
Checker

Release 0.1alpha

Nomadic Labs / Tweag

Feb 15, 2022

CONTENTS:

1	Introduction	1
1.1	What is Checker?	1
1.2	System overview of a Checker deployment	2
1.3	Configuring and building Checker for different use-cases	2
2	Design	5
2.1	Concepts	5
2.1.1	The clock	5
2.1.2	Target and quantity	5
2.1.3	The instantaneous drift	6
2.1.4	Oracles	6
2.2	Algorithmic control	7
2.3	Burrows	8
2.3.1	Burrowing and overburrowing	8
2.3.2	Burrow fee	9
2.3.3	Imbalance adjustment	9
2.4	Liquidation	9
2.5	Liquidation auction	10
2.6	CFMM	10
3	Functional Specification	11
3.1	Checker	11
3.1.1	Working with burrows	11
3.1.2	CFMM Exchange	13
3.1.3	Liquidation Auctions	15
3.1.4	Maintenance entrypoints	16
3.1.5	FA2 Interface	17
3.1.6	FA2 Views	17
3.1.7	Deployment	20
3.2	wtez	20
3.2.1	Overview	20
3.2.2	Deposits and withdrawals	20
3.2.3	FA2 Interface	21
3.2.4	Internal entrypoints	22
3.3	wctez	23
3.3.1	Overview	23
3.3.2	Minting and redeeming tokens	23
3.3.3	FA2 Interface	24
4	Operational descriptions	25

4.1	System Parameters	25
4.1.1	State	25
4.1.2	Initialization	26
4.1.3	Price API	26
4.1.4	Adjustment index	26
4.1.5	Touching	27
4.1.6	Misc	30
4.2	Burrow State & Liquidation	30
4.2.1	State	30
4.2.2	Touching	31
4.2.3	Is a burrow collateralized (i.e. not overburrowed)?	31
4.2.4	Is a burrow a candidate for liquidation?	31
4.2.5	How much collateral should we liquidate?	32
4.2.6	Was the liquidation warranted?	34
4.2.7	What if the liquidation was warranted?	34
4.2.8	Misc	35
4.3	Liquidation Auctions	35
4.3.1	State	35
4.3.2	Initiating a liquidation	36
4.3.3	Cancelling a liquidation slice	36
4.3.4	Lot auction	36
4.3.5	Touching a liquidation_slice	37
4.3.6	Claiming a winning bid	37
4.3.7	Maintenance	37
4.4	CFMM subsystem	37
4.4.1	State	37
4.4.2	Initialization	38
4.4.3	General notes on the interfaces	38
4.4.4	Adding liquidity	38
4.4.5	Removing liquidity	39
4.4.6	Buying Kit	40
4.4.7	Selling Kit	41
4.4.8	Misc	42
5	Deploying Checker	43
6	Glossary	45
6.1	Kit	45
6.2	Burrow	45
6.3	Circulating kits	45
6.4	Outstanding kits	45
6.5	Liquidation lot	45
6.6	Liquidation slice	45
6.7	Liquidation queue	46
6.8	Imbalance	46
6.9	Imbalance adjustment	46
7	Indices and tables	47

INTRODUCTION



1.1 What is Checker?

Checker is a generic piece of software for creating *robocoins* on the [Tezos blockchain](#). It is an open source project supported by [Nomadic Labs](#), [Tweag](#) and [TZ Connect Berlin](#).

A *robocoin* is a cryptographic token (or “coin”) that tracks an external measure of value, by using various feedback mechanisms to algorithmically control its supply. There is no widely accepted definition of the term, which has been created for the purpose of this document.

While it may have some similarities, Checker’s robocoin mechanism differs from the design of other coin systems which aim to track an external value (such as a currency). Coin designs such as those of the [JPM](#) and Facebook’s [Libra](#) relied on regulation by a central authority or administrator, which Checker does not require. The Dai stablecoin

managed by the [MakerDAO project](#) decentralizes its governance, using voting to manage the financial risks of Dai and to ensure its stability: Checker does not require governance by decentralized voting either.

The Checker system eliminates governance by automating the regulation of the robocoin's value. Specifically, Checker algorithmically controls the coin's supply by creating incentives for creation and destruction, in order to maintain a smoothed drift of the value of the coin towards that of its external target measure.

Any number of Checker deployments can exist on a Tezos chain, each managing a separate robocoin which tracks a different external measure of value.

1.2 System overview of a Checker deployment

At its core, Checker is a single smart contract which is tied at build and deployment time to the following external contracts:

1. An oracle contract that will be periodically queried for the value of its external target measure.
2. An FA2 token contract which is used as collateral within Checker. Users transfer these tokens to Checker when creating burrows or depositing collateral and receive tokens when making withdrawals from their burrows.
3. An FA2 token contract which is used in Checker's CFMM.
4. The new `ctez` system, which provides the `ctez` token: this token has a value which tracks that of Tez itself, but without affording the holder any baking rights. `ctez` is used in Checker's CFMM for instances of Checker configured to use `tez` collateral.

A Checker deployment enables its users to mint and burn its robocoin: Checker manages peripheral "burrow" contracts on those users' behalf, and places their collateral deposits there.

For users who wish to exchange the robocoin with other commodities, or who wish to provide liquidity for such exchanges, the deployment includes a CFMM (Constant Function Market Maker) facility. This allows an exchange between the robocoin and a single other FA2 token which can be configured at build time.

Finally, the deployment allows for liquidation of collateral tokens against which depositors have minted robocoins, to manage when relative prices changes render the collateral insufficient. A batched auction mechanism facilitates this liquidation.

The deployment adjusts the terms for minting, burning and collateralising its robocoin algorithmically based on its current market price and the target oracle feed, such that the price drifts towards the target.

An FA2 interface is provided for each deployment's robocoin.

1.3 Configuring and building Checker for different use-cases

Checker uses a configuration file, `checker.yaml`, for building the contract for different use-cases `TODO`: Add reference to a config file doc section. While some configurations such as system constants do not cause structural differences in the Checker contract, other configurations such as the collateral type require slight structural variations.

Checker currently supports the following variations:

1. Collateral type
 - a. `collateral_type=fa2` - In this case, the collateral type is an existing FA2 token and the CFMM is configured to use the same FA2 token.

- b. `collateral_type=tez` - A special case of (1). In this case Checker's CFMM is required to use the *wctez* FA2 wrapper for *ctez* and the FA2 collateral contract must use the *wtez* FA2 wrapper for *tez*, both of which are included in the Checker repository.
- 2. Drift derivative curve type
 - a. `curve_type=bang-bang` - In this case, the drift derivative implementation uses a “bang-bang” curve which is discontinuous.
 - b. `curve_type=continuous` - In this case, the drift derivative implementation uses a continuous curve.
- 3. TODO: Oracle type (index vs. token-based)

This document starts by introducing some important concepts underlying the notion of this robocoin and its implementation, such as the notion of target, quantity, and oracles. Then, we present the algorithmic component that ensures the stability of the target. However, all the robocoin system with its control mechanism needs kits to be created and destroyed. This is made possible by the notion of burrows. Therefore, we present in a subsequent part burrows and their lifecycle, including creation, auctions, and liquidation. Finally, we discuss how the automatic control mechanism can be complemented by on-chain governance, which completes the whole picture of the notion of robocoin.

2.1 Concepts

The following sections define concepts which together work to form Checker.

2.1.1 The clock

Any computation on a blockchain happens in discrete time. We note the timestamps at which the Checker system is updated as a series of increasing timestamps, t_i . Ideally, these updates happen every time a new block is added to the blockchain and thus, under the current economic protocol, those timestamps are separated by about a minute each.

That said, the system is designed to be resilient to changes in the interblock time and also to occasional missing updates upon added blocks.

2.1.2 Target and quantity

In Checker, robocoins, denominated in “kit” are algorithmically balanced to achieve a certain degree of steadiness with respect to a **target**, which is expressed in terms of a **quantity** and an **index**.

The word “kit” is chosen because it’s short, simple to pronounce, and means a baby fox (which seems appropriate for a smart currency).

The **quantity**, q_{t_i} , expressed in kit^{-1} , is a time-dependent property of the system which can fluctuate upwards or downwards.

The **index**, expressed in kit, is an external time-dependent measure of value. Examples of an index include:

- “the median hourly minimum wage across OCDE countries, expressed in kits”, and
- “the value of one CHF (Swiss Franc), expressed in kits”.

This index is provided through a combination of off-chain and on-chain oracles.

The **target** p_{t_i} is the dimensionless product of the **index** and the **quantity**. Examples of **target** include:

- “the median of q_{t_i} hours of minimum wage across all OCDE countries as a number of kits”,

- “the minimal compensation, in kits, that an airline might owe a passenger, pursuant to the Vienna convention, should they lose q_{t_i} kg of luggage as a number of kits”, and
- “how many kits for q_{t_i} Swiss Francs”.

Intuitively, if the target p_{t_i} is below 1, then kits are, in a sense “too expensive” and if the target is above 1 then kits are, in a sense, “too cheap”.

2.1.3 The instantaneous drift

The **instantaneous drift** d_t , or just **drift**, is a dynamic parameter (which varies continuously over time) used for adjusting the quantity q . It represents the growth or shrinking of q per unit of time, and is measured in **Nepers** (Np), or sub-units thereof, such as centinepers (cNp). The drift is implicitly continuous, piecewise-quadratic between two clock ticks with continuous derivatives.

The system applies algorithmic control mechanisms in order to produce a drift that is defined at all time t .

Essentially, our control mechanism provides d' the derivative of d at clock ticks, and we interpolate quadratically between them.

