Lesson 14 - Maths libraries / Stablecoins

“There are two ways of constructing a software design:
One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.
The first method is far more difficult.” -C.A.R. Hoare

> #### Maths Libraries in Solidity
>
> Background [article] series (https://medium.com/coinmonks/math-in-solidity-part-1-numbers-384c8377f26d)
> The series covers
>
> - Number formats
> - Overflow
> - Percentages
> - Compound Interest
> - Logs and exponentiation
>
> TLDR
> Solidity, the primary language for smart contract development, does not go much further, in terms of math, then just exposing what EVM opcodes are able to do.
>
> Libraries try to cover what basic language misses, but suffer from the absence of standardised number formats.
>
> ---

PRB Maths Library

See repo
See article
Smart contract library for advanced fixed-point math that operates with signed 59.18-decimal fixed-point and unsigned 60.18-decimal fixed-point numbers.

Features

- Operates with signed and unsigned denary fixed-point numbers, with 18 trailing decimals
- Offers advanced math functions like logarithms, exponentials, powers and square roots
- Gas efficient, but still user-friendly
- Bakes in overflow-safe multiplication and division
- Reverts with custom errors instead of reason strings
- Well-documented via NatSpec comments
- Thoroughly tested with Hardhat and Waffle

PRBMath comes in four flavours: basic signed, typed signed, basic unsigned and typed unsigned.

You get these functions

- Absolute
- Arithmetic and geometric average
- Exponentials (binary and natural)
- Floor and ceil
- Fractional
- Inverse
- Logarithms (binary, common and natural)
- Powers (fractional number and basic integers as exponents)
- Multiplication and division
- Square root

In addition, there are getters for mathematical constants:

- Euler's number
- Pi
- Scale (1e18, which is 1 in fixed-point representation)

Installation

With yarn:

```
yarn add @prb/math
```

Or npm:

```
npm install @prb/math
```

Other Libraries

Uniswap Full Maths

Working with Signatures

Signing and verifying using ECDSA

ECDSA signatures consist of two integers: `r` and `s`.
Ethereum also uses an additional `v` (recovery identifier) variable.
The signature can be defined as `{r, s, v}`.

To create a signature you need the message to sign and the private key (d_a) to sign it with.
The “simplified” signing process looks something like this:

1. Calculate a hash (e) from the message to sign.
2. Generate a secure random value for k.
3. Calculate point (x_1, y_1) on the elliptic curve by multiplying k with the G constant of the elliptic curve.
4. Calculate $r = x_1 \bmod n$. If r equals zero, go back to step 2.
5. Calculate $s = k^{-1}(e + rd_a) \bmod n$. If s equals zero, go back to step 2.

In Ethereum in step 1 we usually add
`Keccak256("\x19Ethereum Signed Message:\n32"`
to the beginning of the hashed message.

To verify the message you need

- the original message
- the address associated with the private key
- the signature `{s,r,v}`

v is either 27 or 28 in Bitcoin and Ethereum before EIP 155, since then, the chain ID is used in the calculation of v, to give protection against replaying transactions
$v = \{0,1\} + CHAIN\_ID * 2 + 35$

Why do we need the v value ?
There can be up to 4 different points for a particular x coordinate modulo n
2 because each X coordinate has two possible Y coordinates (reflection in x axis), and 2 because r+n may still be a valid X coordinate
The v value is used to determine one of the 4.

From the yellow paper
Recovery :
$ECDSARECOVER(e , v , r , s) \equiv pu$
Where pu is the public key, assumed to be a byte array of size 64
e is the hash of the transaction, h(T).
v, r, s are the values taken from the signature as above.

EIP712 and EIP2612

EIP712 a standard for signing transactions and ensuring that they are securely used

Domain separator (for an ERC-20)

```
bytes32 eip712DomainHash = keccak256(
    abi.encode(
        keccak256(
            "EIP712Domain(string name,string version,
            uint256 chainId,address verifyingContract)"
        ),
        keccak256(bytes(name())), // ERC-20 Name
        keccak256(bytes("1")),    // Version
        chainid(),
        address(this)
    )
);
```

Permit Function

"Arguably one of the main reasons for the success of ERC-20 tokens lies in the interplay between `approve` and `transferFrom`, which allows for tokens to not only be transferred between externally owned accounts (EOA), but to be used in other contracts under application specific conditions by abstracting away `msg.sender` as the defining mechanism for token access control.

However, a limiting factor in this design stems from the fact that the ERC-20 `approve` function itself is defined in terms of `msg.sender`. This means that user's *initial action* involving ERC-20 tokens must be performed by an EOA .
If the user needs to interact with a smart contract, then they need to make 2 transactions (`approve` and the smart contract call which will internally call `transferFrom`). Even in the simple use case of paying another person, they need to hold ETH to pay for transaction gas costs."

The permit function is used for any operation involving ERC-20 tokens to be paid for using the token itself, rather than using ETH.
See EIP2612

EIP2612 adds the following to ERC20

```
function permit(address owner, address spender,
    uint value, uint deadline, uint8 v, bytes32 r, bytes32 s) external
function nonces(address owner) external view returns (uint)
function DOMAIN_SEPARATOR() external view returns (bytes32)
```

A call to `permit(owner, spender, value, deadline, v, r, s)` will set `approval[owner][spender]` to `value`,
increment `nonces[owner]` by 1, and
emit a corresponding `Approval` event,
if and only if the following conditions are met:

- The current blocktime is less than or equal to `deadline`.
- `owner` is not the zero address.
- `nonces[owner]` (before the state update) is equal to `nonce`.
- `r`, `s` and `v` is a valid `secp256k1` signature from `owner` of the message:

If any of these conditions are not met, the `permit` call must revert.

