `fix` patches a bug in your codebase (this correlates with `PATCH` in semantic versioning).
- **feat**: a commit of the *type* `feat` introduces a new feature to the codebase (this correlates with `MINOR` in semantic versioning).
- **BREAKING CHANGE**: a commit that has the text `BREAKING CHANGE:` at the beginning of its optional body or footer section introduces a breaking API change. A BREAKING CHANGE can be part of commits of any *type*.

The use of commit types other than `fix:` and `feat:` is recommended. For example: `docs:`, `style:`, `refactor:`, `test:`, `chore:`, or `improvement:`.

26.1.2 Best Practices

We **recommend** the following best practices for commit messages (taken from How To Write a Commit Message:

- Limit the subject line to 50 characters.
- Use imperative, present tense in the subject line.
- Capitalize the subject line.
- Do not end the subject line with a period.

- Separate the subject from the body with a blank line.
- Wrap the body at 72 characters.
- Use the body to explain what and why vs. how.

26.2 Github Standard Fork and Pull Request Workflow

Original version: https://gist.github.com/Chaser324/ce0505fbed06b947d962

Whether you're trying to give back to the open source community or collaborating on your own projects, knowing how to properly fork and generate pull requests is essential. Unfortunately, it's quite easy to make mistakes or not know what you should do when you're initially learning the process. I know that I certainly had considerable initial trouble with it, and I found a lot of the information on GitHub and around the internet to be rather piecemeal and incomplete - part of the process described here, another there, common hangups in a different place, and so on.

In an attempt to coallate this information for myself and others, this short tutorial is what I've found to be fairly standard procedure for creating a fork, doing your work, issuing a pull request, and merging that pull request back into the original project.

26.2.1 Creating a Fork

Just head over to the GitHub page and click the "Fork" button. It's just that simple. Once you've done that, you can use your favorite git client to clone your repo or just head straight to the command line:

```
# Clone your fork to your local machine
git clone git@github.com:USERNAME/FORKED-PROJECT.git
```

26.2.2 Keeping Your Fork Up to Date

While this isn't an absolutely necessary step, if you plan on doing anything more than just a tiny quick fix, you'll want to make sure you keep your fork up to date by tracking the original "upstream" repo that you forked. To do this, you'll need to add a remote:

```
# Add 'upstream' repo to list of remotes
git remote add upstream https://github.com/UPSTREAM-USER/ORIGINAL-PROJECT.git

# Verify the new remote named 'upstream'
git remote -v
```

Whenever you want to update your fork with the latest upstream changes, you'll need to first fetch the upstream repo's branches and latest commits to bring them into your repository:

```
# Fetch from upstream remote
git fetch upstream

# View all branches, including those from upstream
git branch -va
```

Now, checkout your own master branch and merge the upstream repo's master branch:

```
# Checkout your master branch and merge upstream
git checkout master
git merge upstream/master
```

If there are no unique commits on the local master branch, git will simply perform a fast-forward. However, if you have been making changes on master (in the vast majority of cases you probably shouldn't be), you may have to deal with conflicts. When doing so, be careful to respect the changes made upstream.

Now, your local master branch is up-to-date with everything modified upstream.

26.2.3 Doing Your Work

Create a Branch

Whenever you begin work on a new feature or bugfix, it's important that you create a new branch. Not only is it proper git workflow, but it also keeps your changes organized and separated from the master branch so that you can easily submit and manage multiple pull requests for every task you complete.

To create a new branch and start working on it:

```
# Checkout the master branch - you want your new branch to come from master
git checkout master

# Create a new branch named newfeature (give your branch its own simple informative␣
↪name)
git branch newfeature

# Switch to your new branch
git checkout newfeature
```

Now, go to town hacking away and making whatever changes you want to.

26.2.4 Submitting a Pull Request

Cleaning Up Your Work

Prior to submitting your pull request, you might want to do a few things to clean up your branch and make it as simple as possible for the original repo's maintainer to test, accept, and merge your work.

If any commits have been made to the upstream master branch, you should rebase your development branch so that merging it will be a simple fast-forward that won't require any conflict resolution work.

```
# Fetch upstream master and merge with your repo's master branch
git fetch upstream
git checkout master
git merge upstream/master

# If there were any new commits, rebase your development branch
git checkout newfeature
git rebase master
```

Now, it may be desirable to squash some of your smaller commits down into a small number of larger more cohesive commits. You can do this with an interactive rebase:

```
# Rebase all commits on your development branch
git checkout
git rebase -i master
```

This will open up a text editor where you can specify which commits to squash.

Submitting

Once you've committed and pushed all of your changes to GitHub, go to the page for your fork on GitHub, select your development branch, and click the pull request button. If you need to make any adjustments to your pull request, just push the updates to GitHub. Your pull request will automatically track the changes on your development branch and update.

26.2.5 Accepting and Merging a Pull Request

Take note that unlike the previous sections which were written from the perspective of someone that created a fork and generated a pull request, this section is written from the perspective of the original repository owner who is handling an incoming pull request. Thus, where the "forker" was referring to the original repository as *upstream*, we're now looking at it as the owner of that original repository and the standard *origin* remote.

Checking Out and Testing Pull Requests

Open up the `.git/config` file and add a new line under `[remote "origin"]`:

```
fetch = +refs/pull/*/head:refs/pull/origin/*
```

Now you can fetch and checkout any pull request so that you can test them:

```
# Fetch all pull request branches
git fetch origin

# Checkout out a given pull request branch based on its number
git checkout -b 999 pull/origin/999
```

Keep in mind that these branches will be read only and you won't be able to push any changes.

Automatically Merging a Pull Request

In cases where the merge would be a simple fast-forward, you can automatically do the merge by just clicking the button on the pull request page on GitHub.