We can set $d_0 = d'_0 = 0$; the system will adjust automatically so the initial values do not particularly matter (so long as they are not absurdly large).

$d'_{t_{i+1}}$ is computed as defined in the *algorithmic control section* and $d_{t_{i+1}}$ is then:

$$d_{t_{i+1}} = d_{t_i} + \frac{1}{2}(d'_{t_i} + d'_{t_{i+1}})(t_{i+1} - t_i)$$

The drift $d(t)$ is applied to the quantity such that $q'(t) = d(t)q(t)$, hence:

$$q_{t_{i+1}} = q_{t_i} \exp \left(\left(d_{t_i} + \frac{1}{6}(2d'_{t_i} + d'_{t_{i+1}})(t_{i+1} - t_i) \right) (t_{i+1} - t_i) \right)$$

The term in the exponential is the product between

- the time span $(t_{i+1} - t_i)$, and
- a term $\left(d_{t_i} + \frac{1}{6}(2d'_{t_i} + d'_{t_{i+1}})(t_{i+1} - t_i) \right)$ which is the average of the quadratic function over the period.

Note that given the practical constants involved (d is typically on the order of 10^0 to 10^1 cNp / year), the exponential can be approximated by $\exp(x) = 1 + x$.

2.1.4 Oracles

An oracle feed provides the **tez**-denominated value of the external index (e.g. 1 CHF), which we label tz_t . The contract providing the oracle feed should be reliable: for some external measures it might be advisable for that contract to give Checker the median of three or more externally-observed values.

Filtered oracle feeds

Protected index

The feed of external oracle values is itself filtered.

We define the **protected index**, \widehat{tz}_t , as:

$$\widehat{tz}_{t_i} = \widehat{tz}_{t_{i-1}} \times \text{clamp} \left(\frac{tz_{t_i}}{\widehat{tz}_{t_{i-1}}}, e^{-\epsilon(t_i - t_{i-1})}, e^{\epsilon(t_i - t_{i-1})} \right)$$

where $\text{clamp}(x, \min, \max)$ returns x constrained to the inclusive range from \min to \max .

We suggest a value of $\epsilon = 0.05 \text{ cNp/min}$ – that’s about 72 cNp / day, so the filter can catch up to a 2x or 0.5x move in 24 hours, and a 3% move in an hour.

\widehat{tz}_t is like the suspension of a car, it lags behind large moves, but is insensitive to spikes (real or fabricated).

In addition, we define the following prices

Minting index

The feed $tz_t^{\text{minting}} = \max(tz_t, \widehat{tz}_t)$ is the maximum of tz_t and \widehat{tz}_t .

Liquidation index

The feed $tz_t^{\text{liquidation}} = \min(tz_t, \widehat{tz}_t)$ is the minimum of tz_t and \widehat{tz}_t .

Changing oracle feeds

The Oracle feed is initially fixed. We **strongly recommend** that the current Tezos protocol be upgraded to allow bakers to signal in each block support for adding or removing oracles.

Target

The Checker system includes a [uniswap](#)-like CFMM (Constant Function Market Maker) exchange contract which gives an indication of the price of kit in tez, k_t with unit tez kit^{-1} . The target can be computed as

$$p_t = q_t tz_t / k_t$$

For example: suppose

$$\begin{cases} tz_t &= 0.36 \text{ xtz} \\ k_t &= 0.3 \text{ xtz/kit} \\ q_t &= 0.9 \text{ kit}^{-1} \end{cases}$$

Then $p_t = 1.08$, and since $p_t > 1$, we can say that kit is too cheap.

We do not need to filter the target feed as it only affects the drift in a bounded way that is, even if tz_t experiences wild, short lived swing, it will not have a major effect on the system.

2.2 Algorithmic control

Consider the measure of imbalance

$$\log p_t = \log(q_t tz_t / k_t).$$

All logarithm values are expressed in cNp or centinepers (for small values, a centineper is almost the same as a percentage point so you can safely read 2 cNp and 2% as roughly equivalent).

We algorithmically define the drift d_t via its rate of change, noted d'_t . d'_t is computed, at any clock tick t , based on the imbalance:

$$\begin{cases} |\log p_t| < 0.5 \text{ cNp} & \Rightarrow d'_t = 0 \\ 0.5 \text{ cNp} \leq |\log p_t| < 5 \text{ cNp} & \Rightarrow d'_t = \text{sign}(\log p_t) 0.01 \text{ cNp/day}^2 \\ 5 \text{ cNp} \leq |\log p_t| & \Rightarrow d'_t = \text{sign}(\log p_t) 0.05 \text{ cNp/day}^2 \end{cases}$$

It's easy to imagine models where d'_t depends continuously on $\log p_t$ but our intuition is that such models tend to be less robust than simple bang-bang models such as the one above.

Two remarks: 1. The unit of d_t is cNp/day because it represents the growth or shrinking of q_t per unit of time. Therefore, it is natural that the unit of d'_t is in cNp/day². To get a better intuition of those quantities suppose drift starts at 0 cNp / day and imbalance stays below -0.5 cNp but above -5 cNp for a month, the drift would grow to 0.3 cNp / day, and q_t would increase by 4.65 cNp (about 4.76%). If imbalance stayed below 5 cNp for a month, the drift would go from 0 cNp / day to 1.5 cNp / day in a month, increasing q_t by 23.25 cNp (about 26.18%).

2. When compared to MakerDAO this is essentially setting a rate of increase or decrease for a (potentially negative!) stability fee programmatically, based on prices, as opposed to votes.

2.3 Burrows

Burrows are a form of “deposit account”, and each is an independent smart contract, originated by the Checker contract.

A burrow serves to hold tez collateral against which kits may be minted and subsequently burned, subject to certain restrictions. Collateral may generally be added and withdrawn over time, again subject to restrictions. Kits minted from a burrow (“outstanding kits”) become part of the burrow owner’s personal kit balance, and they may be spent or transferred freely. A corresponding portion of the collateral in the burrow will then be locked up, and it cannot be withdrawn unless enough kits are later returned to the burrow and burned. Burrows are similar to CDPs in MakerDAO.

Burrow creation deposit: When a burrow is created, its owner must pay a burrow creation deposit, which won’t count towards the collateral and is only there to reward people marking the burrow for liquidation. If the owner closes the burrow, the deposit is recovered with it. We propose to set the deposit at 1 tez.

Since the burrow holds tez on the owner’s behalf, the owner may optionally specify a delegate for that balance.

2.3.1 Burrowing and overburrowing

Burrowing is the act of minting kits out of a burrow, and the kits accrue to an **outstanding** kits balance. To avoid overpopulation of kits, the burrowing is limited depending on the number of tez in the burrow in relation to the outstanding kit balance. Generally, kits can be minted so long as the tez in the burrow is at least f^{minting} times the number of outstanding kits multiplied by q_t multiplied by tz_t^{minting} . We propose $f^{\text{minting}} = 2.1$.

Assume for instance $tz_t^{\text{minting}} = 0.36\text{xtz}$ and $q_t = 1.015$. To mint 10 kits, one would require $2.1 \times 10 \times 0.36 \times 1.015 = 7.673$ xtz in the burrow. When further kits can no longer be burrowed due to insufficient tez collateral, the burrow is said to be **overburrowed**.

Even once further minting is blocked due to overburrowing, market fluctuations in kit and tez values may lead to a situation in which the ratio of kits outstanding versus tez in the burrow exceeds a higher safety threshold of $f^{\text{liquidation}} q_t tz_t^{\text{liquidation}}$, in which case the burrow is considered under-collateralized and can be marked for liquidation, as we’ll see later.

2.3.2 Burrow fee

While a burrow has outstanding kits, it continuously incurs a compounding burrow fee. This is an amount added to the outstanding kit balance, but this amount does not represent kits given to the burrow owner. The result of this is that over time slightly more kits are required to be burned in a burrow in order to release its collateral.

A 0.5 cNp fee per year is assessed and implicitly credited to a ctez / kit CFMM exchange contract which is described below in this document. It's important that this is done implicitly, i.e. whenever the CFMM contract is called, it knows exactly what its balance is.

Note: it might seem at first like the fee is “paid” for, individually, by the burrow creators but, from an economic perspective, it is equally valid to view it as being paid for, collectively, by all the kit holders, as the fee can be offset by an adjustment of the drift.

2.3.3 Imbalance adjustment

The *imbalance adjustment* takes the form of either an *adjustment fee* or an *adjustment bonus*. The exact amount of the fee (or bonus) is set depending on the imbalance between the number of kits in circulation and the outstanding number of kits that would need to be burned to close all burrows.

In general those numbers should be equal but, imperfect liquidations could cause the numbers to become different. (Imperfect liquidations happen when a burrow is completely liquidated, but not all of the outstanding kits can be recovered: there is an outstanding balance of kits that were minted out of the burrow, but there are no more tez left in that burrow.) If the former (outstanding kits) is greater than the latter (kits in circulation), the adjustment fee is increased and the extra kits are burned. If some burrows are left unfilled, this restores the balance.

The adjustment fee / bonus is capped at ± 5 cNp per year, is proportional to the imbalance in cNp and saturates when the imbalance hits 20%.

This means that if the system were to end up being undercollateralized, the drift would become lower and dilute the value of the kit, whereas if the system were to end up being overcollateralized the drift would become higher concentrating the value of the kit.

2.4 Liquidation

In situations where a burrow is overburrowed and, furthermore, beyond the liquidation threshold, it can be marked for liquidation by anyone. Liquidation is the process of selling some or all of its tez collateral at auction for kit, which will be burned to reduce the burrow's excessive outstanding kit balance.