Traditional Process = Approve + TransferFrom

The user sends a transaction which will approve tokens to be used via the UI.
The user pays the gas fee for this transaction
The user submits a second transaction and pays gas again.

Permit Process

User signs the signature — via Permit message which will sign the approve function.
User submits signature. This signature does not require any gas — transaction fee.
User submits transaction for which the user pays gas, this transaction sends tokens.

Original introduced by Maker Dao

In more detail

Taken from permit article

1. Our contract needs a domain hash as above and to keep track of nonces for addresses

2. We need a permit struct

```
bytes32 hashStruct = keccak256(
    abi.encode(
        keccak256("Permit(address owner,address spender,
        uint256 value,uint256 nonce,uint256 deadline)"),
        owner,
        spender,
        amount,
        nonces[owner],
        deadline
    )
);
```

This struct will ensure that the signature can only used for

the permit function
to approve from owner
to approve for spender
to approve the given value
only valid before the given deadline
only valid for the given nonce
The nonce ensures someone can not replay a signature, i.e., use it multiple times on the same contract.

We can then put these together

```
bytes32 hash = keccak256(
    abi.encodePacked(uint16(0x1901), eip712DomainHash, hashStruct)
);
```

On receiving the signature we can verify with

```
address signer = ecrecover(hash, v, r, s);
require(signer == owner, "ERC20Permit: invalid signature");
require(signer != address(0), "ECDSA: invalid signature");
```

We can then increase the nonce and perform the approve

```
nonces[owner]++;
_approve(owner, spender, amount);
```

Example from Solmate

Draft Example from Open Zeppelin

```
abstract contract ERC20Permit is ERC20, IERC20Permit, EIP712 {
    using Counters for Counters.Counter;

    mapping(address => Counters.Counter) private _nonces;

    // solhint-disable-next-line var-name-mixedcase
    bytes32 private immutable _PERMIT_TYPEHASH =
        keccak256("Permit(address owner,address spender,uint256 value,
        uint256 nonce,uint256 deadline)");

    /**
     * @dev Initializes the {EIP712} domain separator using the `name` parameter,
     * and setting `version` to `"1"`.
     *
     * It's a good idea to use the same `name` that is defined
     * as the ERC20 token name.
     */
    constructor(string memory name) EIP712(name, "1") {}

    /**
     * @dev See {IERC20Permit-permit}.
     */
    function permit(
        address owner,
        address spender,
        uint256 value,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) public virtual override {
        require(block.timestamp <= deadline, "ERC20Permit: expired deadline");

        bytes32 structHash = keccak256(abi.encode(_PERMIT_TYPEHASH, owner,
        spender, value, _useNonce(owner), deadline));

        bytes32 hash = _hashTypedDataV4(structHash);

        address signer = ECDSA.recover(hash, v, r, s);
        require(signer == owner, "ERC20Permit: invalid signature");

        _approve(owner, spender, value);
    }
```

Malleability

What can be done, with a signature sig for a particular message m and public key X we can find a new public key X' and a different message m' where this signature remains valid for this different message.
However this wouldn't give us the corresponding private key.

So even though we can generate a valid signature for a particular combination of message and public key, without the corresponding private key exploiting this is difficult.

Articles about signature malleability
Derpturkey
Coder's Errand

Discussion of possible exploit

Miscellaneous Solidity Tips

CREATE and CREATE2

CREATE gives the address that a contract will be deployed to

```
keccak256(rlp.encode(deployingAddress, nonce))[12:]
```

CREATE2 introduced in Feb 2019

```
keccak256(0xff + deployingAddr + salt + keccak256(bytecode))[12:]
```

Contracts can be deleted from the blockchain by calling selfdestruct.

selfdestruct sends all remaining Ether stored in the contract to a designated address.

This can be used to preserve the contract address among upgrades, but not its state. It relies on the contract calling selfdestruct then being re deployed via CREATE2

Seen as 'an abomination' by some
See Metamorphic contracts
and
Abusing CREATE2 with Metamorphic Contracts

CREATE3

See Repo and Repo2 and EIP 3171

Adds a CREATE3 opcode that does not include the init code hash, which is difficult to use on-chain to compute addresses in combination with constructor arguments.

The rationale for including the init code hash in the CREATE2 implementation is that it forces the address to contain specific bytecode. However it is not useful for cases where you only care that the contract was deployed by a specific factory account, with certain arguments (encoded in the salt), which is sufficient proof that the contract bytecode is as expected in many use cases.

I.e., if you want to compute the address of a contract with constructor arguments created via CREATE2 on-chain, you have to include the entire bytecode of the contract for which you are computing the address (instead of just the hash) to get the preimage of the contract bytecode.

SLOAD2 and SSTORE2

Repo and SStore2

SLOAD2 is a set of Solidity libraries for writing and reading contract storage paying a fraction of the cost, it uses contract code as storage, writing data takes the form of contract creations and reading data uses EXTCODECOPY.