Manually Merging a Pull Request

To do the merge manually, you'll need to checkout the target branch in the source repo, pull directly from the fork, and then merge and push.

```
# Checkout the branch you're merging to in the target repo
git checkout master

# Pull the development branch from the fork repo where the pull request development␣
↪was done.
git pull https://github.com/forkuser/forkedrepo.git newfeature

# Merge the development branch
git merge newfeature

# Push master with the new feature merged into it
git push origin master
```

Now that you're done with the development branch, you're free to delete it.

```shell git branch -d newfeature ```

26.3 Creating A New Repository

For some contributions it may be required to create a new Curve repository. The Curve repositories aim to employ a consistent code style. In order to make new repositories adhere to this style, there exists a *Curve repository template*, which should be used.

The template repository can be found here. This template already contains dependencies and formatting rules in line with the Curve style guidelines.

Copyright

Additional Reading

- Atlassian - Merging vs. Rebasing

Sources

- GitHub - Fork a Repo
- GitHub - Syncing a Fork
- GitHub - Checking Out a Pull Request
```

CHAPTER

TWENTYSEVEN

TESTING

Curve development follows a strong testing methodology. While testing Ethereum-based protocols can be challenging, the Curve test suite is a powerful tool that shall be used by contributors to help facilitate this task. While the repositories `curve-contract`, `curve-dao-contracts` and `curve-pool-registry` are all stand alone repositories where each repo employs its own test suite, the test suite designs are very similar.

This section outlines how the test suite should be used most effectively for the `curve-contracts` repository.

27.1 Curve Contracts

Test cases for Curve pools are organized across the following subdirectories:

- `forked`: Tests designed for use in a forked mainnet
- `fixtures`: Pytest fixtures
- `pools`: Tests for pool contracts
- `token`: Tests for LP token contracts
- `zaps`: Tests for deposit contracts

Other files:

- conftest.py: Base configuration file for the test suite.
- simulation.py: A python model of the math used within Curve's contracts. Used for testing expected outcomes with actual results.

27.1.1 Organization

- Tests are organized by general category, then split between unitary and integration tests.
- Common tests for all pools are located in `tests/pools/common`, for zaps in `tests/zaps/common`.
- Common metapool tests are located at `tests/pools/meta`, for zaps in `tests/zaps/meta`.
- Valid pool names are the names of the subdirectories within `contracts/pools`.
- For pool templates, prepend `template-` to the subdirectory names within `contracts/pool-templates`. For example, the base template is `template-base`.

Pool Type Tests

Note that the test suite targets tests also on a *pool type* basis. A Curve pool may be of one or more types. The supported pool types are:

- `arate`: These are `aToken`-style pools (interest accrues as balance increases)
- `crate`: These are `cToken`-style pools (interest accrues as rate increases)
- `eth`: These are pools that have `ETH` as one of their tokens
- `meta`: These are metapools

An example of a pool of a single type would be the `aave` pool, which is of type `arate`.

An example of a pool of multiple types would be the `steth` pool, which is of the types `eth` and `arate`.

The type of a pool is given by the key value pair `"pool_types": [<POOL_TYPE>, ...]` in a pool's `pooldata.json` file. If no type is specified, the pool is by default a `template-base`-style pool.When running tests, the test suit targets pool type-specific tests if they exist. To add a pool type-specific test, place the new test into the *pool type subdirectory* (e.g., *meta* for metapool tests).

Pool-specific Tests

There may be pools for which it is required to write multiple tests, which are not applicable to other pools. Rather than using decorators to *skip* (see below) other pools on an individual or type basis, a new subdirectory **named after the pool** can be created to contain the pool-specific tests.

When the test suite is started, for a given pool, all tests for the pool's type get run, as well as any existing pool-specific tests.

For example, assuming there exists a new metapool called `foo`, specifying `"pool_types": ["meta"]` in the pool's `pooldata.json` would ensure that all metapool tests get run. Let's assume there is a token in the pool, which has behavior that is currently not captured by any of the `meta` or `common` tests that get currently run for the `foo` pool. To ensure we test the `foo` pool's behavior thoroughly, new tests should be created and added in a newly created `tests/pools/foo/` subdirectory.

27.1.2 Running the tests

To run the entire suite:

```
brownie test
```

Note that this executes over 10,000 tests and may take a significant amount of time to finish.

Test Collection Filters

The test suite is divided into several logical categories. Tests may be filtered using one or more flags:

- `--pool <POOL NAME>`: only run tests against a specific pool
- `--integration`: only run integration tests (tests within an `integration/` subdirectory)
- `--unitary`: only run unit tests (tests NOT found in an `integration/` subdirectory)

For example, to only run the unit tests for 3pool:

```
brownie test --pool 3pool --unitary
```

Testing against a forked mainnet

To run the test suite against a forked mainnet:

```
brownie test --network mainnet-fork
```

In this mode, the actual underlying and wrapped coins are used for testing. Note that forked mode can be *very slow*, especially if you are running against a public node.

27.1.3 Fixtures

Test fixtures are located within the tests/fixture subdirectory. New fixtures should be added here instead of within the base conftest.py.

All fixtures are [documented](fixtures/README.md) within the fixtures subdirectory readme.

27.1.4 Markers

We use the following custom markers to parametrize common tests across different pools:

skip_pool(*pools)

Exclude one or more pools from the given test.

```
@pytest.mark.skip_pool("compound", "usdt", "y")
def test_only_some_pools(swap):
    ...
```

skip_pool_type(*pool_types)

Exclude specific pool types from the given test.

```
@pytest.mark.skip_pool_type("meta", "eth")
def test_not_metapools(swap):
    ...
```

target_pool(*pools)

Only run the given test against one or more pools specified in the marker.

```
@pytest.mark.target_pool("ren", "sbtc")
def test_btc_pools(swap):
    ...
```

skip_meta

Exclude metapools from the given test.

```
@pytest.mark.skip_meta
def test_not_metapools(swap):
    ...
```

lending

Only run the given test against pools that involve lending.

```
@pytest.mark.lending
def test_underlying(swap):
    ...
```

zap

Only run the given test against pools that use a deposit contract.

```
@pytest.mark.zap
def test_deposits(zap):
    ...
```

itercoins(*arg, underlying=False)

Parametrizes each of the given arguments with a range of numbers equal to the total number of coins for the given pool. When multiple arguments are given, each argument has a unique value for every generated test.

For example, `itercoins("send", "recv")` with a pool of 3 coins will parametrize with the sequence `[(0, 1), (0, 2), (1, 0), (1, 2), (2, 0), (2, 1)]`.

If `underlying` is set as `True`, the upper bound of iteration corresponds to the true number of underlying coins. This is useful when testing metapools.

```
@pytest.mark.itercoins("send", "recv")
def test_swap(accounts, swap, send, recv):
    swap.exchange(send, recv, 0, 0, {'from': accounts[0]})
```

CHAPTER

TWENTYEIGHT

CODE STYLE

28.1 Vyper Style Guide

This document outlines the Vyper code style, structure and practices followed by the Curve development team.

