Simple Arbitrage Keeper

Overview

Typically in the form of an automated bot, Keepers are external actors that participate inmarket making ,auctions ,market arbitrage , and system upkeep within the Maker protocol and the greater Ethereum ecosystem. Under a pure economic assumption, they are solely incentivized by profits or a vested interest. However, their activity provides indirect services such as increased liquidity and price consistency across various financial markets. This guide presents a simple arbitrage keeper, and structure thereof, that can be used out of the box to participate in wildly volatile markets. The purpose of the guide is to motivate the user in improving its functionality to participate in less volatile markets, leading to a more efficient financial ecosystem.

Learning objectives

By reading this guide, you'll gain:

- Better understanding of Arbitrage, how it is achieved and the role it plays in financial markets
- An overview of the simple arbitrage keeper and common structure across the Maker Keeper framework
- Insight into the operation of the simple arbitrage keeper and ways to improve its code
-

Pre-requisites

- At least Python 3.6.2 and some python experience
- Linux, macOS, or Cygwin (on Windows)
- Some general experience with Ethereum Development (account management, contract interaction, node hosting)
-

Guide

Introduction to Arbitrage

Arbitrage is the process of simultaneously purchasing an asset on one exchange and selling a similar, if not identical, asset on another exchange at a higher price. Conceptually, this trading strategy can be broken down into two legs, the first buying leg and the second selling leg. As the timestamp of these legs converge, the risk of asset exposure approaches zero. For example, if both trades are not executed in perfect synchrony, there is a chance that the second trade is completed by another actor, forcing the arbitrageur to hold an asset until another opportunity arises.

With advancement in technology, this trading strategy is typically automated and can detect, as well as, profit from price deviations in a matter or seconds. Arbitrage opportunities exist because ofmarket inefficiencies , such as a temporary shortage of liquidity. As a result, arbitrage provides a mechanism to ensure prices on one exchange/market do not deviate far from thefair value .1

System Process Structure

?

Operation of the Simple Arbitrage Keeper

Thesimple-arbitrage-keeper is an arbitrage keeper for OasisDex and Uniswap. Given a minimum profit per trade and maximum trade size, as defined by the user and in units ofentry_token , the Keeper will scan for arbitrage opportunities. When running, it monitors thearb_token/entry_token price on both exchanges, and when a discrepancy is detected, it executes an atomic multi-step trade in a single Ethereum transaction. The multi-step trade is made up of two legs: the first leg is the purchasing ofarb_token with theentry_token on thestart_exchange , while the second leg is the selling of thearb_token with theentry_token on theend_exchange . The type ofentry_token andarb_token is typically of a stable and volatile type, respectively; since arbitrage opportunities can be scarce, the Keeper sometimes rests in a default state, holding theentry_token for extended periods of time as it looks for new opportunities. Exposure only to a stable token is especially important in this default state. If any leg of the trade fails, the entire trade reverts, thereby upholding the "risk-free" attribute of true arbitrage.

It inherits a system structure that is consistent with other keepers in the Maker Keeper Framework. Apart from libraries typical to python projects, it relies onpymaker andpyexchange , both of which are highly applicable Python APIs for the Maker Protocol Contracts and cryptocurrency exchanges, respectively. As can be seen in the flowchart, the 'Lifecycle box' is apymaker utility class that helps define the proper life cycle of a keeper; it consists of keeper startup, timers and/or subscriptions to Web3 events, and a keeper shutdown phase. The 'Process block' box is executed when the Keeper's node receives a new block. It generally contains the logic to query the blockchain state, evaluate any profit opportunities, and when applicable, publish a transaction.

The step for each process in simple-arbitrage-keeper is as follows:

1. Keeper Entry -> enter into SimpleArbitrageKeeper class
2. Startup -> make all necessary approvals between tokens, exchanges, and tx-manager
3. Evaluate Market -> pull price data from Oasis REST API and Uniswap exchange contracts
4. Profitable Opportunity Found? ->check if profit opportunity exists with a given max engagement and min profit
5. Execute multi-step trade -> with use of TxManager, buy asset on exchange A and sell on exchange B within one Ethereum transaction
6.

Documentation

General high level information and comments for each class/method are in the simple-arbitrage-keeper github repository .