Features

- All SLOAD2 storages are write-once only
- Key Value storage (custom key and auto-gen key)
- Cheaper storage reads (vs SLOAD) after 32 bytes
- Cheaper storage writes (vs SSTORE) after 32 bytes (auto-gen key)
- Cheaper storage writes (vs SSTORE) after 96 bytes (custom key)
- Use strings as keys
- Use bytes32 as keys
- Use address as keys (auto-gen)

Also version in solmate Repo

Bitshifting

Operators

- `&` Bitwise AND
  Performs an AND on each of the bits in the 2 arguments

- `|` BitWise OR
  Performs an OR on each of the bits in the 2 arguments

- `^` Bitwise XOR
  Performs an XOR on each of the bits in the 2 arguments

- `~` Bitwise Not
  Performs an NOT on each of the bits in the 2 arguments

- `<<` Left Shift
  Shifts all bits of the first operand to the left by the number of places specified by the second operand

- `>>` Right Shift
  Shifts all bits of the first operand to the right by the number of places specified by the second operand

Miscellaneous Tips

Solidity Best Practices / Tips & Tricks

Tips From @controlcthenv

- Don't initialise default values
- Before Using Yul, Verify YOUR assembly is better than the compiler's
- Overwrite new values onto old ones you're not using when possible
  Solidity doesn't garbage collect, write new values onto old unused ones to conserve memory and storage.
- Keep Data in Calldata where possible

  Since Calldata has already been paid for with the transaction, if you don't modify a parameter to a function, then don't bother copying the function to memory and just read the value from calldata.
- View Solidity Compiler Yul Output

  If you want to see what your Solidity is doing under the hood, just add `-yul` and `-ir` to your solc compiler options. Very helpful to see how your code is working, see if you order operations unsafely, or just see how Solidity is beating your Yul in gas usage.
- Get to know the compiler optionsSolc Compiler Options
- Using Vanity Addresses with lots of leading zeroes

  Why? Well if you have 2 addresses - 0x000000a4323... and 0x0000000000f38210 because of the leading zeroes you can pack them both into the same storage slot, then just prepend the necessary amount of zeroes when using them. This saves you storage when doing things such as checking the owner of a contract.
- Using Sub 32 Byte values doesn't always save gas

  Sub 32 byte values can save gas in the event of packing, but note that they require extra gas to decode and should be used on a case-by-case basis.

- Writing to an existing Storage Slot is cheaper than using a new one

  Credit - @libevm

  EIP - 2200 changed a lot with gas, and now if you hold 1 Wei of a token it's cheaper to use the token than if you hold 0. There is a lot to unpack here so just google EIP 2200 and learn if you want, but in general, if you need to use a storage slot, don't empty it if you plan to re-fill it later. Goes for all Yul+ and Yul contracts when managing memory.

- Tip - Int's are more expensive the more the leading bits

  Credit - @transmissions11
  Explained here

- Using iszero() in a lot of places because the compiler is smart
  Credit - @transmissions11
  He explains it very well here but because the compiler knows how to optimize, putting it before some pieces of logic can end up reducing overall gas costs, so test out inserting it before JUMP opcodes.

  `if iszero(x)` is actually cheaper than `if x` because of how JUMPI works!

  For more info: `if (X > 0)` is translated into `iszero(gt(X, 0))`. The optimizer translates that into `iszero(iszero(iszero(X))`

  Also see here

- Use Gas() when using call() in Yul
  Credit - @libevm
  When using call() in Yul you can avoid manually counting out all the gas you need to perform the call, and just forward all available gas via using gas() for the gas parameter.

- Store Storage in Code
  Credit - @boredGenius
  So Zefram's blog explains this well, but you can save gas by deploying what you want to store in a new contract, and deploying that contract, then reading from that address. This adds a lot of complexity to code but if you need to cut costs and use SLOAD a lot, I recommend looking into SLOAD2.

- Half of the Zero Address Checks in the NFT spec aren't necessary
  Credit - @transmissions11

  Launching a new NFT collection and looking to cut minting and usage costs? Use Solmate's NFT contracts, as the standard needlessly prevents transfers to the void, unless someone can call a contract from the 0 address, and the 0 address has unique permissions, you don't need to check that the caller isn't the 0 address 90% of the time.

- If it can't overflow without uint256(-1) calls, you don't need to check for overflow

  Save gas and avoid safemath with unchecked {} , this one is Soldity only but I wanted to include it, I was tired of seeing counters using Safemath, it is cost-prohibitive enough to call a contract billions of times to prevent an attack.

- If you are testing in Production, use Self-Destruct and Factory patterns for an Upgradeable contract

Credit - @libevm

Using a technique explained in this Twitter thread you can make it easily upgradeable and testable contracts with re-init and self-destruct. This mostly applied to MEV but if you are doing some cool factory-based programming it's worth trying out.

- Fallback Function Calls are cheaper than regular function calls

The Fallback function (and Sig-less functions in Yul) save gas because they don't require a function signature to call, for an example implementation I recommend looking at @libevm's subway which utilize's a sig-less function

---

- Pack Structs in Solidity

  Struct packing article here

  A basic optimization but important to know, structs should be organized so that they sequentially add up to multiples of 256 bits in size. So uint112 uint112 uint256 vs uint112 uint256 uint112

  Saves read operations needed to get a value

- Making Solidity Values Constant Where Possible

  They are replaced with literals at compile time and will prevent you from having to a read a value from memory. For writing Yul - replace all known values and constant values with literals to save gas and comment what they are.

- Solidity Modifiers Increase Code Size, So sometimes make them functions

  When using modifiers, the code of the modifiers is inserted at the start of the function at compile time, which can massively ballon code size. So sometimes it makes sense to make a modifier a function call instead, as only the function call will be inserted at the start of the function.