Note that this guide is still under development. Do not hesitate to ask if anything is missing or unclear.

28.1.1 Project Organization

Contracts should be structured so that components are logically grouped together. Maintaining a consistent order makes it easier for the reader to locate code.

Each logical section should be separated by two blank lines. Within each section, multi-line statements should be seperated by one blank line. Single-line statements should have no blank lines between them, except to denote a logical seperation.

Content should be ordered as follows:

1. Import statements
2. Implements statements
3. Inlined interfaces
4. Events
5. Structs
6. Constants
7. Storage variables
8. Constructor function
9. Fallback function
10. Regular functions

Imports and Interfaces

Contracts **must** be self contained. Import statements may only be used for built-in interfaces. Other interfaces are always inlined. This aids readability and simplifies the process of source verification on Etherscan.

Inlined interfaces should only include required functions (those that are called within the contract). Interfaces and the functions within should be sorted alphabetically. Each interface should be seperated by one blank line.

Events and Structs

Events and structs should be sorted alphabetically. Each definition should be seperated by one blank line, with two blank lines between the last event and the first struct.

Constants and Storage Variables

Constants should always be defined before storage variables, except when there is a logical reason to group them otherwise. Variable definitions should not be seperated by blank lines, but a single blank line can be used to create logical groupings.

Functions

The constructor function must always be first, followed by the fallback function (if the contract includes one). Regular functions should be logically grouped. Each function should be seperated by two blank lines.

28.1.2 Naming Conventions

Names adhere to PEP 8 naming conventions:

- **Events**, **interfaces** and **structs** use the CapWords convention.
- **Function** names are lowercase, with words separated by underscores when it improves readability. The only exception when adhering to a common interface such as ERC20.
- **Constants** use all capital letters with underscores separating words.
- **Variables** follow the same conventions as functions.

Leading Underscores

A single leading underscore marks an object as private or immutable.

- For functions, a leading underscore indicates that the function is internal.
- For variables, a leading underscore is used to indicate that the variable exists in calldata.

Booleans

- Boolean values **should** be prefixed with `is_`.
- Booleans **must not** represent *negative* properties, (e.g. `is_not_set`). This can result in double-negative evaluations which are not intuitive for readers.

28.1.3 Code Style

As Vyper is syntactically similar to Python, all code **should** conform to the PEP 8 style guide with the following exceptions:

- Maximum line length of 100

In general, we try to mimick the same linting process as would be applied by black if the code were Python.

Decorators

Function decorators **should** be ordered according to mutability, visibility, re-entrancy:

```
@view
@external
@nonreentrant('lock')
def foo():
```

Function Inputs

All input variables **should** be prepended with a single leading underscore to denote their immutability. The only exception is if the variable name is a single letter (such as *i* and *j* for the swap contract exchange methods).

Where possible, the entire function signature should be kept on a single line:

```
def foo(_goo: address, _bar: uint256, _baz: uint256) -> bool:
```

If this line would exceed 100 characters, each input argument should instead be placed on a new line and indented:

```
def multiline_foo(
    _goo: address,
    _bar: uint256,
    _baz: uint256,
) -> bool:
```

Revert Strings

Revert strings **must not** exceed a maximum length of 32 characters. They should only be used in functions that are expected to be frequently called by average users. For other situations you **should** use a dev revert comment.

CHAPTER

TWENTYNINE

PYTHON STYLE GUIDE

This document outlines the Python code style, structure and practices followed by the Curve development team.

Note that this guide is still under development. Do not hesitate to ask if anything is missing or unclear.

29.1 Linting and Pre-Commit Hooks

We use pre-commit hooks to simplify linting and ensure consistent formatting among contributors. Use of `pre-commit` is not a requirement, but is highly recommended.

Install `pre-commit` locally from the brownie root folder:

```
pip install pre-commit
pre-commit install
```

Commiting will now automatically run the local hooks and ensure that your commit passes all lint checks.

29.1.1 Naming Conventions

Names **must** adhere to PEP 8 naming conventions:

- **Modules** have short, all-lowercase names. Underscores can be used in the module name if it improves readability.
- **Class names** use the CapWords convention.
- **Exceptions** follow the same conventions as other classes.
- **Function** names are lowercase, with words separated by underscores when it improves readability.
- **Method** names and **instance** variables follow the same conventions as functions.
- **Constants** use all capital letters with underscores separating words.

Leading Underscores

A single leading underscore marks an object as private.

- Classes and functions with one leading underscore are only used in the module where they are declared. They **must not** be imported.
- Class attributes and methods with one leading underscore **must** only be accessed by methods within the same class.

Booleans

- Boolean values **should** be prefixed with `is_`.
- Booleans **must not** represent *negative* properties, (e.g. `is_not_set`). This can result in double-negative evaluations which are not intuitive for readers.
- Methods that return a single boolean **should** use the *@property* decorator.

Methods

The following conventions **should** be used when naming functions or methods. Consistent naming provides logical consistency throughout the codebase and makes it easier for future readers to understand what a method does (and does not) do.

- `get_`: For simple data retrieval without any side effects.
- `fetch_`: For retreivals that may have some sort of side effect.
- `build_`: For creation of a new object that is derived from some other data.
- `set_`: For adding a new value or modifying an existing one within an object.
- `add_`: For adding a new attribute or other value to an object. Raises an exception if the value already exists.
- `replace_`: For mutating an object. Should return `None` on success or raise an exception if something is wrong.
- `compare_`: For comparing values. Returns `True` or `False`, does not raise an exception.
- `validate_`: Returns `None` or raises an exception if something is wrong.
- `from_`: For class methods that instantiate an object based on the given input data.

For other functionality, choose names that clearly communicate intent without being overly verbose. Focus on *what* the method does, not on *how* the method does it.

29.1.2 Code Style

All code **must** conform to the PEP 8 style guide with the following exceptions:

- Maximum line length of 100

We handle code formatting with black. This ensures a consistent style across the project and saves time by not having to be opinionated.

Imports

Import sequencing is handled with isort. We follow these additional rules:

Standard Library Imports

Standard libraries **should** be imported absolutely and without aliasing. Importing the library aids readability, as other users may be familiar with that library.

```
# Good
import os
os.stat('.')

# Bad
from os import stat
stat('.')
```

Strings

Strings substitutions **should** be performed via formatted string literals rather than the *str.format* method or other techniques.