Startup on Kovan

The usage of the bot can be found in the Usage section of the README. Here are some preparation steps in running on Kovan:

To operate the keeper, call the/bin/simple-arbitrage-keeper script with the required arguments. This adds the pyexchange and pymaker modules to the python path and initializes the SimpleArbitrageKeeper class. It's convenient to write a shell script that can be easily rerun after argument changes, but before we begin, let's prepare our Keeper ethereum address, deploy our TxManager, find the Uniswap exchange addresses and MatchingMarket address.

Keeper Installation

This project uses Python 3.6.2. In order to clone the project and install required third-party packages please execute:

```

Copy git clone https://github.com/makerdao/simple-arbitrage-keeper.git cd simple-arbitrage-keeper git submodule update --init --recursive pip3 install -r requirements.txt

```

For some known Ubuntu and macOS issues see the pymaker README .

Install the DappHub Toolkit

If you are running Linux or macOS you can take advantage of our all in one installer.

```

Copy curl https://dapp.tools/install | sh

```

If you're having issues with installation, head over to the 'Manual Installation' section

Keeper Ethereum Address

Your Keeper will publish transactions through this address. Your ethereum address needs to be accessible by seth , in order to deploy TxManager through dapp.tools . Furthermore, when the Keeper is deployed and in operation, this address must contain some ETH for transaction costs as well as an amount of entry-tokens equal to the max-engagement set by the User. Follow these instructions to ensure seth can see your keystore and passphrase files.

If you'd like to start from scratch, install geth and usegeth account new , to create a new account, set a passphrase, and place the passphrase into a

.txt file in the~/Library/Ethereum/passphrase/ directory; the corresponding keystore file can be found in the~/Library/Ethereum/keystore/ directory. Both the keystore and passphrase files will later be passed into the Keeper as an argument. Make a note of your Keeper Ethereum Address, its keystore file location and passphrase file location.

Deploy TxManager

A TxManager must be deployed and linked with the Keeper ethereum address in order to bundle multiple contract calls in a single ethereum transaction; this is what we call an atomic transaction, where the entire call set is successful, or the state remains unaffected. Once the DappHub Toolkit is installed, execute the following commands to deploy TxManager:

```

*Copy git clone https://github.com/makerdao/tx-manager.git cd tx-manager dapp update dapp --use solc:0.4.25 build --extract*
```

export SETH_CHAIN=kovan export ETH_FROM= export ETH_GAS=2000000 dapp create TxManager --password /path/to/passphraseOfAddress.txt

```

*The output ofdapp create is the address to your TxManager*

*Deploy/Find a parity node*

*You need access to a JSON-RPC host that's connected to a parity node. Unfortunately, Infura APIs are not connected to parity nodes, so you will need to find access to a remote endpoint (one example is[https://rivet.cloud/](https://rivet.cloud/) ) or run a node on your local machine. An example of a remote node could be[https://kovan.sampleparitynode.com:8545](https://kovan.sampleparitynode.com:8545)*

*Uniswap Exchange Addresses*

*You'll need to find the address to the Uniswap exchange corresponding to the entry-token as well as arb-token. To do that, call thegetExchange(TokenAddress) at the Uniswap Factory contract to query the exchange addresses.[You can do that here](here) . For example, you can use the Kovan WETH token address and callgetExchange(0xd0A1E359811322d97991E03f863a0C30C2cF029C)*

*Shell Script*

*As was mentioned before, let's make a shell script so that it's easy to run this Bot. For example, we're usingKovan Sai for the entry-token since we won't be exposed to the arb token's price exposure in between trades. Moreover, for kovan testing, we will be using the WETH/DAI pair, since liquidity is thin across the Kovan versions of MatchingMarket and Uniswap exchanges. AnyNoneType errors are likely due to a lack of liquidity on either exchange and can be resolved once liquidity is added. As mentioned earlier, the min-profit and max-engagement arguments define how the keeper will operate. Maximum engagement refers to the amount of entry-tokens you would like to sell within the first leg of the trade, whereas minimum profit refers to the amount of profit gained from executing the arbitrage; both arguments are denominated in entry-tokens and inunits of ether (in the below script, 1 SAI of min-profit would be 1 SAI). For example, the below script will trade with a max-engagement of 10 SAI, but only if the returned value from the trade results in at least 11 SAI.*

*Create a file called like below, insert the arguments relevant to your environment, make it executable, then run it!*

```

Copy vim run-simple-keeper-kovan.sh

```

*In your run-simple-keeper-kovan.sh file:*