- Trustless calls from L2 to L2 exist, and can be very useful for L2 based DAO's

  Credit - Optimism and Arbitrum teams

  The OVM and ArbOS have built-in functions on contract calls from L1 to L2 to verify msg.sender and vice versa. Therefore if you make an L1 contract that can only be called by a trusted party on one L2 before calling another L2, you can create a trustless bridge. Recommend reading about Hop for this, but a cool design choice for DAO building.

- Use bitmaps, From bitmaps

For example, rather than

```
        // people who showed up: 1, 3, 4, 8, 9
        uint8[] memory a = new uint8[](1, 0, 1, 1, 0, 0, 0, 1, 1, 0);
```

we can do

```
        // people who showed up: 1, 3, 4, 8, 9
        uint16 a = 397; // equals 0110001101 in binary
```

We can read the bits with

```
        uint16 a = 397; // equals 0110001101 in binary

        // Read bits at positions 3 and 7.
        // Note that bits are 0-indexed, thus bit 1 is at position 0, bit 2 is at
position 1, etc.
        uint8 bit3 = a & (1 << 2)
        uint8 bit7 = a & (1 << 6)
```

- For admin roles
  Use a multi-sig such as gnosis that is time-locked

- Gas Refunds
  Some opcodes can trigger gas refunds, which reduces the gas cost of a transaction. However the gas refund is applied at the end of a transaction, meaning that a transaction always need enough gas to run as if there was no refunds. The amount of gas that can be refunded is also limited, to half of the total transaction cost before the hardfork London, otherwise to a fifth. Starting from the hardfork London also, only SSTORE may trigger refunds. Before that, SELFDESTRUCT could also trigger refunds.

- Details about the optimiser
  https://docs.soliditylang.org/en/latest/internals/optimizer.html#optimizer

  Solidity has 2 optimiser modules

  - at opcode level
    - Tries to simplfy expressions
    - Can inline functions, which may help it simplify bytecode
    - uses a rule list :
      https://github.com/ethereum/solidity/blob/develop/libevmasm/RuleList.h
    - removes duplicates
    - removes dead code
  - Yul IR code level - more powerful as it can operate across function calls.
    - Functions have fewer side effects so easier to judge if independent
    - Can remove functions that are multiplied by zero
    - Can reorder functions

- Optimiser - number of runs
  From the docs
  "The number of runs (`--optimize-runs`) specifies roughly how often each opcode of the deployed code will be executed across the life-time of the contract.
  This means it is a trade-off parameter between code size (deploy cost) and code execution cost (cost after deployment). A “runs” parameter of “1” will produce short but expensive code. In contrast, a larger “runs” parameter will produce longer but more gas efficient code. The maximum value of the parameter is `2**32-1`."

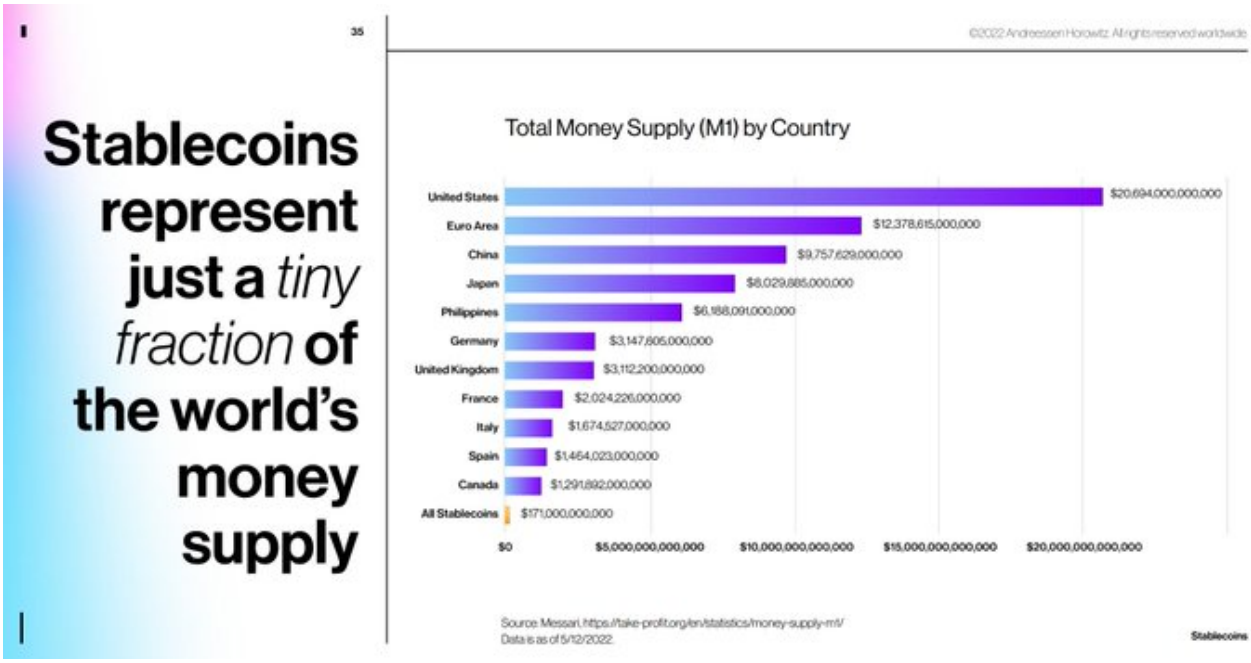
Testing tips

When you send ETH or tokens, don't just hardcode a fixed amount, add some random values as well, so you may catch some unpredictable behaviors depending on the amount.

