<https://curve.readthedocs.io/dao-overview.html>

# The Curve DAO[ℑ](https://curve.readthedocs.io/dao-overview.html#the-curve-dao)

Curve DAO consists of multiple smart contracts connected by [Aragon](https://github.com/aragon/aragonOS). 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). Aragon’s standard one token, one vote method is replaced with a weighting system based on locking tokens.

Curve DAO has a token ([CRV](https://etherscan.io/token/0xd533a949740bb3306d119cc777fa900ba034cd52)) which is used for both governance and value accrual.

Source code for the Curve DAO can be found on [Github](https://github.com/curvefi/curve-dao-contracts).

# Curve DAO: Vote-Escrowed CRV[ℑ](https://curve.readthedocs.io/dao-vecrv.html#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](https://etherscan.io/address/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.

## Implementation Details[ℑ](https://curve.readthedocs.io/dao-vecrv.html#implementation-details)

User voting power wi 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（斜率） and bias（偏差） for the linear function wi(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(但是，我们将用户锁定的结束时间限制为以整周的时间).

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) ( 所有可能的过期时间都为整周，以使区块链的读取次数最多与错过的周数成正比，而不是用户数（可能很大）).

## Querying Balances, Locks and Supply[ℑ](https://curve.readthedocs.io/dao-vecrv.html#querying-balances-locks-and-supply)

**VotingEscrow.balanceOf(*addr: address*, *\_t: uint256 = block.timestamp*)→ uint256**[**ℑ**](https://curve.readthedocs.io/dao-vecrv.html#VotingEscrow.balanceOf)

Get the current voting power for an address.

* addr: User wallet address
* **>>>** vote\_escrow = Contract('0x5f3b5DfEb7B28CDbD7FAba78963EE202a494e2A2')
* **>>>** vote\_escrow.balanceOf('0xF89501B77b2FA6329F94F5A05FE84cEbb5c8b1a0')
* 5464191329389144503333564

**VotingEscrow.balanceOfAt(*addr: address*, *\_block: uint256*)→ uint256**[**ℑ**](https://curve.readthedocs.io/dao-vecrv.html#VotingEscrow.balanceOfAt)

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

This function is taken from the [MiniMe](https://github.com/Giveth/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)
* 5470188311017698310628752

**VotingEscrow.totalSupply()→ uint256**[**ℑ**](https://curve.readthedocs.io/dao-vecrv.html#VotingEscrow.totalSupply)

Calculate the current total voting power.

**>>>** vote\_escrow.totalSupply()

102535077684041114817306735

**VotingEscrow.totalSupplyAt(*\_block: uint256*)→ uint256**[**ℑ**](https://curve.readthedocs.io/dao-vecrv.html#VotingEscrow.totalSupplyAt)

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*)**[**ℑ**](https://curve.readthedocs.io/dao-vecrv.html#VotingEscrow.locked)

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': 5664716612269392397633736,

'end': 1736985600

}

## Working with Vote-Locks[ℑ](https://curve.readthedocs.io/dao-vecrv.html#working-with-vote-locks)

**VotingEscrow.create\_lock(*\_value: uint256*, *\_unlock\_time: uint256*)**[**ℑ**](https://curve.readthedocs.io/dao-vecrv.html#VotingEscrow.create_lock)

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*)**[**ℑ**](https://curve.readthedocs.io/dao-vecrv.html#VotingEscrow.increase_amount)

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*)**[**ℑ**](https://curve.readthedocs.io/dao-vecrv.html#VotingEscrow.increase_unlock_time)

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

# The Curve DAO: Liquidity Gauges and Minting CRV[ℑ](https://curve.readthedocs.io/dao-gauges.html#the-curve-dao-liquidity-gauges-and-minting-crv)

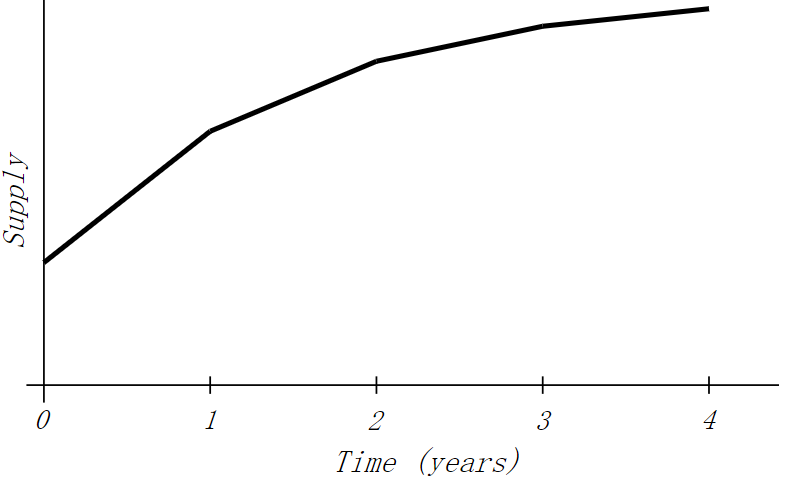
Curve incentivizes liquidity providers with the CRV, the protocol governance token. Allocation, distribution and minting of CRV are managed via several related DAO contracts:

* 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

## Implementation Details[ℑ](https://curve.readthedocs.io/dao-gauges.html#implementation-details)

### CRV Inflation[ℑ](https://curve.readthedocs.io/dao-gauges.html#crv-inflation)

CRV follows a piecewise linear inflation schedule（CRV遵循分段线性通货膨胀时间表）. The inflation is reduced by 21/4 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→∞" 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). 支持上述通胀时间表的初始通货膨胀率为r=22.0%。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.

### Liquidity Gauges[ℑ](https://curve.readthedocs.io/dao-gauges.html#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](https://curve.readthedocs.io/dao-gauges.html#dao-gauges-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](https://curve.readthedocs.io/dao-vecrv.html#dao-vecrv) contract.

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

To calculate a user’s share of r′, we must calculate the integral（微积分）:



where bu(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 Iu gives the amount of tokens which the user has to have minted to them. The user’s balance bu 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 Iu 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):



The value of Iis 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 Iu can be calculated as the current (before user’s action) value of Iis multiplied by the pre-action user’s balance, and sumed up across the user’s balances:



The per-user integral is possible to repalce with this sum because bu(t) changed for all times between tk-1 and tk.

### Boosting[ℑ](https://curve.readthedocs.io/dao-gauges.html#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](https://curve.readthedocs.io/dao-vecrv.html#dao-vecrv) contract, depending on their vote weight



The value of wi 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 bu 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 gauge for the full year). If that ever happens, the abandoned gauge gets less CRV.

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

### The Gauge Controller[ℑ](https://curve.readthedocs.io/dao-gauges.html#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.

## Gauge Types[ℑ](https://curve.readthedocs.io/dao-gauges.html#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 (stableswap pools): 0
* Fantom: 1
* Polygon (Matic): 2
* xDai: 4
* Ethereum (crypto pools): 5
* Arbitrum 7
* Avalanche 8
* Harmony 9

Types 3 and 6 have been deprecated.

## LiquidityGauge[ℑ](https://curve.readthedocs.io/dao-gauges.html" \l "liquiditygauge" \o "Permalink to this heading)

Each pool has a unique liquidity gauge. Deployment addresses can be found in the [addresses reference](https://curve.readthedocs.io/ref-addresses.html#addresses-gauges) section of the documentation.

There are several versions of liquidity gauge contracts in use. Source code for these contracts is available on [Github](https://github.com/curvefi/curve-dao-contracts/tree/master/contracts/gauges).

### Querying Gauge Information[ℑ](https://curve.readthedocs.io/dao-gauges.html#querying-gauge-information)

**LiquidityGauge.lp\_token()→ address: view**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#LiquidityGauge.lp_token)

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**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#LiquidityGauge.working_supply)

The “working supply” of the gauge - the effective total LP token amount after all deposits have been [boosted](https://curve.readthedocs.io/dao-gauges.html#dao-gauges-boost).

### Querying User Information[ℑ](https://curve.readthedocs.io/dao-gauges.html#querying-user-information)

**LiquidityGauge.balanceOf(*addr: address*)→ uint256: view**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#LiquidityGauge.balanceOf)

The current amount of LP tokens that addr has deposited into the gauge.

**LiquidityGauge.working\_balances(*addr: address*)→ uint256: view**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#LiquidityGauge.working_balances)

The “working balance” of a user - their effective balance after [boost](https://curve.readthedocs.io/dao-gauges.html#dao-gauges-boost) has been applied.

**LiquidityGauge.claimable\_tokens(*addr: address*)→ uint256: nonpayable**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#LiquidityGauge.claimable_tokens)

The amount of currently mintable CRV for addr from this gauge.

**Note**

Calling this function [modifies the state](https://vyper.readthedocs.io/en/stable/control-structures.html#mutability). 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**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#LiquidityGauge.integrate_fraction)

The total amount of CRV, both mintable and already minted, that has been allocated to addr from this gauge.

### Checkpoints[ℑ](https://curve.readthedocs.io/dao-gauges.html#checkpoints)

**LiquidityGauge.user\_checkpoint(*addr: address*)→ bool: nonpayable[ℑ](https://curve.readthedocs.io/dao-gauges.html" \l "LiquidityGauge.user_checkpoint" \o "Permalink to this definition)**

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.