There is a reward for marking a burrow for liquidation, equal to 0.1 cNp of the tez collateral plus the burrow creation deposit.

Note that we rely directly on the target and *not* any kit / tez price we might observe on-chain. The reason is that, kits being off target should *not* cause a hardening or loosening of burrowing rules.

Once a burrow is marked for liquidation, one can determine the amount of tez that needs to be sold for kit at the current $tz_t^{minting}$ price in order to return the burrow in a state where any outstanding kits could have just been minted (including refilling the burrow creation deposit, in case another liquidation is later needed). If there would not be enough tez to refill the creation deposit, everything is liquidated and the burrow is simply closed.

That portion of the tez collateral is sent to a queue for auction and the burrow is assigned a corresponding lot number. As the queue receives tez to sell for kit, it chops them up in increments of tez_batch . We suggest $tez_batch = 10,000$ xtz. Each lot is given a lot number which is held by the burrows which contributed the tez to the lot.

Portions of a burrow's tez collateral may be queued in multiple lots, due either to splitting of large amounts across lots, or to successive partial liquidations.

2.5 Liquidation auction

If there are any lots of tez collateral waiting to be sold for kit, Checker starts an open, ascending bid auction. There is a reserve price set using k_t which declines exponentially over time as long as no bid has been placed. Once a bid is placed, the auction continues. Every bid needs to improve over the previous bid by at least 0.33 cNp and adds the longer of 20 blocks or 20 minutes, to the time before the auction expires.

When liquidating, we liquidate 10% more than we are currently computing. We call a liquidation “warranted” when the burrow would have been targettable for liquidation had we used, retrospectively, the average price obtained in the liquidation auction. Once the liquidation price is known (after an auction) we look at whether that liquidation was “warranted” — that is, it was proven to be necessary. If it was, we destroy 10% of the kit proceeds of the auction. These 10% do not go towards reducing the outstanding kit balance of the burrow, they are just gone, for everyone. If it turned out that a liquidation was not warranted, all 100% of the liquidation proceeds are credited to the burrow.

2.6 CFMM

There is a CFMM (Constant Function Market Maker) exchange facility attached to the checker contract. It is much like a standard CFMM contract (including the ability to mint and redeem tokens representing a contribution of liquidity to the contract) except that its balance in kit increases over time as kits are minted out of burrows to pay for part of the burrowing fee. This balance is adjusted any time the checker contract is called, looking back at the last time the contract was called and calculating the fee incurred in between.

The contract’s implied ctez/kit price is used as part of the parameter calculations.

FUNCTIONAL SPECIFICATION

3.1 Checker

3.1.1 Working with burrows

Burrows are implicitly associated with their owner via the sender's address upon their creation. A sender can operate multiple burrows over time: owners are expected to identify each burrow uniquely with an arbitrary numeric ID they supply. These numbers need not be contiguous.

Create a burrow

Create and return a new burrow containing the supplied amount of FA2 token as collateral, minus the creation deposit. Fails if the collateral given is not enough to cover the creation deposit, if the sender does not own said amount of collateral, or if Checker is not authorized as an operator for the sender's collateral.

Before calling:

- In all cases, you will need to ensure that Checker is authorized in the collateral's FA2 contract as an operator.
- For `collateral_type=tez`, wtez is used as collateral in Checker. Please refer to the wtez documentation for depositing tez and getting wtez tokens: [[Deposit tez](#)].

```
create_burrow: (pair (pair nat (option key_hash)) nat)
```

Parameter	Field Type	Description
id	nat	An arbitrary number to identify the burrow among the sender's burrows
delegate	option key_hash	An optional delegate for the created burrow contract
tok	nat	The amount of token supplied as collateral (including creation deposit)

Deposit collateral in a burrow

Deposit an amount of FA2 token as collateral to a burrow. Fails if the burrow does not exist, if the sender does not own said collateral, or if Checker is not authorized as an operator for the sender's collateral.

Before calling:

- In all cases, you will need to ensure that Checker is authorized in the collateral's FA2 contract as an operator.
- For `collateral_type=tez`, wtez is used as collateral in Checker. Please refer to the wtez documentation for depositing tez and getting wtez tokens: [[Deposit tez](#)].

`deposit_collateral:` (pair nat nat)

Parameter	Field Type	Description
id	nat	The sender's ID for the burrow in which to deposit the collateral tokens
tok	nat	The amount of token supplied to be used as collateral

Withdraw collateral from a burrow

Withdraw an amount of collateral from a burrow. Fails if the burrow does not exist, if this action would overburrow it, or if the sender is not the burrow owner.

`withdraw_collateral:` (pair nat nat)

Parameter	Field Type	Description
id	nat	The sender's ID for the burrow from which to withdraw the collateral tokens
amount	nat	The amount of collateral to withdraw

Mint kit

Mint an amount of kit from a specific burrow. Fails if the burrow does not exist, if there is not enough collateral, or if the sender is not the burrow owner.

`mint_kit:` (pair nat nat)

Parameter	Field Type	Description
id	nat	The sender's ID for the burrow in which to mint the kit
amount	nat	The amount of kit to mint

Burn kit

Deposit/burn an amount of kit to a burrow. If there is excess kit, simply credit it back to the burrow owner. Fails if the sender does not own the specified amount of kit, if the burrow does not exist, or if the sender is not the burrow owner.

`burn_kit:` (pair nat nat)

Parameter	Field Type	Description
id	nat	The sender's ID for the burrow in which to burn the kit
amount	nat	The amount of kit to burn

Activate an inactive burrow

Activate a currently inactive burrow. Fails if the burrow does not exist, if the burrow is already active, if the amount of collateral given is not enough to cover the creation deposit, if the sender does not own said collateral, or if Checker is not authorized as an operator for the sender's collateral.

Before calling:

- In all cases, you will need to ensure that Checker is authorized in the collateral's FA2 contract as an operator.
- For `collateral_type=tez`, wtez is used as collateral in Checker. Please refer to the wtez documentation for depositing tez and getting wtez tokens: [[Deposit tez](#)].


```
activate_burrow: (pair nat nat)
```

Parameter	Field Type	Description
id	nat	The sender's ID for the burrow to activate
tok	nat	The amount of token supplied as collateral (including creation deposit)

Deactivate a burrow

Deactivate a currently active burrow. Fails if the burrow does not exist, if it is already inactive, if it is overburrowed, if it has kit outstanding, or if it has collateral sent off to auctions. If deactivation is successful, emits an FA2 transfer to the given address.

```
deactivate_burrow: (pair nat address)
```

Parameter	Field Type	Description
id	nat	The sender's ID for the burrow to deactivate
receiver	address	The address to send the burrow's collateral and creation deposit to

Perform burrow maintenance

Perform maintenance tasks on a burrow (i.e., update it's outstanding kit according to the system changes that have taken place since the last time the burrow was operated on). Fails if the burrow does not exist.

```
touch_burrow: (pair address nat)
```

Parameter	Field Type	Description
owner	address	The burrow owner's address
id	nat	The sender's ID for the burrow to deactivate

Set the delegate for a burrow

Set the delegate of a burrow. Fails if if the sender is not the burrow owner or if the deployed checker instance does not use tez as collateral.

```
set_burrow_delegate: (pair nat (option key_hash))
```

Parameter	Field Type	Description
id	nat	The sender's ID for the burrow
delegate	option key_hash	The key_hash of the new delegate's address, or none

3.1.2 CFMM Exchange

Buy kit using cfmm token

Buy some kit from the CFMM contract in exchange for the FA2 token used in the cfmm. Fails if Checker is not authorized as an operator for the sender's cfmm token, if the desired amount of kit cannot be bought, or if the deadline has passed.

Before calling:

- In all cases, you will need to ensure that Checker is authorized in the cfmm token's FA2 contract as an operator.
- For `collateral_type=tez`, wctez is used as the cfmm token. Please refer to the wctez documentation for depositing ctez and getting wctez tokens: [\[Mint tokens\]](#).

`buy_kit:` (pair (pair nat nat) timestamp)

Parameter	Field Type	Description
ctok	nat	An amount of cfmm token to be sold for kit
kit	nat	The minimum amount of kit expected to be bought
deadline	timestamp	The deadline for the transaction to be valid

Sell kit for cfmm token

Sell some kit in exchange for the FA2 token used in the cfmm. Fails if the sender does not own the specified amount of kit, if the desired amount of cfmm token cannot be bought, or if the deadline has passed.

`sell_kit:` (pair (pair nat nat) timestamp)

Parameter	Field Type	Description
kit	nat	The amount of kit to be sold
ctok	nat	The minimum amount of cfmm token expected to be bought
deadline	timestamp	The deadline for the transaction to be valid

Provide liquidity

Deposit some cfmm token and kit for liquidity in exchange for receiving liquidity tokens. If the given amounts do not have the right ratio, the CFMM contract keeps all the cfmm token given and as much of the given kit as possible with the right ratio, and returns the leftovers, along with the liquidity tokens. Fails if Checker is not authorized as an operator for the sender's cfmm token, or if the sender does not own the specified amount of kit.

Before calling:

- In all cases, you will need to ensure that Checker is authorized in the cfmm token's FA2 contract as an operator.
- For `collateral_type=tez`, wctez is used as the cfmm token. Please refer to the wctez documentation for depositing ctez and getting wctez tokens: [\[Mint tokens\]](#).