Solidity Gas golf

https://g.solidity.cc/

Stablecoins

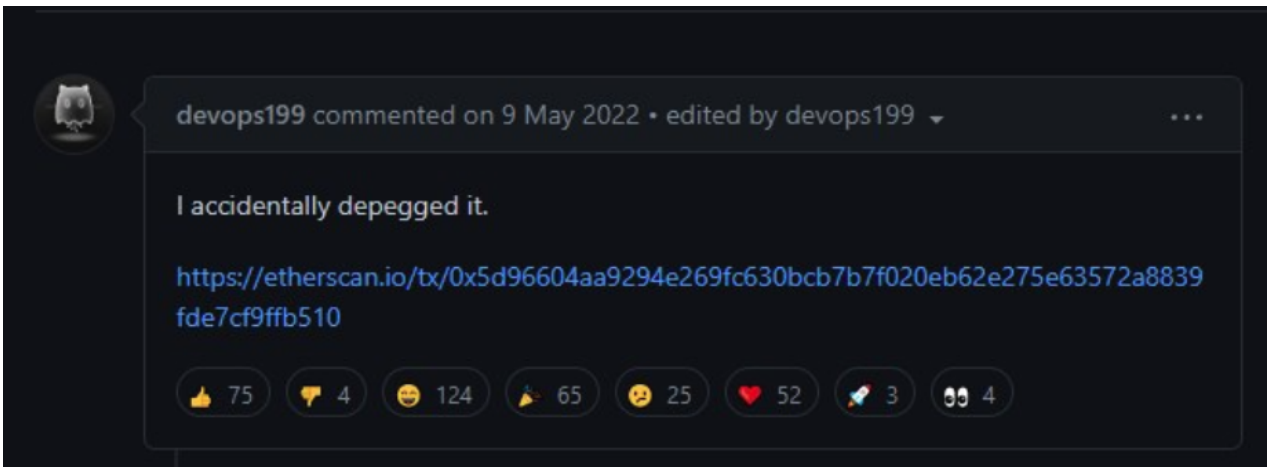


This screenshot from May 12th shows that the problems of maintaining a peg is widespread

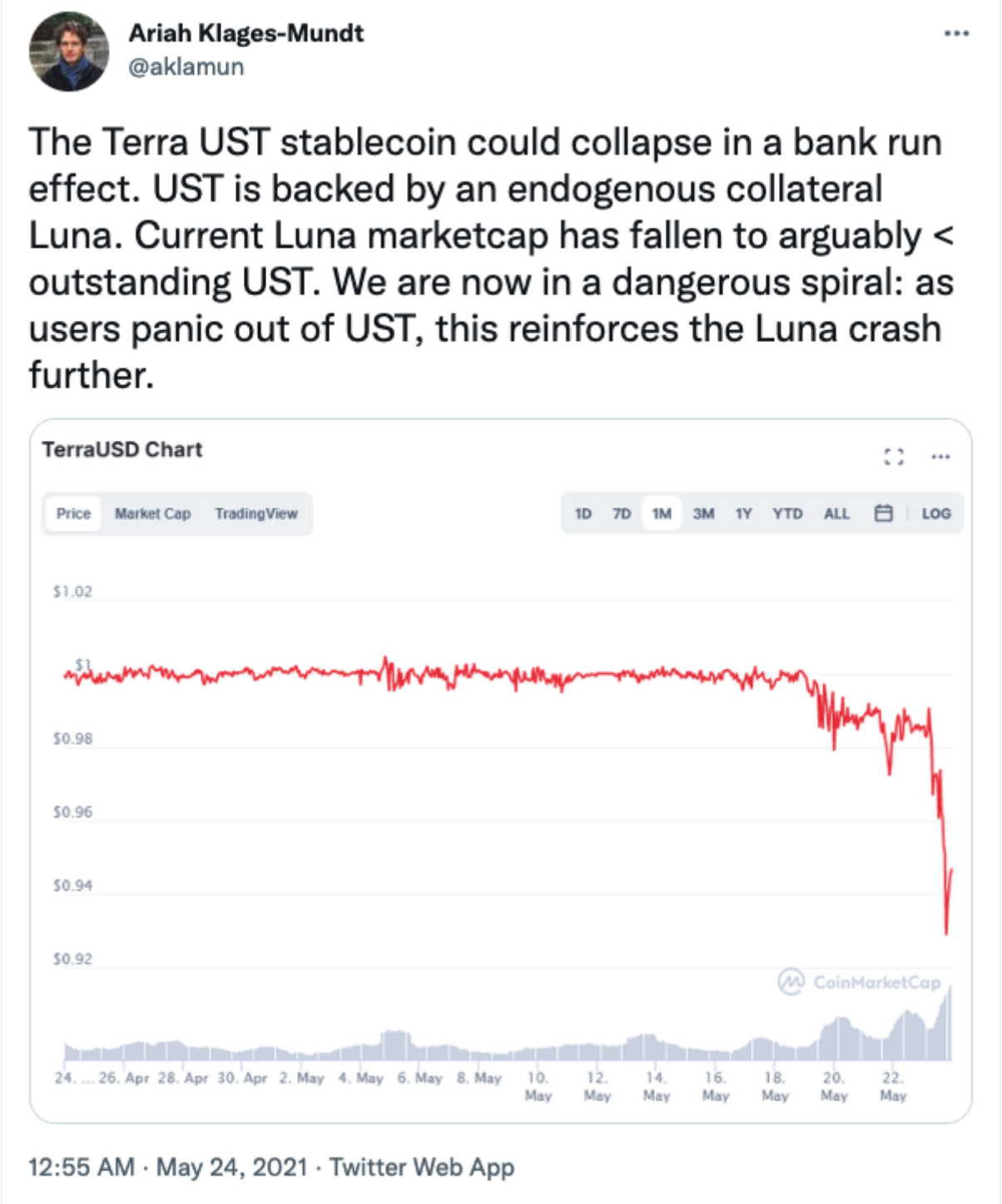
#	Coin		Price	24h Volume	Exchanges	Market Capitalization
☆ 1	Tether	USDT	$0.997248	$175,736,343,052	333	$82,039,445,114
☆ 2	USD Coin	USDC	$1.01	$30,304,157,484	283	$49,033,893,924
☆ 3	Binance USD	BUSD	$1.00	$20,307,488,741	107	$16,791,171,922
☆ 4	Dai	DAI	$1.01	$4,023,465,626	189	$6,009,272,965
☆ 5	TerraUSD	UST	$0.556812	$5,272,643,453	47	$6,015,925,233
☆ 6	Frax	FRAX	$0.990799	$381,409,099	20	$1,964,011,559
☆ 7	Magic Internet Money	MIM	$0.994833	$303,426,034	31	$1,819,839,406
☆ 8	TrueUSD	TUSD	$1.00	$1,047,431,622	73	$1,295,754,437
☆ 9	Pax Dollar	USDP	$1.00	$1,160,375,584	36	$978,199,048
☆ 10	Neutrino USD	USDN	$0.761652	$47,257,926	8	$706,346,413