CHAPTER

THIRTY

DEPLOYMENT ADDRESSES

Here is a list of all current contract deployments within the Curve protocol.

Note: If you find an address which is missing or incorrect, feel free to create a pull request as specified here.

30.1 Base Pools

Base pools in Curve contain two or more tokens and implement the Curve stable swap exchange mechanism. Note that for a single base or meta pool there are multiple deployed contracts, which are of the following formats:

- `StableSwap<pool>.vy`: Curve stablecoin AMM contract
- `Deposit<pool>.vy`: contract used to wrap underlying tokens prior to depositing them into the pool (not always required)
- `CurveContract<version>.vy`: LP token contract for the pool

Here is a list of all base pool contracts currently in use:

Pool	Source	Address
3Pool	StableSwap3Pool.vy	0xbEbc44782C7dB0a1A60Cb6fe97d0b483032FF1C7
3Pool	CurveTokenV2.vy	0x6c3F90f043a72FA612cbac8115EE7e52BDe6E490
AAVE	CurveTokenV3.vy	0xFd2a8fA60Abd58Efe3EeE34dd494cD491dC14900
AAVE	StableSwapAave.vy	0xDeBF20617708857ebe4F679508E7b7863a8A8EeE
ankrETH	StableSwapAETH.vy	0xA96A65c051bF88B4095Ee1f2451C2A9d43F53Ae2
ankrETH	CurveTokenV3.vy	0xaA17A236F2bAdc98DDc0Cf999AbB47D47Fc0A6Cf
BUSD	StableSwapBUSD.vy	0x79a8C46DeA5aDa233ABaFFD40F3A0A2B1e5A4F27
BUSD	DepositBUSD.vy	0xb6c057591E073249F2D9D88Ba59a46CFC9B59EdB
BUSD	CurveTokenV1.vy	0x3B3Ac5386837Dc563660FB6a0937DFAa5924333B
Compound	StableSwapCompound.vy	0xA2B47E3D5c44877cca798226B7B8118F9BFb7A56
Compound	DepositCompound.vy	0xeB21209ae4C2c9FF2a86ACA31E123764A3B6Bc06
Compound	CurveContractV1.vy	0x845838DF265Dcd2c412A1Dc9e959c7d08537f8a2
EURS	StableSwapEURS.vy	0x0Ce6a5fF5217e38315f87032CF90686C96627CAA
EURS	CurveTokenV3.vy	0x194eBd173F6cDacE046C53eACcE9B953F28411d1
hBTC	StableSwapHBTC.vy	0x4CA9b3063Ec5866A4B82E437059D2C43d1be596F
hBTC	CurveTokenV2.vy	0xb19059ebb43466C323583928285a49f558E572Fd
IronBank	StableSwapIB.vy	0x2dded6Da1BF5DBdF597C45fcFaa3194e53EcfeAF
IronBank	CurveTokenV3.vy	0x5282a4eF67D9C33135340fB3289cc1711c13638C
Link	StableSwapLINK.vy	0xf178c0b5bb7e7abf4e12a4838c7b7c5ba2c623c0

continues on next page

Table 1 – continued from previous page

Pool	Source	Address
Link	CurveTokenV3.vy	0xcee60cfa923170e4f8204ae08b4fa6a3f5656f3a
PAX	DepositPax.vy	0xA50cCc70b6a011CffDdf45057E39679379187287
PAX	StableSwapPax.vy	0x06364f10B501e868329afBc005b3492902d6C763
PAX	CurveTokenV1.vy	0xD905e2eaeBe188fc92179b6350807D8bd91Db0D8
renBTC	StableSwapRen.vy	0x93054188d876f558f4a66B2EF1d97d16eDf0895B
renBTC	CurveTokenV1.vy	0x49849C98ae39Fff122806C06791Fa73784FB3675
rETH	StableSwapRETH.vy	0xF9440930043eb3997fc70e1339dBb11F341de7A8
rETH	CurveTokenV3.vy	0x53a901d48795C58f485cBB38df08FA96a24669D5
sAAVE	StableSwapSAAVE.vy	0xEB16Ae0052ed37f479f7fe63849198Df1765a733
sAAVE	CurveTokenV3.vy	0x02d341CcB60fAaf662bC0554d13778015d1b285C
sBTC	StableSwapSBTC.vy	0x7fC77b5c7614E1533320Ea6DDc2Eb61fa00A9714
sBTC	CurveTokenV1.vy	0x075b1bb99792c9E1041bA13afEf80C91a1e70fB3
sETH	StableSwapSETH.vy	0xc5424B857f758E906013F3555Dad202e4bdB4567
sETH	CurveTokenV3.vy	0xA3D87FffcE63B53E0d54fAa1cc983B7eB0b74A9c
stETH	StableSwapSTETH.vy	0xDC24316b9AE028F1497c275EB9192a3Ea0f67022
stETH	CurveTokenV3.vy	0x06325440D014e39736583c165C2963BA99fAf14E
sUSD	DepositSUSD.vy	0xFCBa3E75865d2d561BE8D220616520c171F12851
sUSD	StableSwapSUSD.vy	0xA5407eAE9Ba41422680e2e00537571bcC53efBfD
sUSD	CurveTokenV1.vy	0xC25a3A3b969415c80451098fa907EC722572917F
TriCrypto	CurveCryptoSwap.vy	0x80466c64868E1ab14a1Ddf27A676C3fcBE638Fe5
TriCrypto	DepositZap.vy	0x331aF2E331bd619DefAa5DAc6c038f53FCF9F785
TriCrypto	CurveTokenV4.vy	0xcA3d75aC011BF5aD07a98d02f18225F9bD9A6BDF
USDT	DepositUSDT.vy	0xac795D2c97e60DF6a99ff1c814727302fD747a80
USDT	StableSwapUSDT.vy	0x52EA46506B9CC5Ef470C5bf89f17Dc28bB35D85C
USDT	CurveTokenV1.vy	0x9fC689CCaDa600B6DF723D9E47D84d76664a1F23
Y	DepositY.vy	0xbBC81d23Ea2c3ec7e56D39296F0cbB648873a5d3
Y	StableSwapY.vy	0x45F783CCE6B7FF23B2ab2D70e416cdb7D6055f51
Y	CurveTokenV1.vy	0xdF5e0e81Dff6FAF3A7e52BA697820c5e32D806A8
Yv2	StableSwapYv2.vy	0x8925D9d9B4569D737a48499DeF3f67BaA5a144b9
Yv2	CurveTokenV3.vy	0x571FF5b7b346F706aa48d696a9a4a288e9Bb4091

30.2 MetaPools

Metapools allow for one token to seemingly trade with another underlying base pool. For instance, the GUSD metapool (`[GUSD, [3Pool]]`) contains GUSD and LP tokens of the 3pool (3CRV). This allows for trades between GUSD and any of the three tokens from the 3Pool (DAI, USDC and USDT).

