Curve DAO

Curve DAO consists of multiple smart contracts connected by Aragon. Apart from that, standard Aragon's 1 token = 1 vote method is replaced with the voting weight proportional to locktime, as will be described below.

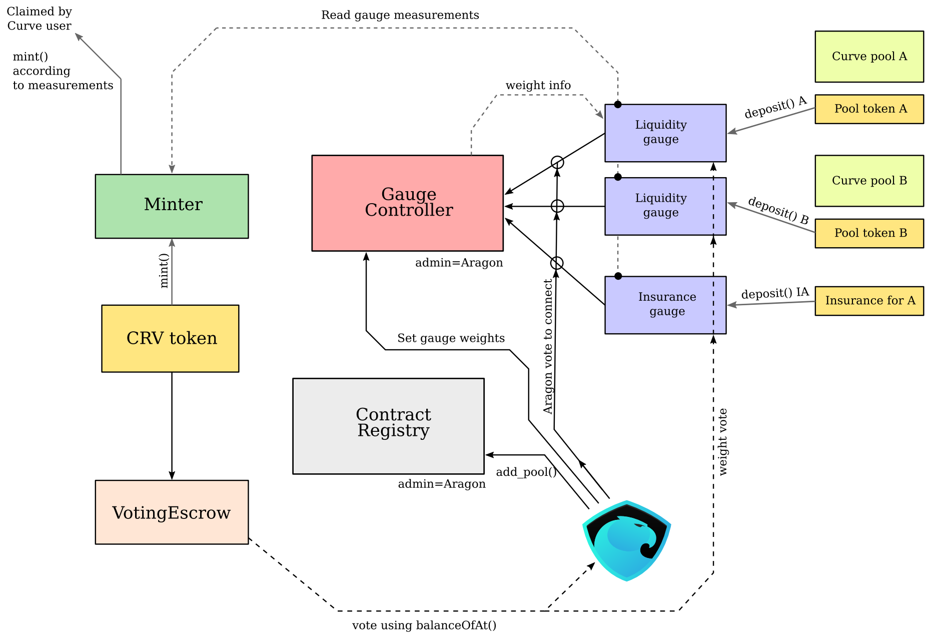

Figure 1: Curve DAO contracts managed by Aragon

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

Time-weighted voting. Vote-locked tokens in VotingEscrow

Instead of voting with token amount a, in Curve DAO tokens are lockable in a *VotingEscrow* for a selectable locktime t_l, where $t_l < t_{\max}$, and $t_{\max} = 4$ years. After locking, the time *left to unlock* is $t \le t_l$. The voting weight is equal to:

$$w = a \frac{t}{t_{\max}}.$$

In other words, the vote is both amount- and time-weighted, where the time counted is how long the tokens cannot be moved in future.

The account which locks the tokens cannot be a smart contract (because can be tradable and/or tokenized), unless it is one of whitelisted smart contracts (for example, widely used multi-signature wallets).

VotingEscrow tries to resemble Aragon's Minime token. Most importantly, `balanceOf() / balanceOfAt()` and `totalSupply() / totalSupplyAt()` return the time-weighted voting weight w and the sum of all of those weights $W = \sum w_i$ respectively. Aragon can interface *VotingEscrow* as if it was a typical governance token.

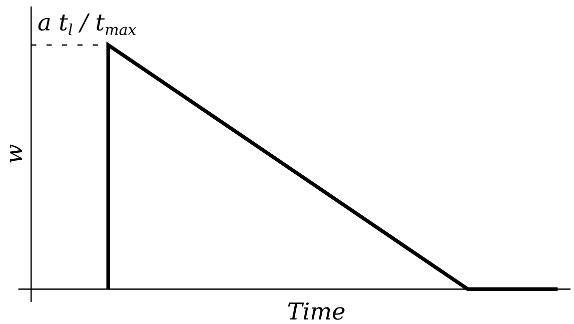

Figure 2: Voting weight of vote-locked tokens

Locks can be created or extended in time or token amount with `deposit()`, and `withdraw()` can remove tokens from the escrow when the lock is expired.