Tether to USD Chart
Price
Market Cap
TradingView
1D
7D
1M
3M
1Y
YTD
ALL
LOG
83.0B
82.0B
81.0B
80.0B
79.0B
78.0B
77.0B
76.0B
75.0B
74.0B
73.0B
CoinMarketCap
USD
18
Apr
7
14

UST Crash


devops199 commented on 9 May 2022 • edited by devops199
I accidentally depegged it.
https://etherscan.io/tx/0x5d96604aa9294e269fc630bcb7b7f020eb62e275e63572a8839fde7cf9ffb510
75
4
124
65
25
52
3
4

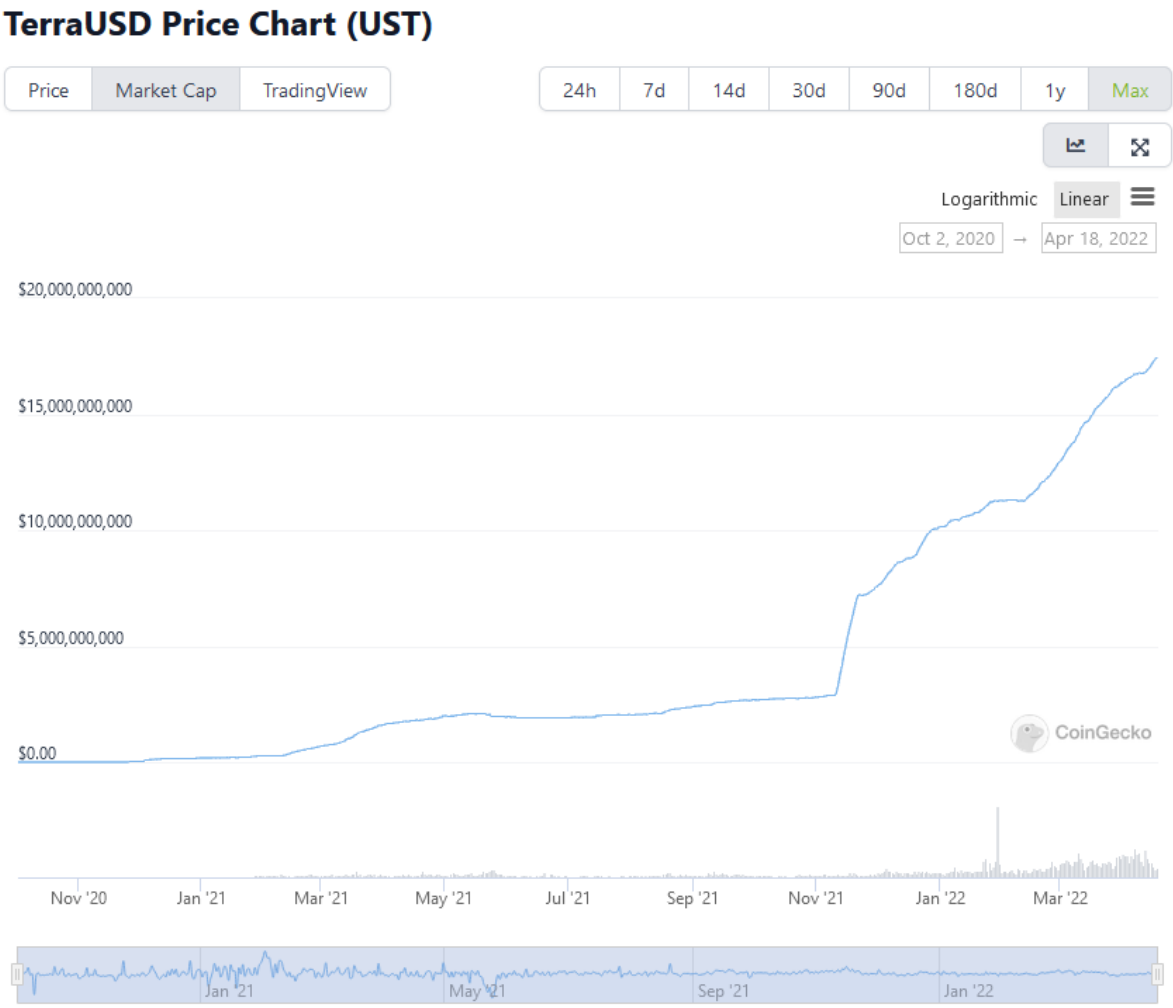

Is UST in a DEATH SPIRAL today?
@UstDeathSpiral
Daily tweets to see if $UST is in a death spiral.
Hell
Joined March 2022
20 Following
1,603 Followers
Not followed by anyone you're following
Tweets
Tweets & replies
Media
Likes
Is UST in a DEATH SPIRAL today? @UstDeathSpiral · May 12
YES
99
374
2,248
Tip

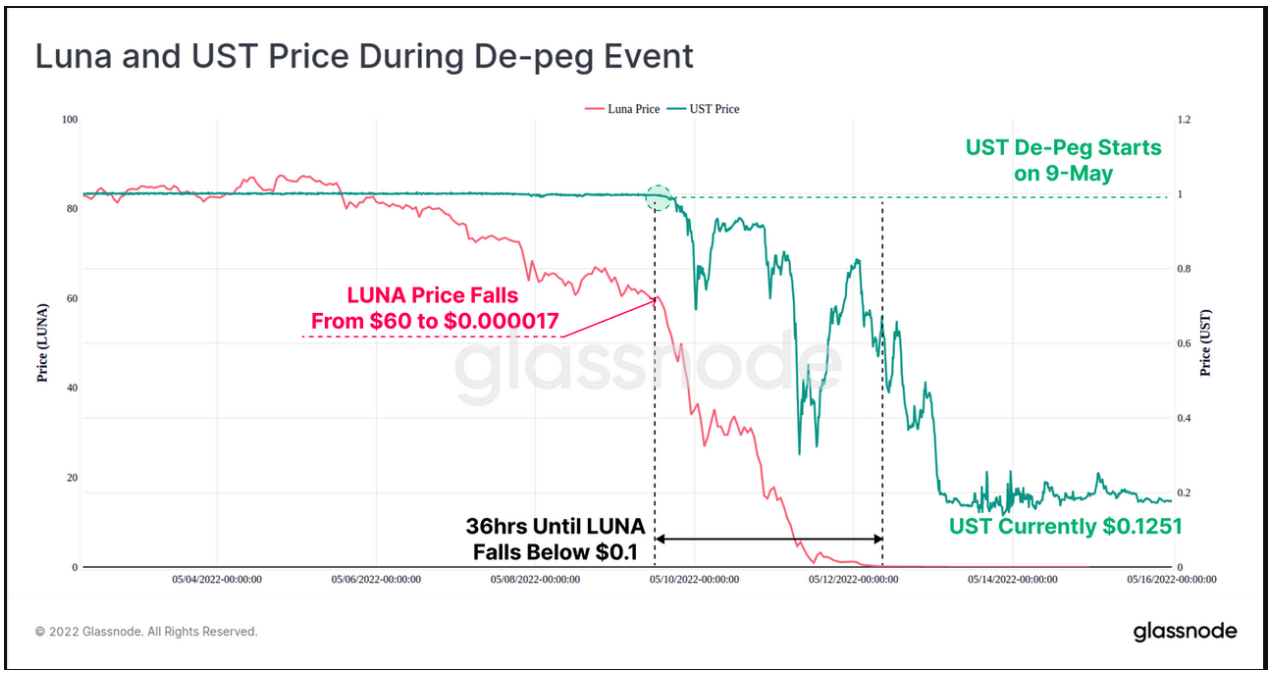


From Decrypt article
LUNA, formerly a top 10 coin by market cap, fell 100% to fraction of a fraction of a cent, and UST, designed to stay pegged at $1, bottomed out at 13 cents.

Before the crash

On April 18, UST flipped Binance USD to become the third-largest stablecoin on the market after Tether and USD Coin based on market capitalization.


TerraUSD Price Chart (UST)
Price
Market Cap
TradingView
24h
7d
14d
30d
90d
180d
1y
Max
Logarithmic
Linear
Oct 2, 2020
Apr 18, 2022
$20,000,000,000
$15,000,000,000
$10,000,000,000
$5,000,000,000
$0.00
CoinGecko
Nov '20
Jan '21
Mar '21
May '21
Jul '21
Sep '21
Nov '21
Jan '22
Mar '22


Luna and UST Price During De-peg Event
Luna Price
UST Price
UST De-Peg Starts on 9-May
LUNA Price Falls From $60 to $0.000017
36hrs Until LUNA Falls Below $0.1
UST Currently $0.1251
Price (LUNA)
Price (UST)
05/04/2022-00:00:00
05/06/2022-00:00:00
05/08/2022-00:00:00
05/10/2022-00:00:00
05/12/2022-00:00:00
05/14/2022-00:00:00
05/16/2022-00:00:00
© 2022 Glassnode. All Rights Reserved.
glassnode

Background

UST is a fully algorithmic stablecoin, intended to maintain stability through a 1:1 mint and redeem mechanism.