Here is a list of all meta pools currently in use:

Pool	Source	Address
bBTC	StableSwapBBTC.vy	0x071c661B4DeefB59E2a3DdB20Db036821eeE8F4b
bBTC	DepositBBTC.vy	0xC45b2EEe6e09cA176Ca3bB5f7eEe7C47bF93c756
bBTC	CurveTokenV3.vy	0x410e3E86ef427e30B9235497143881f717d93c2A
DUSD	DepositDUSD.vy	0x61E10659fe3aa93d036d099405224E4Ac24996d0
DUSD	StableSwapDUSD.vy	0x8038C01A0390a8c547446a0b2c18fc9aEFEcc10c
DUSD	CurveTokenV2.vy	0x3a664Ab939FD8482048609f652f9a0B0677337B9
GUSD	StableSwapGUSD.vy	0x4f062658EaAF2C1ccf8C8e36D6824CDf41167956
GUSD	DepositGUSD.vy	0x64448B78561690B70E17CBE8029a3e5c1bB7136e

continues on next page

Table 2 – continued from previous page

Pool	Source	Address
GUSD	CurveTokenV2.vy	0xD2967f45c4f384DEEa880F807Be904762a3DeA07
HUSD	DepositHUSD.vy	0x09672362833d8f703D5395ef3252D4Bfa51c15ca
HUSD	StableSwapHUSD.vy	0x3eF6A01A0f81D6046290f3e2A8c5b843e738E604
HUSD	CurveTokenV2.vy	0x5B5CFE992AdAC0C9D48E05854B2d91C73a003858
LinkUSD	DepositLinkUSD.vy	0x1de7f0866e2c4adAC7b457c58Cc25c8688CDa1f2
LinkUSD	StableSwapLinkUSD.vy	0xE7a24EF0C5e95Ffb0f6684b813A78F2a3AD7D171
LinkUSD	CurveTokenV2.vy	0x6D65b498cb23deAba52db31c93Da9BFFb340FB8F
MUSD	DepositMUSD.vy	0x803A2B40c5a9BB2B86DD630B274Fa2A9202874C2
MUSD	StableSwapMUSD.vy	0x8474DdbE98F5aA3179B3B3F5942D724aFcdec9f6
MUSD	CurveTokenV2.vy	0x1AEf73d49Dedc4b1778d0706583995958Dc862e6
oBTC	DepositOBTC.vy	0xd5BCf53e2C81e1991570f33Fa881c49EEa570C8D
oBTC	StableSwapOBTC.vy	0xd81dA8D904b52208541Bade1bD6595D8a251F8dd
oBTC	CurveTokenV3.vy	0x2fE94ea3d5d4a175184081439753DE15AeF9d614
pBTC	DepositPBTC.vy	0x11F419AdAbbFF8d595E7d5b223eee3863Bb3902C
pBTC	StableSwapPBTC.vy	0x7F55DDe206dbAD629C080068923b36fe9D6bDBeF
pBTC	CurveTokenV2.vy	0xDE5331AC4B3630f94853Ff322B66407e0D6331E8
RSV	DepositRSV.vy	0xBE175115BF33E12348ff77CcfEE4726866A0Fbd5
RSV	StableSwapRSV.vy	0xC18cC39da8b11dA8c3541C598eE022258F9744da
RSV	CurveTokenV2.vy	0xC2Ee6b0334C261ED60C72f6054450b61B8f18E35
tBTC	DepositTBTC.vy	0xaa82ca713D94bBA7A89CEAB55314F9EfFEdDc78c
tBTC	StableSwapTBTC.vy	0xC25099792E9349C7DD09759744ea681C7de2cb66
tBTC	CurveTokenV2.vy	0x64eda51d3Ad40D56b9dFc5554E06F94e1Dd786Fd
USDK	DepositUSDK.vy	0xF1f85a74AD6c64315F85af52d3d46bF715236ADc
USDK	StableSwapUSDK.vy	0x3E01dD8a5E1fb3481F0F589056b428Fc308AF0Fb
USDK	CurveTokenV2.vy	0x97E2768e8E73511cA874545DC5Ff8067eB19B787
USDN	DepositUSDN.vy	0x094d12e5b541784701FD8d65F11fc0598FBC6332
USDN	StableSwapUSDN.vy	0x0f9cb53Ebe405d49A0bbdBD291A65Ff571bC83e1
USDN	CurveTokenV2.vy	0x4f3E8F405CF5aFC05D68142F3783bDfE13811522
USDP	DepositUSDP.vy	0x3c8cAee4E09296800f8D29A68Fa3837e2dae4940
USDP	StableSwapUSDP.vy	0x42d7025938bEc20B69cBae5A77421082407f053A
USDP	CurveTokenV3.vy	0x7Eb40E450b9655f4B3cC4259BCC731c63ff55ae6
UST	DepositUST.vy	0xB0a0716841F2Fc03fbA72A891B8Bb13584F52F2d
UST	StableSwapUST.vy	0x890f4e345B1dAED0367A877a1612f86A1f86985f
UST	CurveTokenV3.vy	0x94e131324b6054c0D789b190b2dAC504e4361b53

30.3 Liquidity Gauges

Liquidity Gauges are used to stake LP tokens and handle distribution of the CRV governance token and are part of the Curve DAO.