`add_liquidity:` (pair (pair nat nat) nat timestamp)

Parameter	Field Type	Description
ctok	nat	The amount of cfmm token to supply as liquidity
kit	nat	The maximum amount of kit to supply as liquidity
min_tokens	nat	The minimum number of liquidity tokens expected to be bought
deadline	timestamp	The deadline for the transaction to be valid

Withdraw liquidity

Redeem some liquidity tokens in exchange for cfmm tokens and kit in the right ratio. Fails if the sender does not own the specified liquidity tokens.

```
remove_liquidity: (pair (pair nat nat) nat timestamp)
```

Parameter	Field Type	Description
amount	nat	The number of liquidity tokens to redeem
ctok	nat	The minimum amount of cfmm token expected
kit	nat	The minimum amount of kit expected
deadline	timestamp	The deadline for the transaction to be valid

3.1.3 Liquidation Auctions

Mark a burrow for liquidation

Mark a burrow for liquidation. Fails if the burrow does not exist, or if it is not a candidate for liquidation. If the operation is successful, a payment is made to `Tezos.sender` with the liquidation reward.

```
mark_for_liquidation: (pair address nat)
```

Parameter	Field Type	Description
owner	address	The burrow owner's address
id	nat	The sender's ID for the burrow to mark for liquidation

Process completed liquidation slices

Process a number of liquidation slices (i.e., amounts of collateral that have been auctioned off as part of completed liquidation auctions). Fails if any of the identifiers given are not valid. Pointers to slices of incomplete auctions are ignored.

```
touch_liquidation_slices: (list nat)
```

Parameter	Field Type	Description
slice_ptrs	list nat	The unique identifiers of the slices to be processed

Cancel pending liquidation slices

Cancel the liquidation of a liquidation slice. Fails if the sender is not the burrow owner, if the slice is part of an ongoing or completed auction, or if the burrow is currently overburrowed.

```
cancel_liquidation_slice: nat
```

Parameter	Field Type	Description
slice_ptr	nat	The unique identifier of the slice whose liquidation is to be cancelled

Bid in the current liquidation auction

Bid in the current liquidation auction. Fails if the sender does not own the specified amount of kit, if there is no ongoing auction, or if the bid is too low.

```
liquidation_auction_place_bid: (pair nat nat)
```

Parameter	Field Type	Description
auction_id	nat	The unique identifier of the currently ongoing liquidation auction
kit	nat	The amount of kit to be bid

Claim the collateral from a winning auction bid

Claim the rewards of a completed liquidation auction. Fails if the sender is not the auction winner, if the auction is still ongoing, or if the completed auction still has unprocessed liquidation slices. If the operation is successful, an FA2 transfer of the collateral is made to `Tezos.sender` with the auction winnings.

```
liquidation_auction_claim_win: nat
```

Parameter	Field Type	Description
auction_id	nat	The unique identifier of the completed auction

3.1.4 Maintenance entrypoints

Perform Checker internal maintenance

Perform housekeeping tasks on the contract state. This includes:

1. updating the system parameters;
2. accruing burrowing fees to the cfmm;
3. updating auction-related info (completing an old / starting a new auction);
4. processing a limited number of liquidation slices from completed auctions;
5. updating the index by consulting the oracle.

This operation credits an amount of kit (that is a function of time passed since the last time `touch` was called) to `Tezos.sender`.

```
touch: unit
```

Parameter	Field Type	Description
unit	unit	()

Apply an Oracle update

Internal. Receive a price update from the registered oracle.

```
receive_price: pair nat nat
```

Parameter	Field Type	Description
price	pair nat nat	The current index, as a pair of the numerator and the denominator

3.1.5 FA2 Interface

Query balance

```
balance_of: (pair (list %requests (pair (address %owner) (nat %token_id)))
  (contract %callback
    (list (pair (pair %request (address %owner) (nat %token_id))
      ↪ (nat %balance))))))
```

Update operators

```
update_operators: (list (or (pair %add_operator (address %owner) (address %operator)
  ↪ (nat %token_id))
  (pair %remove_operator (address %owner) (address
  ↪ %operator) (nat %token_id))))
```

3.1.6 FA2 Views

Checker exposes a number of FA2 views in its contract metadata. Standard token views are provided, as are a number of custom views provided for integration convenience, e.g. for use by front-end applications.

Standard FA2 views

The following standard FA2 views are supported:

- `get_balance`
- `total_supply`
- `all_tokens`
- `is_operator`

Estimate yield when buying kit with cfmm tokens

Get the maximum amount of kit that can be expected to be received for the given amount of cfmm token (when calling `buy_kit`), based on the current market price.

```
buy_kit_min_kit_expected : nat -> nat
```

Parameter	Field Type	Description
ctok	nat	The amount of cfmm token to be sold to the cfmm

Estimate yield when selling kit for cfmm tokens

Get the maximum amount of cfmm token that can be expected to be received for the given amount of kit (when calling `sell_kit`), based on the current market price.

```
sell_kit_min_ctok_expected : nat -> nat
```

Parameter	Field Type	Description
kit	nat	The amount of kit to be sold to the cfmm

Estimate kit requirements when adding liquidity

Get the minimum amount of kit that needs to be deposited when adding liquidity for the given amount of cfmm token (when calling `add_liquidity`), based on the current market price.

```
add_liquidity_max_kit_deposited : nat -> nat
```

Parameter	Field Type	Description
ctok	nat	The amount of cfmm token to be given as liquidity

Estimate yield when adding liquidity

Get the maximum amount of the liquidity token that can be expected to be received for the given amount of cfmm token (when calling `add_liquidity`), based on the current market price.

```
add_liquidity_min_lqt_minted : nat -> nat
```

Parameter	Field Type	Description
ctok	nat	The amount of cfmm token to be given as liquidity

Estimate cfmm token yield when removing liquidity

Get the maximum amount of cfmm token that can be expected to be received for the given amount of liquidity token (when calling `remove_liquidity`), based on the current market price.

```
remove_liquidity_min_ctok_withdrawn : nat -> nat
```

Parameter	Field Type	Description
liquidity	nat	The amount of liquidity token to be returned to the cfmm

Estimate kit yield when removing liquidity

Get the maximum amount of kit that can be expected to be received for the given amount of liquidity token (when calling `remove_liquidity`), based on the current market price.

```
remove_liquidity_min_kit_withdrawn : nat -> nat
```

Parameter	Field Type	Description
liquidity	nat	The amount of liquidity token to be returned to the cfmm

Find maximum kit that can be minted

Returns the maximum amount of kit that can be minted from the given burrow.

```
burrow_max_mintable_kit : pair address nat -> nat
```

Parameter	Field Type	Description
owner	address	The burrow owner's address
id	nat	The sender's ID for the burrow in question

Check whether a burrow is overburrowed

```
is_burrow_overburrowed : pair address nat -> bool
```

Parameter	Field Type	Description
owner	address	The burrow owner's address
id	nat	The sender's ID for the burrow in question

Check whether a burrow can be liquidated

```
is_burrow_liquidatable : pair address nat -> bool
```

Parameter	Field Type	Description
owner	address	The burrow owner's address
id	nat	The sender's ID for the burrow in question

Get details on the current liquidation auction

Fails if there is currently no liquidation auction.

```
current_liquidation_auction_details: unit -> view_current_liquidation_auction_details_res
```

Parameter	Field Type	Description
unit	unit	()

3.1.7 Deployment

Deploy a lazy function

Prior to sealing, the bytecode for each lazy function must be deployed.

```
deployFunction: (pair int bytes)
```

Deploy metadata

Prior to sealing, the bytecode for all metadata must be deployed.

```
deployMetadata: bytes
```

Seal the contract and make it ready for use

```
sealContract: (pair (pair (pair address address) address) address)
```

3.2 wtez

3.2.1 Overview

wtez is a wrapper contract which issues wtez FA2 tokens which are always equal in value to tez. It is designed for use cases where users wish to use tez collateral in Checker. In this case Checker deals with the wtez FA2 tokens instead of with tez directly.

Each account can have up to exactly one vault contract associated with it which holds the tez deposited from that account.

3.2.2 Deposits and withdrawals

Deposit tez

Deposit the amount of tez in the transaction to the sender's vault in exchange for wtez tokens. If the account does not already have a vault contract an operation will be emitted originating it.

```
deposit: unit
```

Parameter	Field Type	Description
unit	unit	()

Withdraw tez

Withdraw the specified amount of `tez` from the sender's vault in exchange for `wtez` tokens. Fails if `amount` is greater than the sender's `wtez` balance. If the account does not already have a vault contract an operation will be emitted originating it.

```
withdraw: nat
```

Parameter	Field Type	Description
amount	nat	The amount of tez to withdraw

Set the delegate for a vault

Set the delegate of the sender's vault. If the account does not already have a vault contract an operation will be emitted originating it.

```
set_delegate: (pair nat (option key_hash))
```

Parameter	Field Type	Description
delegate	option key_hash	The key_hash of the new delegate's address, or none

3.2.3 FA2 Interface

Query balance

```
balance_of: (pair (list %requests (pair (address %owner) (nat %token_id)))
  (contract %callback
    (list (pair (pair %request (address %owner) (nat %token_id))
      ↪ (nat %balance))))))
```

Update operators

```
update_operators: (list (or (pair %add_operator (address %owner) (address %operator))
  ↪ (nat %token_id))
  (pair %remove_operator (address %owner) (address
  ↪ %operator) (nat %token_id))))
```

Transfer tokens

```
transfer: (list %transfer
  (pair
    (address %from_)
    (list %txs
      (pair
        (address %to_)
        (pair
          (nat %token_id)
          (nat %amount)
        )
      )
    )
  )
```

(continues on next page)

(continued from previous page)

_____)
_____)
_____)
_____)

3.2.4 Internal entryptoints

The following entrypoints are used internally in wtez and cannot be called by anyone except the wtez contract itself.