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 `user_point_history`. We also change slope and bias for the total voting power $W(t)$ and record in `point_history`. In addition, when 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 can be potentially large).

Essential interfaces

```
@external
@view
```

```
def get_last_user_slope(addr: address) -> int128:
    """
    @notice Get the recent recorded rate of voting power decrease
    @param  addr  Address of the user wallet
    @return Value of the slope
    """

@external
@view
def user_point_history__ts(_addr: address, _idx: uint256) -> uint256:
    """
    @notice Get the timestamp for the last recorded user's checkpoint
    @param  _addr  User wallet address
    @param  _idx   User epoch number
    @return  Timestamp of the checkpoint
    """

@external
@view
def locked__end(_addr: address) -> uint256:
    """
    @notice Get timestamp when user's lock finishes
    @param _addr User wallet
    @return Timestamp of the lock end
    """

@external
def checkpoint():
    """
    @notice Record global data to checkpoint
    """

@external
def deposit_for(_addr: address, _value: uint256):
    """
    @notice Deposit tokens for someone else and add to the lock
            Anyone (even a smart contract) can deposit for someone else,
            but cannot extend their locktime and cannot do it for a brand
            new user
    @param _addr User's wallet address
    @param _value Amount to add to user's lock
    """

@external
def create_lock(_value: uint256, _unlock_time: uint256):
    """
```

```
    @notice Deposit tokens for the sender
    @param _value Amount deposited
    @param _unlock_time Timestamp when the tokens will unlock. Rounded down
                        to whole weeks
    """

@external
def increase_amount(_value: uint256):
    """
    @notice Deposit more tokens for the sender while keeping the unlock time unchanged
    @param _value Amount of tokens to deposit and add to the lock
    """

@external
def increase_unlock_time(_unlock_time: uint256):
    """
    @notice Prolong the lock for the sender
    @param _unlock_time New timestamp for unlocking
    """

@external
def withdraw():
    """
    @notice Withdraw all tokens if the lock has expired
    """

@external
@view
def balanceOf(addr: address) -> uint256:
    """
    @notice Standard ERC20-compatible balanceOf which actually measures voting power
    @param addr User's wallet address
    @return User's voting power
    """

@external
@view
def balanceOfAt(addr: address, _block: uint256) -> uint256:
    """
    @notice Minime-compatible function to measure voting power at certain block in the past
    @param addr User's wallet address
    @param _block Block to calculate the voting power at
    @return Voting power
    """

@internal
```

```
@view
def supply_at(point: Point, t: uint256) -> uint256:
    """
    @notice Calculate total voting power at some point in the past
    @param point The point (bias/slope) to start search from
    @param t Time to calculate the total voting power at
    @return Total voting power at that time
    """

@external
@view
def totalSupply() -> uint256:
    """
    @notice Calculate current total voting power
    @return Total voting power
    """

@external
@view
def totalSupplyAt(_block: uint256) -> uint256:
    """
    @notice Calculate total voting power at some point in the past
    @param _block Block to calculate the total voting power at
    @return Total voting power at that block
    """
```

Inflation schedule. ERC20CRV

Token *ERC20CRV* is an ERC20 token which allows a piecewise linear inflation schedule. The inflation is dropping by $2^{1/4}$ every year. Only *Minter* contract can directly mint *ERC20CRV*, but only within the limits defined by inflation.

Each time the inflation changes, a new mining epoch starts.

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 those initial tokens 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 users of Curve, according to measurements taken by *gauges*. During the first year, the approximate inflow into circulating supply is 2 millions CRV per day, starting from 0.