Here is a list of all liquidity gauges currently in use:

Gauge	Source	Address
3pool	LiquidityGauge.sol	0xbFcF63294aD7105dEa65aA58F8AE5BE2D9d0952A
AAVE	LiquidityGaugeV2.vy	0xd662908ADA2Ea1916B3318327A97eB18aD588b5d
alUSD	LiquidityGaugeV3.vy	0x9582C4ADACB3BCE56Fea3e590F05c3ca2fb9C477
ankrETH	LiquidityGaugeV2.vy	0x6d10ed2cF043E6fcf51A0e7b4C2Af3Fa06695707
bBTC	LiquidityGaugeV2.vy	0xdFc7AdFa664b08767b735dE28f9E84cd30492aeE

continues on next page

Table 3 – continued from previous page

Gauge	Source	Address
BUSD	LiquidityGauge.vy	0x69Fb7c45726cfE2baDeE8317005d3F94bE838840
Compound	LiquidityGauge.sol	0x7ca5b0a2910B33e9759DC7dDB0413949071D7575
DUSD	LiquidityGaugeReward.vy	0xAEA6c312f4b3E04D752946d329693F7293bC2e6D
EURS	LiquidityGaugeV2.vy	0x90Bb609649E0451E5aD952683D64BD2d1f245840
FRAX	LiquidityGaugeV2.vy	0x72e158d38dbd50a483501c24f792bdaaa3e7d55c
GUSD	LiquidityGauge.vy	0xC5cfaDA84E902aD92DD40194f0883ad49639b023
hBTC	LiquidityGauge.vy	0x4c18E409Dc8619bFb6a1cB56D114C3f592E0aE79
HUSD	LiquidityGauge.vy	0x2db0E83599a91b508Ac268a6197b8B14F5e72840
MUSD	LiquidityGaugeReward.vy	0x5f626c30EC1215f4EdCc9982265E8b1F411D1352
oBTC	LiquidityGaugeV2.vy	0x11137B10C210b579405c21A07489e28F3c040AB1
PAX	LiquidityGauge.vy	0x64E3C23bfc40722d3B649844055F1D51c1ac041d
IronBank	LiquidityGaugeV2.vy	0xF5194c3325202F456c95c1Cf0cA36f8475C1949F
Link	LiquidityGaugeV2.vy	0xFD4D8a17df4C27c1dD245d153ccf4499e806C87D
pBTC	LiquidityGaugeV2.vy	0xd7d147c6Bb90A718c3De8C0568F9B560C79fa416
renBTC	LiquidityGauge.vy	0xB1F2cdeC61db658F091671F5f199635aEF202CAC
RSV	LiquidityGaugeReward.vy	0x4dC4A289a8E33600D8bD4cf5F6313E43a37adec7
sAAVE	LiquidityGaugeV2.vy	0x462253b8F74B72304c145DB0e4Eebd326B22ca39
sBTC	LiquidityGaugeReward.vy	0x705350c4BcD35c9441419DdD5d2f097d7a55410F
sETH	LiquidityGaugeV2.vy	0x3C0FFFF15EA30C35d7A85B85c0782D6c94e1d238
stETH	LiquidityGaugeV2.vy	0x182B723a58739a9c974cFDB385ceaDb237453c28
sUSDv2	LiquidityGaugeReward.vy	0xA90996896660DEcC6E997655E065b23788857849
rETH	LiquidityGaugeV3.vy	0x824F13f1a2F29cFEEa81154b46C0fc820677A637
tBTC	LiquidityGaugeReward.vy	0x6828bcF74279eE32f2723eC536c22c51Eed383C6
TriCrypto	LiquidityGaugeV3.vy	0x6955a55416a06839309018A8B0cB72c4DDC11f15
USDK	LiquidityGauge.vy	0xC2b1DF84112619D190193E48148000e3990Bf627
USDN	LiquidityGauge.vy	0xF98450B5602fa59CC66e1379DFfB6FDDc724CfC4
USDP	LiquidityGaugeV2.vy	0x055be5DDB7A925BfEF3417FC157f53CA77cA7222
USDT	LiquidityGauge.vy	0xBC89cd85491d81C6AD2954E6d0362Ee29fCa8F53
UST	LiquidityGaugeV2.vy	0x3B7020743Bc2A4ca9EaF9D0722d42E20d6935855
Y	LiquidityGauge.vy	0xFA712EE4788C042e2B7BB55E6cb8ec569C4530c1
Yv2	LiquidityGaugeV2.vy	0x8101E6760130be2C8Ace79643AB73500571b7162

30.4 Curve DAO

Curve DAO consists of multiple smart contracts connected by Aragon. Interaction with Aragon occurs through a modified implementation of the Aragon Voting App. Aragon's standard one token, one vote method is replaced with a weighting system based on locking tokens. Curve DAO has a token (CRV) which is used for both governance and value accrual.

View the documentation for an in-depth overview of how the Curve DAO works.

Here is a list of contract deployments that are used in the Curve DAO:

Name	Source	Address
CRV Token	ERC20CRV.sol	0xD533a949740bb3306d119CC777fa900bA034cd52
Fee Distributor	FeeDistributor.vy	0xA464e6DCda8AC41e03616F95f4BC98a13b8922Dc
Gauge Controller	GaugeController.vy	0x2F50D538606Fa9EDD2B11E2446BEb18C9D5846bB
Minter	Minter.vy	0xd061D61a4d941c39E5453435B6345Dc261C2fcE0
Voting Escrow	VotingEscrow.vy	0x5f3b5DfEb7B28CDbD7FAba78963EE202a494e2A2
Vesting Escrow	VestingEscrow.vy	0x575ccd8e2d300e2377b43478339e364000318e2c

30.4.1 Ownership Proxies

The following contracts allow DAO ownership of the core Curve AMM contracts:

Name	Source	Address
Gauge Owner	GaugeProxy.vy	0x519AFB566c05E00cfB9af73496D00217A630e4D5
Pool Owner	PoolProxy.vy	0xeCb456EA5365865EbAb8a2661B0c503410e9B347
Crypto Pool Owner	CryptoPoolProxy.vy	0x3687367CcAEBBE89f1bc8Eae7592b4Eed44Ac0Bd
Factory Pool Owner	OwnerProxy.vy	0x8cf8af108b3b46ddc6ad596aebb917e053f0d72b

30.4.2 Aragon

Main documentation: *Curve DAO: Governance*

Voting App

Aragon Voting App deployments are the main entry points used to create new votes, vote, checking the status of a vote, and execute a successful vote.

Vote Type	Address
Ownership	0xE478de485ad2fe566d49342Cbd03E49ed7DB3356
Parameter	0xBCfF8B0b9419b9A88c44546519b1e909cF330399
Emergency	0x1115c9b3168563354137cDc60efb66552dd50678

Agent

Aragon Agent deployments correspond to the different owner accounts within the DAO. Contract calls made as a result of a successful vote will execute from these addresses. When deploying new contracts, these addresses should be given appropriate access to admin functionality.