Send tez to a vault

Internal. Calls the specified vault's `%vault_receive_tez` entrypoint with the specified amount of `mutez`. Fails if the sender is not the `wtez` contract.

```
call_vault_receive_tez: (pair address tez)
```

Parameter	Field Type	Description
vault_address	address	The address of the vault
amount	mutez	The amount of mutez to send from the main contract to the vault

Send tez from a vault to a contract

Internal. Calls the specified vault's `%vault_send_tez_to_contract` entrypoint, sending the specified amount of mutez from the vault to the provided contract address. Fails if the sender is not the wtez contract.

```
call_vault_send_tez_to_contract: (pair address (pair tez address))
```

Parameter	Field Type	Description
vault_address	address	The address of the vault
amount	tez	The amount of mutez to send from the main contract to the vault
recipient	address	The address of the contract which will receive the mutez

Send tez from a vault to another vault

Internal. Calls the specified vault's `%vault_send_tez_to_vault` entrypoint, sending the specified amount of `mutez` from the vault to the provided vault address. Fails if the sender is not the `wtez` contract.

```
call_vault_send_tez_to_vault: (pair address (pair tez address))
```

Parameter	Field Type	Description
vault_address	address	The address of the vault
amount	tez	The amount of mutez to send from the main contract to the vault
recipient	address	The address of the vault which will receive the mutez

Set a vault's delegate

Internal. Calls the specified vault's `%vault_set_delegate` entrypoint with the provided delegate. Fails if the sender is not the `wtez` contract.

`call_vault_set_delegate: (pair address (option key_hash))`

Parameter	Field Type	Description
<code>vault_address</code>	<code>address</code>	The address of the vault
<code>delegate</code>	<code>option key_hash</code>	An optional delegate for the vault contract

3.3 wtez

3.3.1 Overview

`wtez` is a wrapper contract which issues `wtez` FA2 tokens in exchange for `ctez` tokens. It was designed for use in Checker's CFMM which only works with FA2 tokens since `ctez` itself only provides an FA1.2 interface.

Each `wtez` token is always worth exactly one `ctez` token.

3.3.2 Minting and redeeming tokens

Mint tokens

Mint `wtez` tokens by transferring the corresponding amount of `ctez` tokens from the sender to the contract. Fails if the contract is not approved to spend the specified amount of `ctez` tokens on the sender's behalf or if the sender's `ctez` balance is less than the specified amount.

Before calling:

- In order to mint `wtez`:
 - You must already have some `ctez`. This is generally done by calling `ctez`'s `%create` entrypoint to create an oven then calling its `%mint_or_burn` entrypoint to mint `ctez`. Please refer to the [ctez project](#) for more details.
 - You must also approve `wtez` to spend the specified amount of `ctez` by calling `ctez`'s FA1.2 contract's `%approve` entrypoint.

`mint: unit`

Parameter	Field Type	Description
<code>amount</code>	<code>nat</code>	The amount of <code>ctez</code> tokens to transfer to the contract

Redeem tokens

Redeem the specified amount of `wctez` tokens for the same amount of `ctez` tokens. Fails if the sender's `wctez` balance is less than the specified amount.

`withdraw: nat`

Parameter	Field Type	Description
<code>amount</code>	<code>nat</code>	The amount of <code>wctez</code> tokens to redeem

3.3.3 FA2 Interface

Query balance

```
balance_of: (pair (list %requests (pair (address %owner) (nat %token_id)))
  (contract %callback
    (list (pair (pair %request (address %owner) (nat %token_id))
      ↪ (nat %balance))))))
```

Update operators

```
update_operators: (list (or (pair %add_operator (address %owner) (address %operator)
  ↪ (nat %token_id))
  (pair %remove_operator (address %owner) (address
  ↪ %operator) (nat %token_id))))
```

Transfer tokens

```
transfer: (list %transfer
  (pair
    (address %from_)
    (list %txs
      (pair
        (address %to_)
        (pair
          (nat %token_id)
          (nat %amount)
        )
      )
    )
  )
)
```

OPERATIONAL DESCRIPTIONS

This section contains a breakdown of the logic and calculations in various aspects of Checker.

4.1 System Parameters

A operational description of Checker's internal parameters, and operations on them. NOTE: here we focus primarily on the specifics of the calculations; for the meaning of the concepts, see *Design*.

4.1.1 State

- `q`: of type (1 / kit).
- `index`: of type `tez`.
- `protected_index`: of type `tez`.
- `target`: TODO: said dimensionless, but I think `tez/kit`? Hmmm. I have to re-check the units of measure.
- `drift_derivative`
- `drift`
- `outstanding_kit`: approximation of the total amount of kit that would be currently required to close all burrows.
- `circulating_kit`: approximation of the total amount of kit that is currently in circulation.
- `last_touched`: the last time the parameters were touched.

And two additional indices, one used in the calculation of *burrowing fees* and one in the calculation of the *imbalance adjustment*:

- `burrow_fee_index`
- `imbalance_index`

4.1.2 Initialization

We currently initialize checker with the following parameters:

```
q = 1
index = 1xtz
protected_index = 1xtz
target = 1
drift = 0
drift_derivative = 0
outstanding_kit = 0kit
circulating_kit = 0kit
last_touched = now
burrow_fee_index = 1
imbalance_index = 1
```

4.1.3 Price API

```
tz_minting      = max index protected_index (in tez)
tz_liquidation  = min index protected_index (in tez)
```

To calculate the current prices in (tez/kit), we multiply with the current quantity:

```
minting_price    = q * tz_minting
liquidation_price = q * tz_liquidation
```

Note

The definition of `tz_minting` and `tz_liquidation` implies that at any given moment, `tz_minting >= tz_liquidation > 0`. Combined with `fminting > fliquidation`, we have that

```
tz_minting * fminting > tz_liquidation * fliquidation
```

which is useful in liquidation logic (see `burrow-state-liquidations.md`).

4.1.4 Adjustment index

The adjustment index, as required by burrowing logic, can be calculated from the system parameters as the product of the burrow fee index and the imbalance index:

```
adjustment_index = burrow_fee_index * imbalance_index
```

4.1.5 Touching

Touching the system parameters has the effect of updating all aforementioned fields, and calculating the burrowing fees that need to be accrued to the cfmm sub-contract. This is done under the assumption that we have available the current time `now`, the current index `index_now` (calculated by the medianizer), and the current price of kit in tez `kit_in_tez_now` (calculated by the cfmm sub-contract). In fact, the cfmm sub-contract gives us the one calculated at the end of the last block, to make manipulation a little harder. We update each field:

`last_touched`

Update the timestamp from the last time it was touched to now

```
new_last_touched = now
```

`index`

Update the index from the last time the parameters were touched to the current one

```
new_index = index_now
```

`protected_index`

Update the protected index, by multiplying it with a bounded factor:

```
new_protected_index = old_protected_index * clamp (current_index / protected_index, ↵
↵low, high)
```

where `low` and `high` depend on how much time has passed since the last time the parameters were touched, effectively limiting how fast `protected_index` can change:

```
low  = exp (-epsilon * (now - last_touched))
high = exp (+epsilon * (now - last_touched))
```

NOTE: $\exp(x) = 1 + x$ here; we expect the contract to be touched rather frequently, which keeps the exponent rather small, which makes this a good approximation of \exp .

`drift_derivative`

For the calculation of the derivative of drift, `drift_derivative`, we only use the last-observed target (TODO: show how we get from the original formula with the logarithms to this?) We calculate as follows:

```
new_drift_derivative =
  -0.0005 / (secs_in_a_day ^ 2) , if target <= exp (-high_
↵bracket)
  -0.0001 / (secs_in_a_day ^ 2) , if exp (-high_bracket) < target <= exp (-low_
↵bracket)
  0 , if exp (-low_bracket) < target < exp ( low_
↵bracket)
  0.0001 / (secs_in_a_day ^ 2) , if exp ( low_bracket) <= target < exp ( high_
↵bracket)
  0.0005 / (secs_in_a_day ^ 2) , if exp ( high_bracket) <= target
```

drift

For the calculation of the current drift, we use the following formula:

```
new_drift = old_drift + (1/2) * (old_drift_derivative + new_drift_derivative) * (now -  
↪ last_touched)
```

q

For the calculation of the current quantity q , we use the following formula:

```
new_q = old_q  
      * exp (  
          (old_drift + (1/6) * ((2 * old_drift_derivative) + new_drift_derivative) *  
↪ (now - last_touched))  
          * (now - last_touched)  
      )
```

NOTE: $\exp(x) = 1 + x$ here; **TODO:** not sure if the exponent is small enough for this to be a good approximation.

target

```
new_target = new_q * (new_index / kit_in_tez_now)
```

burrow_fee_index

The burrow fee index is updated linearly on the number of seconds that have passed since the last time the parameters were touched.

```
new_burrow_fee_index = old_burrow_fee_index  
                      * (1 + burrow_fee_rate * (now - last_touched) / seconds_in_a_  
↪ year)
```

imbalance_index

The imbalance index is also updated linearly on the number of seconds that have passed since the last time the parameters were touched

```
new_imbalance_index = old_imbalance_index  
                      * (1 + imbalance_rate * (now - last_touched) / seconds_in_a_year)
```

but `imbalance_rate` varies, depending on the difference between `old_outstanding_kit` and `old_circulating_kit`:

```
imbalance_rate =  
    clamp  
    ( imbalance_scaling_factor * (circulating - outstanding) / circulating,  
      -imbalance_limit,  
      +imbalance_limit  
    )
```


or, equivalently:

```

imbalance_rate =
  min (imbalance_scaling_factor * (circulating - outstanding) / circulating, ↵
↵+imbalance_limit), if circulating >= outstanding
  max (imbalance_scaling_factor * (circulating - outstanding) / circulating, -
↵-imbalance_limit), if circulating < outstanding

```

And in the edge cases the `imbalance_rate` is calculated as follows:

- if `old_circulating_kit = 0` and `old_outstanding_kit = 0` then `imbalance_rate = 0`.
- if `old_circulating_kit = 0` and `old_outstanding_kit > 0` then `imbalance_rate = -imbalance_limit`. (the outstanding kit is *infinitely* greater than the circulating kit, so the rate is saturated).