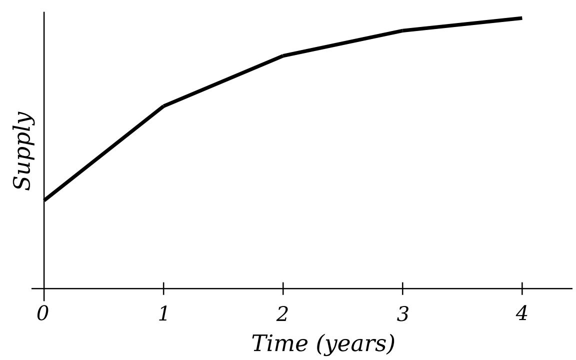

Figure 3: CRV token inflation schedule

System of Gauges. LiquidityGauge and GaugeController

In Curve, inflation is going towards users who use it. The usage is measured with Gauges. Currently there is just *LiquidityGauge* which measures, how much liquidity does the user provide. The same type of gauge can be used to measure "liquidity" provided for insurance.

For *LiquidityGauge* to measure user liquidity over time, the user deposits his LP tokens into the gauge using `deposit()` and can withdraw using `withdraw()`.

Coin rates which the gauge is getting depends on current inflation rate, and gauge *type weights* (which get voted on in Aragon). Each user gets inflation proportional to his LP tokens locked. Additionally, the rewards could be *boosted* by up to factor of 5 if user vote-locks tokens for Curve governance in *VotingEscrow*.

The user *does not* require to periodically check in. We describe how this is achieved in technical details.

GaugeController keeps a list of Gauges and their types, with weights of each gauge and type.

Gauges are per pool (each pool has an individual gauge).

LiquidityGauge implementation details

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.

In order to calculate user's fair share of r', we essentially need to calculate the integral:

$$I_u = \int \frac{r'(t)\, b_u(t)}{S(t)}\, dt,$$

where $b_u(t)$ is the balance supplied by 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 user has to have minted to him. The user's balance b_u changes every time 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 *LiquidityGauge* contract, the vaule of I_u is recorded in the `integrate_fraction` map, per-user.

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

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

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.

In order to incentivize users to participate in governance, and additionally create stickiness for liquidity, we implement the following mechanism. User's balance counted in the *LiquidityGauge* gets boosted by users locking CRV tokens in *VotingEscrow*, depending on their vote weight w_i:

$$b_u^* = \min\left(0.2\, b_u + 0.8\, S\frac{w_i}{W},\, b_u\right).$$

The value of w_i is taken at the time 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 much enough CRV, he is able to boost his stream of CRV by up to factor of 5 (reducing it slightly for all users who are not doing that).

Weight voting for gauges

Instead of simply voting for weight change in Aragon, users can allocate their vote-locked tokens towards one or other Gauge (pool). That pool will be getting a fraction of CRV tokens minted proportional to how much vote-locked tokens are allocated to it. Eeach user with tokens in VotingEscrow can change his/her preference at any time.

GaugeController implementation details

In order to implement weight voting, *GaugeController* has to include parameters handling linear character of voting power each user has.

Similarly to how it is done in *VotingEscrow*, *GaugeController* records points (bias+slope) per gauge in `vote_points`, *scheduled* changes in biases and slopes for those points in `vote_bias_changes` and `vote_slope_changes`, with those changes happening every round week, as well as current slopes for every user per-gauge in `vote_user_slopes`, along with the power the user has used and the time their vote-lock ends. Scheduling of the changes is handled by the private method `enact_vote`.

When user changes his preferences, the change of the gauge weight is scheduled for the next round week, not immediately. This is done in order to reduce the number of blockchain reads which need to be performed by each user: that will be proportional to the number of weeks since the last change instead of the number of interactions other users did.

Fee burner

Every pool allows the admin to collect fees using `withdraw_admin_fees`. Aragon should be able to collect those fees to the admin account and use them to buy and burn CRV on a free market once that free market exists. That should be possible to be done by anyone without a vote.

Gauges to rewards trading volume and governance votes

Both votes and trades are discrete events, so they can use the same sort of gauge. The idea is that each event has a weight which exponentially decays over time.

It should be possible to call a gauge contract every time a user votes in Aragon.