**Curve DAO: Fee Collection and Distribution**[**ℑ**](https://curve.readthedocs.io/dao-fees.html#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](https://curve.readthedocs.io/factory-pools.html" \l "StableSwap.remove_liquidity" \o "StableSwap.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](https://curve.readthedocs.io/dao-ownership.html#dao-ownership-pool-proxy). This contract includes functionality to withdraw the fees, convert them to [3CRV](https://etherscan.io/token/0x6c3F90f043a72FA612cbac8115EE7e52BDe6E490), 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.

**Withdrawing Admin Fees**[**ℑ**](https://curve.readthedocs.io/dao-fees.html#withdrawing-admin-fees)

Admin fees are stored within each exchange contract and viewable via the **[admin\_balances](https://curve.readthedocs.io/exchange-pools.html" \l "StableSwap.admin_balances" \o "StableSwap.admin_balances)** public getter method. The contract owner may call to claim the fees at any time using **[withdraw\_admin\_fees](https://curve.readthedocs.io/exchange-pools.html" \l "StableSwap.withdraw_admin_fees" \o "StableSwap.withdraw_admin_fees)**. Most pools also include a function to donate pending fees to liquidity providers via **[donate\_admin\_fees](https://curve.readthedocs.io/exchange-pools.html" \l "StableSwap.donate_admin_fees" \o "StableSwap.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.

**The Burn Process**[**ℑ**](https://curve.readthedocs.io/dao-fees.html#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](https://github.com/curvefi/curve-dao-contracts/tree/master/contracts/burners).

**LPBurner[ℑ](https://curve.readthedocs.io/dao-fees.html" \l "lpburner" \o "Permalink to this heading)**

The LP Burner handles non-3CRV LP tokens, collected from metapools. The most common token burned via the LP burner is [sbtcCRV](https://etherscan.io/address/0x075b1bb99792c9E1041bA13afEf80C91a1e70fB3) 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**[**ℑ**](https://curve.readthedocs.io/dao-fees.html#LPBurner.set_swap_data)

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.

**SynthBurner[ℑ](https://curve.readthedocs.io/dao-fees.html" \l "synthburner" \o "Permalink to this heading)**

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](https://curve.readthedocs.io/exchange-cross-asset-swaps.html#cross-asset-swaps).

When the synth burner is called to burn a non-synthetic asset, it uses **[RegistrySwap.exchange\_with\_best\_rate](https://curve.readthedocs.io/registry-exchanges.html" \l "Swaps.exchange_with_best_rate" \o "Swaps.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](https://curve.readthedocs.io/dao-fees.html#dao-fees-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](https://docs.synthetix.io/integrations/settlement/) 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:**[**ℑ**](https://curve.readthedocs.io/dao-fees.html#SynthBurner.set_swap_for)

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:**[**ℑ**](https://curve.readthedocs.io/dao-fees.html#SynthBurner.add_synths)

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](https://docs.synthetix.io/contracts/source/contracts/Synth/" \l "currencykey) to validate the addresss and obtain the synth currency key.

Returns True.

**ABurner, CBurner, YBurner[ℑ](https://curve.readthedocs.io/dao-fees.html" \l "aburner-cburner-yburner" \o "Permalink to this heading)**

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](https://curve.readthedocs.io/dao-fees.html#dao-fees-underlying-burner).

There is no configuration required for these burners.

**MetaBurner[ℑ](https://curve.readthedocs.io/dao-fees.html" \l "metaburner" \o "Permalink to this heading)**

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](https://curve.readthedocs.io/registry-exchanges.html" \l "Swaps.exchange_with_best_rate" \o "Swaps.exchange_with_best_rate)** and transfers 3CRV directly to the [fee distributor](https://curve.readthedocs.io/dao-fees.html#dao-fees-distributor).

There is no configuration required for this burner.

**USDNBurner[ℑ](https://curve.readthedocs.io/dao-fees.html" \l "usdnburner" \o "Permalink to this heading)**

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](https://curve.readthedocs.io/dao-fees.html#dao-fees-distributor).

There is no configuration required for this burner.

**UniswapBurner[ℑ](https://curve.readthedocs.io/dao-fees.html" \l "uniswapburner" \o "Permalink to this heading)**

UniswapBurner is used for burning assets that are not supported by Curve, such as SNX recieved by the DAO via the [Synthetix trading incentives](https://sips.synthetix.io/sips/sip-63) 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](https://curve.readthedocs.io/dao-fees.html#dao-fees-underlying-burner).

There is no configuration required for this burner.

**UnderlyingBurner[ℑ](https://curve.readthedocs.io/dao-fees.html" \l "underlyingburner" \o "Permalink to this heading)**

The underlying burner handles assets that can be directly swapped to USDC, and deposits DAI/USDC/USDT into [3pool](https://www.curve.fi/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](https://docs.synthetix.io/contracts/source/contracts/Synthetix/#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:**[**ℑ**](https://curve.readthedocs.io/dao-fees.html#UnderlyingBurner.execute)

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.

**Fee Distribution**[**ℑ**](https://curve.readthedocs.io/dao-fees.html#fee-distribution)

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

[0xA464e6DCda8AC41e03616F95f4BC98a13b8922Dc](https://etherscan.io/address/0xa464e6dcda8ac41e03616f95f4bc98a13b8922dc)

Source code for this contract is available on [Github](https://github.com/curvefi/curve-dao-contracts/blob/master/contracts/FeeDistributor.vy).

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**[**ℑ**](https://curve.readthedocs.io/dao-fees.html#FeeDistributor.claim)

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](https://curve.readthedocs.io/dao-fees.html" \l "FeeDistributor.claim_many" \o "FeeDistributor.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**[**ℑ**](https://curve.readthedocs.io/dao-fees.html#FeeDistributor.claim_many)

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.

# The Curve DAO: Governance and Voting[ℑ](https://curve.readthedocs.io/dao-voting.html#the-curve-dao-governance-and-voting)

Curve uses [Aragon](https://aragon.org/) for governance and control of the protocol admin functionality. Interaction with Aragon occurs through a [modified implementation](https://github.com/curvefi/curve-aragon-voting) of the Aragon [Voting App](https://github.com/aragon/aragon-apps/tree/master/apps/voting).

Much of the following functionality is possible via the [DAO section](https://dao.curve.fi/dao) of the Curve website. The following section outlines DAO interactions via the CLI using the [Brownie console](https://eth-brownie.readthedocs.io/en/stable/interaction.html#using-the-console).

Deployment addresses can be found in the [addresses reference](https://curve.readthedocs.io/ref-addresses.html#addresses-aragon) section of the documentation.

## Creating a Vote[ℑ](https://curve.readthedocs.io/dao-voting.html#creating-a-vote)

A single vote can perform multiple actions. The [new\_vote.py](https://github.com/curvefi/curve-dao-contracts/blob/master/scripts/voting/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:
3. 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.

1. 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:
   3. brownie run voting/new\_vote make\_vote --network mainnet
   4. The vote should automatically appear within the site UX shortly.

## Inspecting Votes[ℑ](https://curve.readthedocs.io/dao-voting.html#inspecting-votes)

The [decode\_vote.py](https://github.com/curvefi/curve-dao-contracts/blob/master/scripts/voting/new_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.

## Voting[ℑ](https://curve.readthedocs.io/dao-voting.html#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

## Executing a Vote[ℑ](https://curve.readthedocs.io/dao-voting.html#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