Intermediate `outstanding_kit`

In order to compute the updates for the two remaining fields (`outstanding_kit` and `circulating_kit`), we first need to calculate the current amount of kit outstanding, taking into account the accrued burrowing fee, thus

```

outstanding_with_fees = old_outstanding_kit * (new_burrow_fee_index / old_burrow_fee_
↵index)

```

Accrual to cfmm

The accrued burrowing fees are to be given to the cfmm sub-contract. The total amount we easily compute as

```

accrual_to_cfmm = outstanding_with_fees - old_outstanding

```

`outstanding_kit`

To obtain the updated `outstanding_kit`, we need to account for both the accrued burrowing fees, and the imbalance adjustment

```

new_outstanding_kit = old_outstanding_kit
                      * (new_burrow_fee_index / old_burrow_fee_index)
                      * (new_imbalance_index / old_imbalance_index)

```

or equivalently

```

new_outstanding_kit = outstanding_with_fees * (new_imbalance_index / old_imbalance_
↵index)

```

`circulating_kit`

Finally, to obtain the up-to-date `circulating_kit`, we just need to record the new kit in circulation, that is, `accrual_to_cfmm`:

```
new_circulating_kit = old_circulating_kit + accrual_to_cfmm
```

NOTE: If the current timestamp is identical to that stored in the parameters, we do not perform any of the above.

4.1.6 Misc

- `seconds_in_a_year` = 31556952 (= (365 + 1/4 - 1/100 + 1/400) days * 24 * 60 * 60)
- `seconds_in_a_day` = 86400 (= 24 * 60 * 60)
- `low_bracket` = 0.005
- `high_bracket` = 0.05
- `imbalance_scaling_factor` = 0.75
- `imbalance_limit` = 0.05

4.2 Burrow State & Liquidation

An operational interpretation of the burrow state and operations on it.

4.2.1 State

- `active`: whether the burrow is supported by a creation deposit. If not, it's considered "inactive".
- `address`: the address of the contract holding the burrow's collateral and creation deposit.
- `collateral`: the amount of tez stored in the burrow. Collateral that has been sent to auctions **does not** count towards this amount; for all we know, it's gone forever.
- `outstanding_kit`: the amount of kit that is outstanding from the burrow. This **does not** take into account kit we expect to receive (to burn) from pending auctions. However, `outstanding_kit` does increase over time, since the burrowing fee and the adjustment fee are added to it. So, effectively, before doing anything else with a burrow, we update its state ("touch" it, see below).
- `collateral_at_auction`: the total amount of tez that has been sent to auctions from this burrow, to be sold for kit.
- `last_touched`: the last time the burrow was touched.
- `adjustment_index`: the last observed adjustment index (at time `last_touched`).

4.2.2 Touching

First thing to do before considering any of the things below is to update the state of the burrow, by touching it. The effect of this is to

- Update the timestamp in the burrow to reflect the last time it was touched

```
new_last_touched = now
```

- To add accrued burrow and adjustment fee to its outstanding kit

```
new_outstanding = old_outstanding * (new_adjustment_index / old_adjustment_index)``
```

Note that if the current timestamp is identical to that stored in the burrow, we do not perform any of the above.

Each of the following operations implicitly touch the burrow (i.e., perform the above updates) before doing anything else.

4.2.3 Is a burrow collateralized (i.e. not overburrowed)?

The burrow is considered collateralized if the following holds:

```
collateral >= outstanding * fminting * current_minting_price (1)
```

`collateral` here refers to the amount of tez stored in the burrow (collateral that has been sent to auctions **does not** count towards this amount; for all we know, it's gone forever).

`outstanding_kit` here refers to the accrued amount of kit that is outstanding from the burrow (kit we expect to receive from pending auctions **is not** considered burned here, but still outstanding).

4.2.4 Is a burrow a candidate for liquidation?

The burrow cannot be marked for liquidation if the following holds:

```
collateral >= optimistic_outstanding * fliquidation * liquidation_price (2)
```

`collateral` here refers to the amount of tez stored in the burrow (collateral that has been sent to auctions **does not** count towards this amount; for all we know, it's gone forever).

`outstanding_kit` here refers to the accrued amount of kit that is outstanding from the burrow. In this case we optimistically **do take into account** kit we expect to receive from pending auctions at the `current_minting_price`, but pessimistically assume that these pending auctions are warranted (so we lose the liquidation penalty). That is

```
optimistic_outstanding = outstanding - (1 - liquidation_penalty) * (collateral_at_
↪ auction / current_minting_price)
```

4.2.5 How much collateral should we liquidate?

In general, in order to calculate how much should we auction off we assume that a) this auction is warranted, b) all pending auctions are also warranted, c) the price we'll get for everything is `current_minting_price`. a) and b) effectively mean that we can only expect $(1 - \text{liquidation_penalty})$ returns from all auctions considered. Formally:

- First, from auctioning we expect to get the current `minting_price`, so, if we send `collateral_to_auction` and `repaid_kit` is received, we have

$$\begin{aligned} \text{collateral_to_auction} &= \text{repaid_kit} * \text{minting_price} &<=> \\ \text{repaid_kit} &= \text{collateral_to_auction} / \text{minting_price} &(3) \end{aligned}$$

- Second, we assume that the auction is warranted (so we lose the liquidation penalty), thus

$$\text{actual_repaid_kit} = \text{repaid_kit} * (1 - \text{liquidation_penalty}) \quad (4)$$

- Third, we consider all pending auctions to be successful, using the current `minting_price`, but also warranted:

$$\begin{aligned} \text{optimistic_outstanding} &= \text{outstanding_kit} - (1 - \text{liquidation_penalty}) * \text{collateral_at_auction} / \text{minting_price} \\ &\quad (5) \end{aligned}$$

- Fourth, under the above conditions we aim to bring the burrow into a state where it's not overburrowed anymore, thus

$$\text{collateral} - \text{collateral_to_auction} \geq (\text{optimistic_outstanding} - \text{actual_repaid_kit}) * \text{fminting} * \text{minting_price} \quad (6)$$

Solving (3), (4), (5), and (6) the above gives us (7):

$$\begin{aligned} \text{collateral_to_auction} &\geq \\ & (\text{outstanding_kit} * \text{fminting} * \text{minting_price} \\ & - (1 - \text{liquidation_penalty}) * \text{fminting} * \text{collateral_at_auction} \\ & - \text{collateral}) \\ & / \\ & ((1 - \text{liquidation_penalty}) * \text{fminting} - 1) \end{aligned}$$

if $((1 - \text{liquidation_penalty}) * \text{fminting} - 1) > 0$.

Say the burrow has been touched and all its parameters are up to date. Concerning liquidation, we have the following cases:

Case 1: The burrow is not a candidate for liquidation

If (2) above holds then the burrow should not be liquidated. Send nothing to auctions and leave the burrow as is.

Case 2: The burrow is a candidate for liquidation

If (2) above does not hold, then the burrow should be liquidated. Either partially, completely, or even be closed.

First things first, the actor who initiated liquidation should get their reward (burrow creation deposit + percentage of collateral):

```
liquidation_reward = creation_deposit + (collateral * liquidation_reward_percentage)
```

That is, before we compute anything else, we leave the burrow with less collateral and without a creation deposit:

```
active      = false
collateral = collateral - (collateral * liquidation_reward_percentage)
```

Now, depending on how much collateral remains, we have the following cases:

Case 2A: `collateral < creation_deposit`

We cannot replenish the creation deposit.

- We send all the remaining collateral to be auctioned off for kit.
- The burrow remains deactivated.

```
collateral          = 0
collateral_at_auction = collateral_at_auction + collateral_to_auction
```

Case 2B: `collateral >= creation_deposit`

We can replenish the creation deposit, and this is the first thing we do:

```
collateral = collateral - creation_deposit
```

Now all that remains is to compute what should we auction off to bring the burrow to a state where “*any outstanding kits could have just been minted*”. For that, we use the (7):

```
collateral_to_auction = ceil (
  ( outstanding_kit * fminting * minting_price
    - (1 - liquidation_penalty) * fminting * collateral_at_auction
    - collateral
  )
  /
  ((1 - liquidation_penalty) * fminting - 1)
)
```

- If `collateral_to_auction < 0` or `collateral_to_auction > collateral`, then restoration is impossible: liquidate the entire remaining collateral (Note that the resulting burrow can be targeted for liquidation one last time (with the creation deposit being the only reward). Alternatively, we could (rather harshly) liquidate the deposit too and close the burrow.):

```
active      = true
collateral  = 0
collateral_at_auction = collateral_at_auction + collateral
```

- Otherwise auction off exactly `collateral_to_auction`:

```
active           = true
collateral       = collateral - collateral_to_auction
collateral_at_auction = collateral_at_auction + collateral_to_auction
```

4.2.6 Was the liquidation warranted?

We sent 10% extra tez to be auctioned off as a penalty, but in case the actual selling price of the tez would not have triggered a liquidation (retrospectively), we wish to bring that back to the burrow, if possible.