Vote Type	Address
Ownership	0x40907540d8a6c65c637785e8f8b742ae6b0b9968
Parameter	0x4eeb3ba4f221ca16ed4a0cc7254e2e32df948c5f
Emergency	0x00669DF67E4827FCc0E48A1838a8d5AB79281909

Tokens

The following token addresses are used for determining voter weights within Curve's Aragon DAOs.

Vote Type	Address
Ownership / Parameter	0x5f3b5DfEb7B28CDbD7FAba78963EE202a494e2A2
Emergency	0x4c0947B16FB1f755A2D32EC21A0c4181f711C500

30.4.3 Fee Burners

Burners are a fundamental component of the fee payout mechanism in Curve. A burner converts collected pool fees to an asset which can be converted to USDC. Ultimately, the exchanged for USDC is deposited to the 3Pool, as fees are paid out in 3CRV to veCRV holders. Depending on which tokens a pool contains, a specific burner implementation is used.

Here is a list of all burner contracts currently in use:

Gauge	Source	Address
ABurner	ABurner.vy	0x12220a63a2013133d54558c9d03c35288eac9b34
CryptoLPBurner	CryptoLPBurner.vy	0x0B5B9210d5015fD0c97FB19B32675b19703b0453
CBurner	CBurner.vy	0xdd0e10857d952c73b2fa39ce86308299df8774b8
LPBurner	LPBurner.vy	0xaa42C0CD9645A58dfeB699cCAeFBD30f19B1ff81
MetaBurner	MetaBurner.vy	0xE4b65889469ad896e866331f0AB5652C1EcfB3E6
SynthBurner	SynthBurner.vy	0x67a0213310202DBc2cbE788f4349B72fbA90f9Fa
USDNBurner	USDNBurner.vy	0x06534b0BF7Ff378F162d4F348390BDA53b15fA35
UnderlyingBurner	UnderlyingBurner.vy	0x786b374b5eef874279f4b7b4de16940e57301a58
UniswapBurner	UniswapBurner.vy	0xf3b64840b39121b40d8685f1576b64c157ce2e24
YBurner	YBurner.vy	0xd16ea3e5681234da84419512eb597362135cd8c9

30.5 Pool Registry

The pool registry serves as an on-chain information hub about the current state of Curve pools. For instance, on-chain integrators can fetch the current address of a Curve pool and query information about it.

Here is a list of all components of the pool registry currently in use:

Name	Source	Address
Address Provider	AddressProvider.vy	0x0000000022d53366457f9d5e68ec105046fc4383
Curve Calculator	CurveCalc.vy	0xc1DB00a8E5Ef7bfa476395cdbcc98235477cDE4E
Pool Info	PoolInfo.vy	0xe64608E223433E8a03a1DaaeFD8Cb638C14B552C
Registry	Registry.vy	0x90E00ACe148ca3b23Ac1bC8C240C2a7Dd9c2d7f5

30.6 MetaPool Factory

The metapool factory allows for the permissionless deployment of Curve metapools. As discussed *here*, the metapool factory has the following core components:

- The *factory* is the main contract used to deploy new metapools. It also acts a registry for finding the deployed pools and querying information about them.
- *Pools* are deployed via a proxy contract. The implementation contract targetted by the proxy is determined according to the base pool. This is the same technique used to create pools in Uniswap V1.
- *Deposit contracts* ("zaps") are used for wrapping and unwrapping underlying assets when depositing into or withdrawing from pools.

Name	Source	Address
Factory	Factory.vy	0x0959158b6040D32d04c301A72CBFD6b39E21c9AE
Migrator	PoolMigrator.vy	0xd6930b7f661257DA36F93160149b031735237594

30.6.1 Implementation Contracts

The implementation contracts used for factory metapools are deployed to the mainnet at the following addresses:

Name	Source	Address
3pool	MetaImplementationUSD.vy	0x5F890841f657d90E081bAbdB532A05996Af79Fe6
sBTC	MetaImplementationBTC.vy	0x2f956eee002b0debd468cf2e0490d1aec65e027f

30.6.2 Deposit Zaps

Deposit zaps for factory metapools are deployed to the mainnet at the following addresses:

Name	Source	Address
3pool Deposit Zap	DepositZapUSD.vy	0xA79828DF1850E8a3A3064576f380D90aECDD3359
sBTC Deposit Zap	DepositZapBTC.vy	0x7AbDBAf29929e7F8621B757D2a7c04d78d633834

30.6.3 Promoted Factory Pools

Factory metapools which have been promoted to the flagship Curve UI.

Pool	Source	Address
alUSD	MetaImplementationUSD.vy	0x43b4FdFD4Ff969587185cDB6f0BD875c5Fc83f8c
FRAX	MetaImplementationUSD.vy	0xd632f22692FaC7611d2AA1C0D552930D43CAEd3B

30.7 Other Chains

30.7.1 Polygon

Curve has several contracts deployed on Polygon. UI for these contracts is available at polygon.curve.fi.

Pools and Gauges

Name	Source	Address
ATriCrypto Pool	CurveCryptoSwap-Matic.vy	0x751B1e21756bDbc307CBcC5085c042a0e9AaEf36
ATriCrypto Zap	ZapAave.vy	0x3FCD5De6A9fC8A99995c406c77DDa3eD7E406f81
ATriCrypto LP Token	CurveTokenV4.vy	0x8096ac61db23291252574D49f036f0f9ed8ab390
ATriCrypto Root Chain Gauge	RootGaugePolygon.vy	0x060e386eCfBacf42Aa72171Af9EFe17b3993fC4F
ATriCrypto Child Chain Streamer	ChildChainStreamer.vy	0x060e386eCfBacf42Aa72171Af9EFe17b3993fC4F
ATriCrypto Reward Claimer	RewardClaimer.vy	0xe84AE0321f88349B5F1119464EEB242b7De51a69
ATriCrypto Gauge	RewardsOnlyGauge.vy	0xb0a366b987d77b5eD5803cBd95C80bB6DEaB48C0
Aave Pool	StableSwapAave.vy	0x445FE580eF8d70FF569aB36e80c647af338db351
Aave LP Token	CurveTokenV3.vy	0xE7a24EF0C5e95Ffb0f6684b813A78F2a3AD7D171
Aave Root Chain Gauge	RootGaugePolygon.vy	0xC48f4653dd6a9509De44c92beb0604BEA3AEe714
Aave Child Chain Streamer	ChildChainStreamer.vy	0xC48f4653dd6a9509De44c92beb0604BEA3AEe714
Aave Reward Claimer	RewardClaimer.vy	0x1de441Ef347c3E7fd512B1662B77B5bc4AC28Cc8
Aave Gauge	RewardsOnlyGauge.vy	0x19793B454D3AfC7b454F206Ffe95aDE26cA6912c
renBTC Pool	StableSwapREN.vy	0xC2d95EEF97Ec6C17551d45e77B590dc1F9117C67
renBTC LP Token	CurveTokenV3.vy	0xf8a57c1d3b9629b77b6726a042ca48990A84Fb49
renBTC Root Chain Gauge	RootGaugePolygon.vy	0x488E6ef919C2bB9de535C634a80afb0114DA8F62
renBTC Child Chain Streamer	ChildChainStreamer.vy	0x488E6ef919C2bB9de535C634a80afb0114DA8F62
renBTC Reward Claimer	RewardClaimer.vy	0xe89BC681C5cb6A3499E9dB97e0CE8558877Dd1A4
renBTC Gauge	RewardsOnlyGauge.vy	0xffbACcE0CC7C19d46132f1258FC16CF6871D153c