### Deposits and Withdrawals[ℑ](https://curve.readthedocs.io/dao-gauges.html#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**](https://curve.readthedocs.io/dao-gauges.html#LiquidityGauge.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})
* 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**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#LiquidityGauge.approved_to_deposit)

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

### Killing the Gauge[ℑ](https://curve.readthedocs.io/dao-gauges.html#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](https://curve.readthedocs.io/dao-ownership.html#dao-ownership-agents) 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**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#LiquidityGauge.is_killed)

The current killed status of the gauge.

## LiquidityGaugeReward[ℑ](https://curve.readthedocs.io/dao-gauges.html" \l "liquiditygaugereward" \o "Permalink to this heading)

Along with measuring liquidity for CRV distribution, LiquidityGaugeReward stakes LP tokens into an SNX [staking rewards](https://github.com/Synthetixio/synthetix/blob/master/contracts/StakingRewards.sol) contract and handles distribution of an the additional rewards token. Rewards gauges include the full API of [LiquidityGauge](https://curve.readthedocs.io/dao-gauges.html#dao-gauges-liquidity-gauge), with the following additional methods:

### Querying Reward Information[ℑ](https://curve.readthedocs.io/dao-gauges.html#querying-reward-information)

**LiquidityGaugeReward.reward\_contract()→ address: view**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#LiquidityGaugeReward.reward_contract)

The address of the [staking rewards](https://github.com/Synthetixio/synthetix/blob/master/contracts/StakingRewards.sol) contract that LP tokens are staked into.

**LiquidityGaugeReward.rewarded\_token()→ address: view**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#LiquidityGaugeReward.rewarded_token)

The address of the reward token being received from **[reward\_contract](https://curve.readthedocs.io/dao-gauges.html" \l "LiquidityGaugeReward.reward_contract" \o "LiquidityGaugeReward.reward_contract)**.

**LiquidityGaugeReward.is\_claiming\_rewards()→ bool: view**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#LiquidityGaugeReward.is_claiming_rewards)

Boolean indicating if rewards are currently being claimed by this gauge.

### Calculating Claimable Rewards[ℑ](https://curve.readthedocs.io/dao-gauges.html#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**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#LiquidityGaugeReward.claimable_reward)

The total earned reward tokens, both claimed and unclaimed, for addr.

**LiquidityGaugeReward.claimed\_rewards\_for(*addr: address*)→ uint256: view**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#LiquidityGaugeReward.claimed_rewards_for)

The number of reward tokens already claimed for addr.

### Claiming Rewards[ℑ](https://curve.readthedocs.io/dao-gauges.html#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.

## LiquidityGaugeV2[ℑ](https://curve.readthedocs.io/dao-gauges.html#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.

### Querying Reward Information[ℑ](https://curve.readthedocs.io/dao-gauges.html#id2)

**LiquidityGaugeV2.reward\_contract()→ address: view**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#LiquidityGaugeV2.reward_contract)

The address of the [staking rewards](https://github.com/Synthetixio/synthetix/blob/master/contracts/StakingRewards.sol) contract that LP tokens are staked into.

**LiquidityGaugeV2.rewarded\_tokens(*idx: uint256*)→ address: view**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#LiquidityGaugeV2.rewarded_tokens)

Getter for an array of rewarded tokens currently being received by **[reward\_contract](https://curve.readthedocs.io/dao-gauges.html" \l "LiquidityGaugeV2.reward_contract" \o "LiquidityGaugeV2.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.

### Approvals and Transfers[ℑ](https://curve.readthedocs.io/dao-gauges.html#approvals-and-transfers)

**LiquidityGaugeV2.transfer(*\_to: address*, *\_value: uint256*)→ bool:**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#LiquidityGaugeV2.transfer)

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:**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#LiquidityGaugeV2.transferFrom)

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:**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#LiquidityGaugeV2.approve)

Approve the passed address to transfer the specified amount of tokens on behalf of the caller.

Returns True on success. Reverts on failure.

### Checking and Claiming Rewards[ℑ](https://curve.readthedocs.io/dao-gauges.html#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**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#LiquidityGaugeV2.claimable_reward)

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](https://vyper.readthedocs.io/en/stable/control-structures.html#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.sender): nonpayable**

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.

### Setting the Rewards Contract[ℑ](https://curve.readthedocs.io/dao-gauges.html#setting-the-rewards-contract)

**LiquidityGaugeV2.set\_rewards(\_reward\_contract: address, \_sigs: bytes32, \_reward\_tokens: address[8]): nonpayable**

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

"0xa694fc3a2e1a7d4d3d18b9120000000000000000000000000000000000000000"

# now we are ready to set the rewards contract

**>>>** gauge.set\_rewards(rewards, sigs, [reward\_token] + [ZERO\_ADDRESS] \* 7, {'from': alice})

## LiquidityGaugeV3[ℑ](https://curve.readthedocs.io/dao-gauges.html#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.