Calculations: In order to see whether liquidation should occur, we used equation (2) above, which we can rewrite as

```
liquidation_price <= collateral / (optimistic_outstanding * fliquidation)    (3)
```

So, if (3) was satisfied, we wouldn't have triggered a liquidation. If we assume that at the end we sent `collateral_to_auction` to be auctioned off and we received `repaid_kit` for it, we have:

```
maximum_non_liquidating_price = collateral / (optimistic_outstanding * fliquidation)
real_price                     = collateral_to_auction / repaid_kit      # derived from
↳ the auction outcome
```

If `real_price <= maximum_non_liquidating_price` then the liquidation was not warranted (i.e. the liquidation price we used when calculating `collateral_to_auction` was off) and we wish to return the kit we received from the auction in its entirety to the burrow:

```
real_price <= maximum_non_liquidating_price
collateral_to_auction / repaid_kit <= collateral / (fliquidation * optimistic_
↳ outstanding) <=>
collateral_to_auction * (fliquidation * optimistic_outstanding) <= repaid_kit *
↳ collateral <=>
collateral_to_auction * (fliquidation * optimistic_outstanding) / collateral <=
↳ repaid_kit <=>
repaid_kit >= collateral_to_auction * (fliquidation * optimistic_outstanding) /
↳ collateral
```

So, if the kit that the auction yields is more than

```
min_received_kit_for_unwarranted = collateral_to_auction * (fliquidation * optimistic_
↳ outstanding) / collateral
```

then this liquidation was unwarranted.

4.2.7 What if the liquidation was warranted?

When we send `collateral_to_auction` to an auction, we also send `min_received_kit_for_unwarranted` so that—after the auction is over—we can determine whether it was warranted. If it was warranted, then we wish to return the received kit in its entirety to the burrow. Otherwise we burn 10% of the kit earnings.

The auction logic might end up splitting `collateral_to_auction` into parts (slices) that can be sold for different prices; we perform the above check per slice.

```
collateral_to_auction = tez_1 + tez_2 + ... + tez_n
```

If we end up selling slice `tez_i` for `kit_i`, this part of the liquidation is considered unwarranted (and thus `kit_i` is returned to the burrow) only if

```
kit_i >= min_received_kit_for_unwarranted * (tez_i / collateral_to_auction) <=>
collateral_to_auction * kit_i >= min_received_kit_for_unwarranted * tez_i
```

4.2.8 Misc

- `fminting > fliquidation`
- `minting_price >= liquidation_price`
- `liquidation_penalty = 10%`

4.3 Liquidation Auctions

4.3.1 State

- `avl_storage`: data structure containing a mapping from pointers to auctions and liquidation slices, serving as a memory.
- `queued_slices`: a pointer to the queue of liquidation slices awaiting inclusion in an auction.
- `current_auction`: information about the current auction if there is an active auction.
 - `contents`: a pointer to the set of slices in the current auction.
 - `auction_state`: whether the auction is in the descending or ascending phase, and data used to calculate the current price.
- `completed_auctions`: a queue (represented as a doubly-linked list) of completed auctions, each auction containing:
 - a set of untouched slices
 - the result of an auction, containing the amount of tez sold, amount of kit gained and the winner of an auction.

At any point in time, any liquidation slice is in only one of the above sets, and they always move from `queued_slices`, to `current_auction` and then to `completed_auctions`, (then they disappear). Additionally, this move always happens in order, so an older `liquidation_slice` is always further in the process than a younger one.

NOTE: *Per-burrow liquidation_slices* We need to have access to the liquidation slices for a specific burrow; so slices for a burrow form a doubly-linked list, each burrow storing a pair of pointers called `liquidation_slices`, pointing to the first and the last liquidation slice of that burrow (if they exist).

See `</avl_diagram.drawio>` file for an illustration.

4.3.2 Initiating a liquidation

When liquidation of a burrow is triggered, the amount of tez to be liquidated form a `liquidation_slice`. * For details about this process, see `<./burrow-state-liquidations.md>`.

The new slice is added to the back of the `queued_slices` queue.

- NOTE: This operation also updates the per-burrow linked list.

4.3.3 Cancelling a liquidation slice

Burrows can cancel auctioning off their liquidation slices on certain conditions. When cancelling a slice, we check if the slice belongs to the `queued_slices`, if so, remove it from the set (returning contents back to the burrow). If not, the process fails.

- NOTE: This operation also updates the per-burrow linked list.
- NOTE: This requires a the queue to have an efficient membership test.
- NOTE: This requires a the queue to support efficient random deletes.

4.3.4 Lot auction

At any time checker is touched, when there is no auction running and there is at least one queued slice, we start an auction.

Our aim is to take a prefix of the `queued_slices` queue which contains exactly this amount of tez:

```
min
    total_queued_tez
(max
    Constant.max_lot_size
    (total_queued_tez * Constants.min_lot_auction_queue_fraction))
```

However, it is likely that in this process the slices will not add up to the exact amount. In this case, we take the `liquidation_slice` causing the overload, split it into two, and push the halves to the end of the new auction and in front of the `queued_slices`.

- NOTE: This splitting process has to be efficient, since a single auction likely consists of many small slices. So it needs to be done without traversing the entire prefix. This pretty much forces us to use a tree-like structure with branches containing the aggregate tez information of their sub-trees.

Then we start an auction. An auction has `minimum_bid` value that is a function of current time and the latest bid.

Every bid should be of at least `minimum_bid` amount of kit. The bidding process debits the bid's kit from the contract's kit ledger and credits back the kit of the previously winning bid if one exists.

The auction is initially a **descending** auction, with the minimum bid calculated as:

```
amount_of_tez_inside_auction
* tz_minting
* q
* ((1 - Constants.auction_decay_rate) ^ time_elapsed_since_auction_start)
```

After the first bid, it becomes an **ascending** auction, with the minimum bid calculated as:

```
leading_bid * (1+Constants.bid_improvement_factor)
```

The auction finishes when the longer of 20 blocks or 20 minutes are passed after the last bid.

4.3.5 Touching a liquidation_slice

“Touching the liquidation slice” is the process of propagating the result of a completed auction back to the burrows. When it is triggered, we:

1. Check if the given slice belongs to a completed auction, ignore otherwise.
2. Remove the slice from the contents of the relevant `completed_auction`.
3. Remove the slice from the linked list at relevant burrows `liquidation_slices`.

4.3.6 Claiming a winning bid

If a bid is the winning bid of a completed auction where all the `liquidation_slices` are touched (in other words, its contents are empty), the bidder can claim the auction’s winnings. This process is the final step of an auction, and after that the auction itself is cleaned up.

4.3.7 Maintenance

Every time the main checker contract is touched, it touches `Constants.number_of_slices_to_process` amount of oldest `liquidation_slice`’s automatically.

4.4 CFMM subsystem

NOTE: CFMM stands for *Constant Function Market Maker*. What this means is that when parties exchange kit for ctez and vice versa, using checker, checker tries to keep the product of kit and ctez within it unchanged (ignoring the fees of course).

This file gives an operational interpretation of the cfmm API inside the checker contract, and operations on it.

4.4.1 State

- `ctez`: the total amount of ctez currently held by the cfmm contract.
- `kit`: the total amount of kit currently held by the cfmm contract.
- `lqt`: the total amount of liquidity held by the cfmm contract.

Additional fields:

- `kit_in_ctez_in_prev_block`: the price of kit in ctez (kit / ctez) at the end of the previous block (as a ratio).
- `last_level`: the last block that the cfmm contract was touched on (as a nat).

NOTE 1: The reason we store `kit_in_ctez_in_prev_block` and `last_level` in the state of cfmm is security. When the price implied by cfmm is queried to compute the drift derivative (see `system-parameters.md`), we don’t want to give the current price, but instead return the last price at the end of the previous block. This makes it just a little harder to manipulate these small price fluctuations.

NOTE 2: `kit_in_ctez_in_prev_block` is always computed as the amount of kit divided by the amount of ctez, so it can never really grow too much in size. Hence we use a lossless rational for its representation.

4.4.2 Initialization

When the system starts, all parameters are set to the lowest non-zero amount. Given that Checker gets deployed on the chain at level `lvl`, we initialize the parameters thus:

```
ctez          = 1
kit           = 1
lqt          = 1
kit_in_ctez_in_prev_block = 1    # same as kit/ctez now
last_level    = lvl
```

Effectively, given that (a) no one can remove the first liquidity token and (b) how rounding works in the operations that follow, the contract will never be completely out of ctez, kit, or liquidity. So, setting the initial values to one removes the need for division-by-zero checks, and the first/non-first liquidity provider distinction that is e.g. adopted by Dexter. Of course, this price is only for the beginning, and it is expected that through trading it will eventually move closer to the *real* price.

4.4.3 General notes on the interfaces

- None of the interfaces below refers to prices. Instead, we pass inputs, and minimum and maximum expected values for things (e.g. kit, ctez, liquidity, or time). If the criteria cannot be met the operations fail. This agrees with e.g. the API offered by Dexter.
- All the following happen within the smart contract, which means that in the calculations below we often refer to `level` (the current block height), as well as `now` (the timestamp of the current block, as provided by this block's baker).

