Deposits and Withdrawals

`deposit`

User deposits δ MATIC into the vault. When this happens the vault immediately delegates funds to the validator. Claimed rewards are piggy-backed on the user deposit. Changes in values:

- $\Sigma' = \Sigma + \Delta + \delta$
- $\Delta' = R$
- $S' = S + \delta/P + \phi R/P$
- $R' = 0$
- $s'_{user} = s_{user} + \delta/P$
- $s'_{treasury} = s_{treasury} + \frac{\phi R}{P}$

Note, when the total number of shares is zero, we set the price to be 1 by default. We also impose the condition that a minimum of 1 MATIC can be deposited to avoid inflation attacks.

▼ *Overflows and order of operations*

$s'_{user} = s_{user} + \delta/P$ we round down here so that the dust in the vault will skew towards positive.

$s'_{treasury} = s_{treasury} + \frac{\phi R}{P}$

Accruing Rewards

Rewards r accrue from the amount staked. This is not an explicit assignment, rather comes from the validator. It is included here simply for completeness.

- Σ remains constant
- S remains constant
- $R \to R + r$
- User share balances all remain constant.

Implicitly, both share price P and dust D increase from the change in R, but neither of these needs to be calculated.

`withdraw`

The changes in shares work in a similar way to deposit but in reverse. User initiates a withdrawal for w MATIC. This is sent to the validator. *Note that withdrawals are always taken from the validator and never from the rewards.*

Before the balances are changed etc, we perform a check to see if user has sufficient balance of MATIC to be able to withdraw. Check that the withdrawal amount is less than the user's shares valued in MATIC.

$$s_{user}P \geq w$$

If this passes, the main balances are changed as follows:

- $\Sigma' = \Sigma - w$
- $S' = S - w/P + \phi R/P$ user shares are burnt and treasury shares are minted.
- $R' = 0$ rewards are auto-claimed from the validator.
- $\Delta' = R$
- $s'_{user} = s_{user} - w/P$ User shares are burnt.
- $s'_{treasury} = s_{treasury} + \phi R/P$

The slight subtlety here is that the total amount that a user is entitled to is given by $s_{user}P$, which may in extreme circumstances be greater than Σ (i.e. the user is entitled to some of the reward balance R as well as their share of the staked balance). **This is the purpose of the reserve.** If all funds are withdrawn by users, $\Sigma = 0, S = s_{reserve}, R = s_{reserve}P$. As long as the initial balance deposited for the reserve account is as large as the threshold, there is never an unstaked balance of rewards that is larger than the reserve holdings.

Note on ϵ and `maxWithdraw`

Precision can lead to odd behaviour. For example if:

1. User deposits 1000 MATIC

2. User immediately tries to withdraw 1000 MATIC

The second transaction can revert due to rounding. Under the hood, the deposit function divides 1000 by the current share price and mints that amount of TruMATIC. The withdraw function now gets the current balance in MATIC by taking that TruMATIC amount and multiplying by the share price. There can be a small loss of precision which leads to the user only being able to withdraw 1000 MATIC - 1 WEI for example. Whilst this precision loss due to arithmetic operations in Solidity does not affect the user economically, it may have significant UX implications. A non-technical user will expect to be able to immediately withdraw or allocate the exact same amount that he deposited. As a result, we have decided to introduce the epsilon term to circumvent this precision loss. As such we allow users to withdraw slightly more MATIC than their TruMATIC balance would imply.

Inside the `maxWithdraw` function we add epsilon. If the user puts in a request to withdraw w MATIC, we:

- check to see if `maxWithdraw()` > w
  - If so, check to see if w/P > `balanceOf()`.
    - If so, burn all the user's TruMATIC
    - If not, burn w/P TruMATIC.
    - In either case create an unbond request for w MATIC and send to the validator.

This will have the effect of allowing the user to withdraw up to epsilon more than the share balance and price would imply.