```

Copy

!/bin/bash

*/full/path/to/githubClone/simple-arbitrage-keeper/bin/simple-arbitrage-keeper \ --rpc-host 'kovan.sampleparitynode.com' \ --eth-from '0xABC' \ --eth-key 'key_file=/path/to/keystore.file,pass_file=/path/to/passphrase.txt' \ --uniswap-entry-exchange '0x47D4Af3BBaEC0dE4dba5F44ae8Ed2761977D32d6' \ --uniswap-arb-exchange '0x1D79BcC198281C5F9B52bf24F671437BaDd3a688' \ --oasis-address '0x4A6bC4e803c62081ffEbCc8d227B5a87a58f1F8F' \ --oasis-api-endpoint 'https://kovan-api.oasisdex.com' \ --tx-manager '0xABC' \ --entry-token '0xC4375B7De8af5a38a93548eb8453a498222C4fF2' \ --arb-token '0xd0A1E359811322d97991E03f863a0C30C2cF029C' \ --arb-token-name 'WETH' \ --min-profit 1 \ --max-engagement 10 *

```

*Make your Keeper executable, and run it!*

```

Copy chmod +x run-simple-keeper-kovan.sh ./run-simple-keeper-kovan.sh

```

*Initial Startup*

*During the initial startup, several approval transactions will be sent to the tx-manager, and relevant exchanges. When complete, you'll see a “Watching for new blocks” message that indicates that the bot is now looping through thelifecycle.process_block() method; following that message and with every new block is a "Best Trade regardless of profit" message that acts as a "heartbeat" to the Keeper.*

```

Copy Kentons-Macbook:scripts kentonprescott ./run-simple-arbitrage-keeper-mainnet-XYZ-XYZ.sh 2019-10-19 17:21:57,463 INFO Keeper connected to RPC connection https://... 2019-10-19 17:21:57,463 INFO Keeper operating as 0xABCD 2019-10-19 17:21:58,523 INFO Executing keeper startup logic 2019-10-19 17:22:04,820 INFO Watching for new blocks 2019-10-19 17:22:13,983 INFO Best trade regardless of profit/min-profit: -1.243077085275947008 DAI from Uniswap to Oasis

```

*Nominal Operation*

*After the initial approval phase the Keeper's startup will look something like the below output and will continue to print "Best Trade ... " messages with every new block witnessed. When the keeper publishes a transaction, all relevant info will be logged and printed to the console. Finally, when you’d like to shut down the keeper, simply click CTRL-C, and it will gracefully shutdown.*

*Troubleshooting*

*During keeper operation, if the following error is shown, just CTRL-C and rerun:*

```

Copy ValueError: {'code': -32010, 'message': 'Transaction with the same hash was already imported.'}

```

*Next steps*

*Improvements*

*Here are some suggestions to improve the usability, versatility, and profitability of this keeper. We hope that everyone tinkers with the parameters, runs the keeper for their own benefit, and eventually upgrades the bot to meet their risk tolerance and profit appetite.*

- *Develop unit tests for each method*
- *Use the Uniswap Factory contract to query theuniswap-entry-exchange*
- *anduniswap-arb-exchange*
- *addresses rather than requiring it as an argument*
- *Rather than the default gas price implement a[dynamic gas price strategy]*
- *Increase scope*
  - *Monitor more than one pair*
  - *Monitor more than two decentralized exchanges*
  - *Implement support for centralized exchanges*
  - *Increase number of intermediary arb tokens*
    - *(e.x. DAI → WETH → BAT → DAI)*
    - *[https://math.stackexchange.com/a/94420]*
```

-
    -
        - https://www.dailycodingproblem.com/blog/how-to-find-arbitrage-opportunities-in-python/

- *
- Improve efficiency of TxManager
-
    - Do we really need to send all of our token balance to the contract during every atomic transaction?
- *
- Read the state of the MatchingMarket (OasisDex) contract rather than using the Oasis REST API
-
    - To save gas,get all active orders
-
    - andtake specific orders
-
    - by ID rather than useMatchingMarket.offer(...)
-
    - and the on-chain matching engine
- *
- Send trade updates through a text/email Python API
-

Resources

- https://github.com/makerdao/market-maker-keeper
- https://github.com/makerdao/arbitrage-keeper
- https://github.com/makerdao/tx-manager
- https://github.com/makerdao/pymaker
- https://github.com/makerdao/pyexchange
-

Previous Cage Keeper Next Chief Keeper Last updated4 years ago On this page *Overview * Learning objectives * Pre-requisites * Guide * Troubleshooting * Next steps * Resources

Export as PDF