4.4.4 Adding liquidity

First things first: if `last_level < level`, it means that this is the first time that the cfmm contract is touched in this block, so we update `kit_in_ctez_in_prev_block` to the price observed now, and set `last_level` to the current height, so that we don't update `kit_in_ctez_in_prev_block` again in this block:

```
kit_in_ctez_in_prev_block = ctez/kit
last_level                = level
```

If `last_level = level`, then we don't perform the update; this is not the first time we've touched the cfmm contract in this block.

Inputs

- `ctez_amount`: The amount of ctez to be added to the cfmm contract.
- `max_kit_deposited`: The maximum amount of kit to be added to the cfmm contract.
- `min_lqt_minted`: The minimum amount of liquidity expected to be received.
- `deadline`: The deadline; starting from this timestamp the transaction can no longer be executed.

If any of the following holds, the transaction fails:

- If we are on or past the deadline (`now >= deadline`), the transaction fails.
- If no ctez is given (`ctez_amount = 0`), the transaction fails.
- If no kit is offered (`max_kit_deposited = 0`), the transaction fails.

- If no liquidity is to be added (`min_lqt_minted = 0`), the transaction fails.

So, we calculate the amount of liquidity to mint and the amount of kit that needs to be deposited using the ratio of the provided ctez vs. the ctez currently in the cfmm contract:

```
lqt_minted    = lqt * (ctez_amount / ctez)    # floor
kit_deposited = kit * (ctez_amount / ctez)    # ceil
```

Because of this calculation, we need to know that the pool of ctez is not empty, but this should be ensured by the initial setup of the cfmm sub-contract. Also

- If `lqt_minted < min_lqt_minted` then the transaction fails.
- If `max_kit_deposited < kit_deposited` then the transaction fails.
- If `kit_deposited = Kit.zero` then the transaction fails.

If all is good, we proceed with updating the parameters

```
kit  = kit + kit_deposited
ctez = ctez + ctez_amount
lqt  = lqt + lqt_minted
```

Note that the complete `ctez_amount` is consumed. However, `kit_deposited` might differ from `max_kit_deposited`. Hence, we return the leftovers:

```
kit_to_return = max_kit_deposited - kit_deposited
```

4.4.5 Removing liquidity

First things first: if `last_level < level`, it means that this is the first time that the cfmm contract is touched in this block, so we update `kit_in_ctez_in_prev_block` to the price observed now, and set `last_level` to the current height, so that we don't update `kit_in_ctez_in_prev_block` again in this block:

```
kit_in_ctez_in_prev_block = ctez/kit
last_level                 = level
```

If `last_level = level`, then we don't perform the update; this is not the first time we've touched the cfmm contract in this block.

Inputs

- `lqt_burned`: The amount of liquidity to be removed from the cfmm contract.
- `min_ctez_withdrawn`: The minimum amount of ctez to be received for the removed liquidity.
- `min_kit_withdrawn`: The minimum amount of kit to be received for the removed liquidity.
- `deadline`: The deadline; starting from this timestamp the transaction can no longer be executed.

If any of the following holds, the transaction fails

- If we are on or past the deadline (`now >= deadline`), the transaction fails.
- If no liquidity is to be removed (`lqt_burned = 0`), the transaction fails.
- If no ctez is expected to be received from this transaction (`min_ctez_withdrawn = 0`), the transaction fails.
- If no kit is expected to be received from this transaction (`min_kit_withdrawn = 0`), the transaction fails.

Otherwise, we compute how much ctez and kit should be returned, using the ratio of the provided liquidity vs. the liquidity currently in the cfmm contract:

```
ctez_withdrawn = ctez * (lqt_burned / lqt)    # floor
kit_withdrawn  = kit  * (lqt_burned / lqt)    # floor
```

Also, we check that the bounds are respected:

- If `ctez_withdrawn < min_ctez_withdrawn`, the transaction fails.
- If `ctez_withdrawn > ctez`, the transaction fails.
- If `kit_withdrawn < min_kit_withdrawn`, the transaction fails.
- If `kit_withdrawn > kit`, the transaction fails.
- If `lqt_burned > lqt`, the transaction fails.

If all is good, we proceed with updating the parameters

```
kit  = kit  - kit_withdrawn
ctez = ctez - ctez_withdrawn
lqt  = lqt  - lqt_burned
```

and return the withdrawn amounts:

```
ctez_to_return = ctez_withdrawn
kit_to_return  = kit_withdrawn
```

4.4.6 Buying Kit

First things first: if `last_level < level`, it means that this is the first time that the cfmm contract is touched in this block, so we update `kit_in_ctez_in_prev_block` to the price observed now, and set `last_level` to the current height, so that we don't update `kit_in_ctez_in_prev_block` again in this block:

```
kit_in_ctez_in_prev_block = ctez/kit
last_level                 = level
```

If `last_level = level`, then we don't perform the update; this is not the first time we've touched the cfmm contract in this block.

Inputs

- `ctez_amount`: The amount of ctez to be added to the cfmm contract.
- `min_kit_expected`: The minimum amount of kit to be bought.
- `deadline`: The deadline; starting from this timestamp the transaction can no longer be executed.

If any of the following holds, the transaction fails

- If the amount of ctez given is zero (`ctez_amount = 0`), the transaction fails.
- If we are on or past the deadline (`now >= deadline`), the transaction fails.
- If no amount of kit is expected (`min_kit_expected = 0`), the transaction fails.

Otherwise, we compute how much kit can be bought for the `ctez_amount` of ctez as follows:

```
price      = kit / ctez
slippage   = ctez / (ctez + ctez_amount)
kit_bought = ctez_amount * price * slippage * (1 - cfmm_fee)  # floor
```

Also, we check that the bounds are respected:

- If `kit_bought < min_kit_expected`, the transaction fails.
- If `kit_bought > kit`, the transaction fails.

If all is good, we proceed with updating the parameters

```
kit  = kit  - kit_bought
ctez = ctez + ctez_amount
```

and return the bought amount of kit:

```
kit_to_return = kit_bought
```

4.4.7 Selling Kit

First things first: if `last_level < level`, it means that this is the first time that the cfmm contract is touched in this block, so we update `kit_in_ctez_in_prev_block` to the price observed now, and set `last_level` to the current height, so that we don't update `kit_in_ctez_in_prev_block` again in this block:

```
kit_in_ctez_in_prev_block = ctez/kit
last_level                 = level
```

If `last_level = level`, then we don't perform the update; this is not the first time we've touched the cfmm contract in this block.

Inputs

- `kit_given`: The amount of kit to be sold to the cfmm contract.
- `min_ctez_expected`: The minimum amount of ctez to be bought.
- `deadline`: The deadline; starting from this timestamp the transaction can no longer be executed.

If any of the following holds, the transaction fails

- If the amount of kit given is zero (`kit_given = 0`), the transaction fails.
- If we are on or past the deadline (`now >= deadline`), the transaction fails.
- If no amount of ctez is expected (`min_ctez_expected = 0`), the transaction fails.

Otherwise, we compute how much ctez can be bought for the `kit_given` as follows:

```
price      = ctez / kit
slippage   = kit / (kit + kit_given)
ctez_bought = kit * price * slippage * (1 - cfmm_fee)  # floor
```

Also, we check that the bounds are respected:

- If `ctez_bought < min_ctez_expected`, the transaction fails.
- If `ctez_bought > ctez`, the transaction fails.

If all is good, we proceed with updating the parameters

```
kit  = kit  + kit_given
ctez = ctez - ctez_bought
```

and return the bought amount of ctez:

```
ctez_to_return = ctez_bought
```

NOTE: There are more than one ways to calculate things when buying and selling kit. Given that `da` amount of one quantity is given, what we do essentially computes first what should the return be for the product of quantities kept by `cfmm` to stay the same:

```
db = da * (b / (a + da))
```

and then keeps `fee` of that, thus returning `db` calculated instead like this:

```
db = da * (b / (a + da)) * (1 - fee)
```

Dexter takes an alternative approach, where the fee is (conceptually, at least) on the amount given. That is, the returned amount is

```
db = da' * (b / (a + da'))
```

where

```
da' = da * (1 - fee)
```

The two calculations give slightly different results, but hopefully that is not a problem.

4.4.8 Misc

- `cfmm_fee = 0.002`

DEPLOYING CHECKER

TBD

GLOSSARY

6.1 Kit

A coin / token created and destroyed as part of the system.

6.2 Burrow

A contract for a “deposit account” that supports a few operations, e.g. “minting” (ie. borrowing) kit, or “burning” (ie. repaying) kit. A fresh burrow contract is created for every depositor, and only Checker is allowed to access it, so operations on burrows are performed via the Checker contract.

6.3 Circulating kits

The number of kits that exist. See also: *outstanding kits*.

6.4 Outstanding kits

The number of kits that it would take to close all currently open burrows. See also: *circulating kits*.

6.5 Liquidation lot

A batch of *liquidation slices* currently being auctionned of.

6.6 Liquidation slice

Some amount of tez, tied to a burrow, which is inserted in the liquidation queue to be auctionned of for *kit*

6.7 Liquidation queue

A dequeue implemented as a balanced binary tree representing an ordered list of *liquidation slices*. Slices at the front of the queue are periodically batched into a *liquidation lot*

6.8 Imbalance

The ratio of the number of *circulating kits* to the number of *outstanding kits*.

6.9 Imbalance adjustment

A compounding fee or reward applied to *burrows* which implicitly increases or decreases the number of *outstanding kits* over time to bring it closer to the number of *circulating kits* so as to bring the *imbalance* closer to 1.

INDICES AND TABLES

- `genindex`