### Querying Reward Information[ℑ](https://curve.readthedocs.io/dao-gauges.html#id4)

**LiquidityGaugeV3.rewards\_receiver(*addr: address*)→ address: view**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#LiquidityGaugeV3.rewards_receiver)

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**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#LiquidityGaugeV3.last_claim)

The epoch timestamp of the last call to claim from **reward\_contract**.

### Checking and Claiming Rewards[ℑ](https://curve.readthedocs.io/dao-gauges.html#id5)

**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\_ADDRESS): nonpayable**

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**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#LiquidityGaugeV3.claimed_reward)

Get the number of already claimed reward tokens for a user.

**LiquidityGaugeV3.claimable\_reward(*\_addr: address*, *\_token: address*)→ uint256: view**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#LiquidityGaugeV3.claimable_reward)

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](https://curve.readthedocs.io/dao-gauges.html" \l "LiquidityGaugeV3.claimable_reward_write" \o "LiquidityGaugeV3.claimable_reward_write)** as a view method.

**LiquidityGaugeV3.claimable\_reward\_write(*\_addr: address*, *\_token: address*)→ uint256: nonpayable**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#LiquidityGaugeV3.claimable_reward_write)

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](https://curve.readthedocs.io/dao-gauges.html" \l "LiquidityGaugeV3.claimable_reward" \o "LiquidityGaugeV3.claimable_reward)**. This function does not claim/distribute pending rewards for a user.

## GaugeController[ℑ](https://curve.readthedocs.io/dao-gauges.html" \l "gaugecontroller" \o "Permalink to this heading)

GaugeController is deployed to the Ethereum mainnet at:

[0x2F50D538606Fa9EDD2B11E2446BEb18C9D5846bB](https://etherscan.io/address/0x2F50D538606Fa9EDD2B11E2446BEb18C9D5846bB).

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

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

### Querying Gauge and Type Weights[ℑ](https://curve.readthedocs.io/dao-gauges.html#querying-gauge-and-type-weights)

**GaugeController.gauge\_types(*gauge\_addr: address*)→ int128: view**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#GaugeController.gauge_types)

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**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#GaugeController.get_gauge_weight)

The current gauge weight for gauge\_addr.

**GaugeController.get\_type\_weight(*type\_id: int128*)→ uint256: view**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#GaugeController.get_type_weight)

The current type weight for type\_id as an integer normalized to 1e18.

**GaugeController.get\_total\_weight()→ uint256: view**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#GaugeController.get_total_weight)

The current total (type-weighted) weight for all gauges.

**GaugeController.get\_weights\_sum\_per\_type(*type\_id: int128*)→ uint256: view**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#GaugeController.get_weights_sum_per_type)

The sum of all gauge weights for type\_id.

### Vote-Weighting[ℑ](https://curve.readthedocs.io/dao-gauges.html#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**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#GaugeController.vote_user_power)

The total vote weight power allocated by user.

**GaugeController.last\_user\_vote(*user: address*, *gauge: address*)→ uint256: view**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#GaugeController.last_user_vote)

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*)**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#GaugeController.vote_user_slopes)

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): nonpayable**

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

### Adding New Gauges and Types[ℑ](https://curve.readthedocs.io/dao-gauges.html#adding-new-gauges-and-types)

All of the following methods are only be callable by the DAO [ownership admin](https://curve.readthedocs.io/dao-ownership.html#dao-ownership-agents) as the result of a successful [vote](https://curve.readthedocs.io/dao-voting.html#dao-voting).

**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**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#GaugeController.gauge_relative_weight)

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*)**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#GaugeController.change_type_weight)

Change the weight for a given gauge type.

Only callable by the DAO [ownership admin](https://curve.readthedocs.io/dao-ownership.html#dao-ownership-agents).

* type\_id Gauge type id
* weight New Gauge weight

## Minter[ℑ](https://curve.readthedocs.io/dao-gauges.html#minter)

Minter is deployed to the Ethereum mainnet at:

[0xd061D61a4d941c39E5453435B6345Dc261C2fcE0](https://etherscan.io/address/0xd061D61a4d941c39E5453435B6345Dc261C2fcE0).

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

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

### Minting CRV[ℑ](https://curve.readthedocs.io/dao-gauges.html#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**[**ℑ**](https://curve.readthedocs.io/dao-gauges.html#Minter.allowed_to_mint_for)

Getter method to check if minter has been approved to call mint\_for on behalf of for.