Rewards and Admin Fees

Name	Source	Address
WMATIC Distributor	RewardStream.vy	0xBdFF0C27dd073C119ebcb1299a68A6A92aE607F0
ABurner	ABurner.vy	0xA237034249290De2B07988Ac64b96f22c0E76fE0
Admin Fee Bridge (Polygon)	ChildBurner.vy	0x4473243A61b5193670D1324872368d015081822f
Admin Fee Bridge (Ethereum)	RootForwarder.vy	0x4473243A61b5193670D1324872368d015081822f

30.7.2 Fantom

Curve has several contracts deployed on Fantom. UI for these contracts is available at ftm.curve.fi.

Pools and Gauges

Name	Source	Address
2Pool Pool	StableSwap2Pool.vy	0x27E611FD27b276ACbd5Ffd632E5eAEBEC9761E40
2Pool LP Token	StableSwap2Pool.vy	0x27E611FD27b276ACbd5Ffd632E5eAEBEC9761E40
2Pool Root Chain Gauge	RootGaugeAnyswap.vy	0xb9C05B8EE41FDCbd9956114B3aF15834FDEDCb54
2Pool Child Chain Streamer	ChildChainStreamer.vy	0xb9C05B8EE41FDCbd9956114B3aF15834FDEDCb54
2Pool Gauge	RewardsOnlyGauge.vy	0x8866414733F22295b7563f9C5299715D2D76CAf4
fUSDT Pool	StableSwapFUSDT.vy	0x92D5ebF3593a92888C25C0AbEF126583d4b5312E
fUSDT LP Token	StableSwap2Pool.vy	0x92D5ebF3593a92888C25C0AbEF126583d4b5312E
fUSDT Root Chain Gauge	RootGaugeAnyswap.vy	0xfE1A3dD8b169fB5BF0D5dbFe813d956F39fF6310
fUSDT Child Chain Streamer	ChildChainStreamer.vy	0xfE1A3dD8b169fB5BF0D5dbFe813d956F39fF6310
fUSDT Gauge	RewardsOnlyGauge.vy	0x06e3C4da96fd076b97b7ca3Ae23527314b6140dF
renBTC Pool	StableSwapREN.vy	0x3eF6A01A0f81D6046290f3e2A8c5b843e738E604
renBTC LP Token	CurveTokenV3.vy	0x5B5CFE992AdAC0C9D48E05854B2d91C73a003858
renBTC Root Chain Gauge	RootGaugeAnyswap.vy	0xfDb129ea4b6f557b07BcDCedE54F665b7b6Bc281
renBTC Child Chain Streamer	ChildChainStreamer.vy	0xfDb129ea4b6f557b07BcDCedE54F665b7b6Bc281
renBTC Gauge	RewardsOnlyGauge.vy	0xBdFF0C27dd073C119ebcb1299a68A6A92aE607F0

30.7.3 XDai

Curve has several contracts deployed on XDai. UI for these contracts is available at xdai.curve.fi.

Pools and Gauges

Name	Source	Address
x3Pool Pool	StableSwap3Pool.vy	0x7f90122BF0700F9E7e1F688fe926940E8839F353
x3Pool LP Token	StableSwap3Pool.vy	0x1337BedC9D22ecbe766dF105c9623922A27963EC
x3Pool Root Chain Gauge	RootGaugeAnyswap.vy	0x6C09F6727113543Fd061a721da512B7eFCDD0267
x3Pool Child Chain Streamer	ChildChainStreamer.vy	0x6C09F6727113543Fd061a721da512B7eFCDD0267
x3Pool Gauge	RewardsOnlyGauge.vy	0x78CF256256C8089d68Cde634Cf7cDEFb39286470

CHAPTER

THIRTYONE

GLOSSARY OF TERMS

This glossary of terms contains definitions of commonly used terms within the Curve documentation.

This section is a work in progress - if a term is missing, feel free to open a pull request to add it.

Automated Market Maker (AMM) A decentralized asset trading pool that allows participants to buy or sell cryptocurrencies.

Base Pool The pool issuing the LP token that is used by a metapool.

Burning The process of withdrawing admin fees from the excahange contracts and distributing them to veCRV holders.

ERC20 A technical standard for implementing tokens within Ethereum. Often used interchangeably with the term token. The standard is viewable here.

LP Token Short for Liquidity Provider token. An ER20 token which represents a deposit into a Curve exchange contract, or other *AMM*.

Metapool A Curve pool where one of the tradeable assets is the *LP token* for another pool (base pool). Metapools are used to prevent liquidity fragmentation.

Pool See *automated market maker*.

Synth Short for "synthetic asset" - a derivative which tracks the price of another asset, offering exposure to price movements without requiring the user to hold the actual asset.

Underlying Coin An ERC20 token that has been deposited into a protocol and where the deposit is represented by another token. The other token (the "*wrapped coin*") may be used to claim back this original token.

veCRV Short for "vote-escrowed CRV". CRV that has been locked in the *voting contract*.

Wrapped Coin An ERC20 token used to represent the deposit of another token within a protocol. The original token has been "wrapped" in this new token. The originial token is referred to as the "*underlying coin*".

INDEX

C

D

F

G

L

M

P

R

S

U

V