▼ Justification for $\epsilon = 10^4 \text{ WEI}$

Over time the rounding errors will generally compound. If we assume that rounding errors are percentages to be conservative, then let the rounding error from successive operations be a $\epsilon_0, \epsilon_1, \ldots$ where for each operation, $O_p(1 + \epsilon_0) = O_a$ where O_a and O_p are the actual and precise outputs respectively. Then the cumulative error from successive operations is

$$(1 + \epsilon_0)(1 + \epsilon_1)\ldots - 1 = \sum_i \epsilon_i + \sum_{i,j} \epsilon_i \epsilon_j + \ldots$$

Each of these errors we expect to be of the order of 10^{-18} or smaller, so all but the first term can be more or less neglected. To allow users to be able to withdraw funds then, the error in the calculation of balance stored versus the

precise number should be no more than the threshold ϵ. Setting $\epsilon = 10,000$ then allows for rounding errors from 10,000 successive operations without causing issues.

In reality, this is a vast overestimate since the share price will change with each checkpoint as rewards accrue. After a checkpoint then, the TruMATIC balance returned by `balanceOf()` becomes the single source of truth for users' balances in the vault since their MATIC balances will have increased by an amount larger than any rounding errors. The threshold ϵ then only need be larger than the number of operations that a single user would perform since the last checkpoint. Setting it to 10,000 should hopefully be ample.

Recovering Staking fees

Staking fees sit across shares held by the Treasury user account and Dust that sits in the rewards account. These can be redeemed by the protocol at any time by:

- Calling `Restake` to drain the rewards vault and to convert any dust into shares.
- Calling `Withdraw` to redeem TruMATIC for MATIC.
- Distribution fees are automatically sent to the treasury as part of the `distribute` call, described later.

`Deposits and withdrawals to/from a specific validator`

Multi-validators support

The contracts maintains a list of validators that can be used for staking MATIC.
These are added by us and can be enabled or disabled.
One of them is designed as the *default validator*.

Deposits

When depositing funds, users are given the option to stake their MATIC into a specific validator of their choice.

If they decide to not choose the validator in which to stake their funds, the default validator will be the one receiving their funds to stake.

If they decide to stake into a specific validator, their MATIC will be sent to the designed validator to be staked.

Withdrawals

When withdrawing funds, users are given the option to withdraw their MATIC from a specific validator of their choice.

If they decide to not choose the validator from which to withdraw their funds, the default validator will be the one from which funds will be unstaked.

If they decide to withdraw from a specific validator, their MATIC will be unstaked and withdrawn from the designed validator.

A few notes:

- In the case when the validator doesn't have enough MATIC for the user to withdraw, it's the user's responsibility to either choose another validator or to withdraw from multiple validators by splitting the desired amount, as the contract doesn't offer any way to withdraw from multiple validators at once.
- The contract doesn't keep track of who deposited in which validator. A user depositing in validator A can withdraw from validator B, provided there are enough MATIC present in B to withdraw.

`compoundRewards`

We run a cron job constantly which checks the value of R. If $R > \theta R$ then we call the restake function.

The function can in principle be called by anyone. Gas is paid by the caller.

- $\Sigma' = \Sigma + R$. Rewards are sent to the validator.
- $s'_{treasury} = s_{treasury} + (\Sigma + \Delta + R)/P - S$. Treasury account has new shares added to it to keep the share price constant. Note, this additional number of shares could just be $\phi R/P$. Leaving this long form has the effect of sweeping any excess MATIC in the vault into the treasury in the form of shares. It is less computationally efficient however and so should only be used in the `compoundRewards` function which we expect will mostly be called by the protocol and not by users.
- $S' = (\Sigma + \Delta + R)/P$. New shares are added to the total.
- $R' = 0$. Rewards balance is emptied.

Implicitly, when this happens, the dust balance *D* goes to zero, since the rewards balance does too. The new shares minted that go into the treasury account represent the fees taken by the protocol.

▼ Overflows and order of operations

$$s_{treasury} = s_{treasury} + (\Sigma + \Delta + R)/P - S$$
$$S\prime = (\Sigma + \Delta + R)/P.$$

In addition to the restake function, claimed rewards (aka pending deposits) also need to be staked periodically when they hit a threshold *θΔ*. At this point we simply call `deposit(amount=0, address = 0x)`. This automatically sweeps any balances of MATIC in the vault and stakes them.