Here's how it works.

1. To mint UST, a user must burn an equal amount of LUNA (e.g. burn $1 of LUNA to mint $1 of UST).
2. To exit the system, a user can redeem their UST for a commensurate amount of LUNA.

This means that when the stablecoin is trading above peg, i.e. greater than $1, arbitrageurs are incentivized to come in and capture the difference between its current price and $1 by burning LUNA, increasing the supply of UST and lowering its price.

This same mechanism is also employed when LUNA is trading below $1. In that scenario, an arbitrageur can redeem UST for $1 of LUNA, similarly capturing the difference between the market price of UST and its intended peg, in doing so increasing the price of UST through decreasing its supply.

Anchor protocol

May 07 - $16.7B TVL

From Decrypt article 23rd April 2022https://decrypt.co/98482/we-need-to-talk-about-terras-anchor

Anchor, Terra's most popular project, is basically a high-interest savings account for TerraUSD (UST) stablecoins.
You can earn a steady rate of 19.46% for deposits.
More than 72% of all UST is currently deposited in Anchor.
This frenzied, Anchor-driven demand for UST is also part of the reason behind the stablecoin's blazing expansion and perhaps the excellent price action of Terra's native LUNA token.
In November, UST's market cap was just $2.73 billion—now it's $17.8 billion. During that same time, LUNA has also doubled from a price of roughly $50.

The interest rate however was designed to fall 1.5 percentage points each month if there were more lenders than borrowers on the platform

Once it became clear that that 20% interest wasn't going to last, UST holders began to leave.

On Friday, May 6, there was roughly 14 billion UST in Anchor. By Sunday, this figure was 11.7 billion.
UST was still pretty much pegged to the dollar at that point, which means roughly $2.3 billion in capital left over the course of that weekend.

To exit UST, you have two options.

1. Burn-and-mint mechanism.

   This mechanism lets holders swap 1 UST for $1 of LUNA, destroying the UST in the process. This creates an arbitrage opportunity whenever 1 UST falls below $1, as speculators can buy the discounted UST and trade it in for $1 in LUNA, making a small profit. The opposite is also true: If UST trades above $1, you can swap (and burn) $1 of LUNA for that UST.

2. Using the stablecoin exchange Curve Finance.

   Typically, when a stablecoin faces a minor price change, savvy arbitragers will head over to DeFi's deepest liquidity pools on Curve and trade the discounted stablecoin to whichever alternative has held its peg.

How these have worked in practice

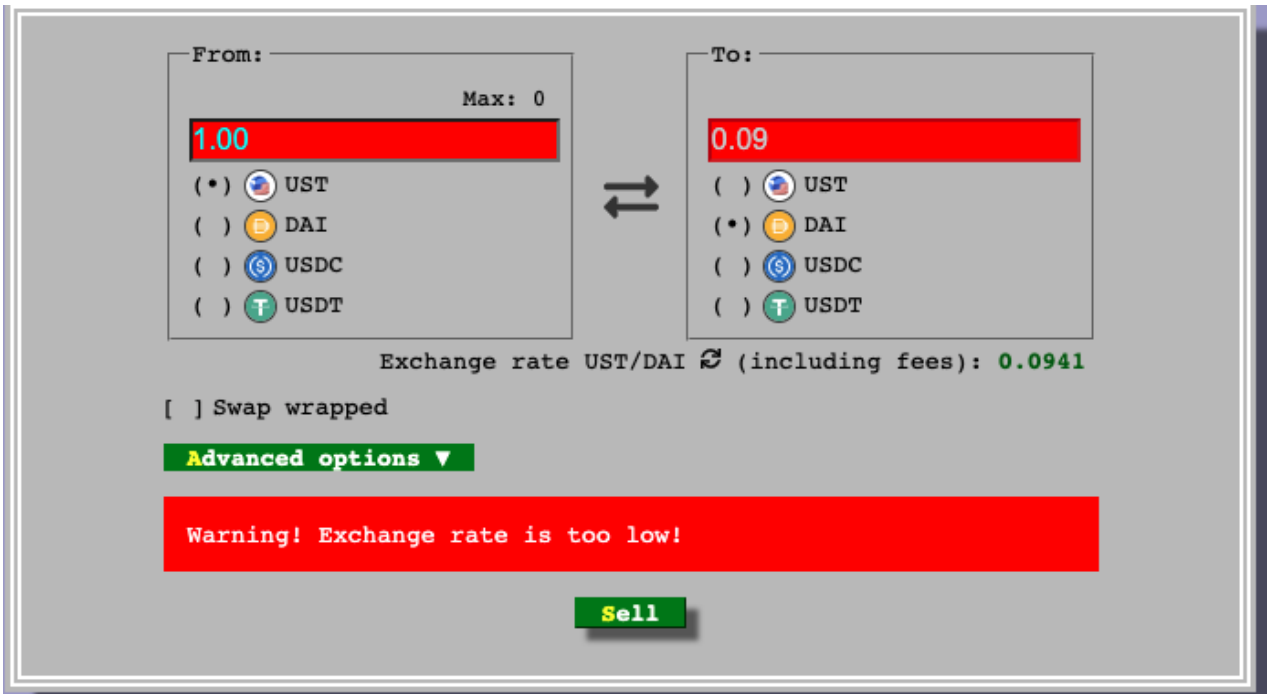
Burn and Mint

Swapping and burning UST for LUNA means minting more LUNA, diluting the supply and dropping the price of this token. Additionally, as the price of LUNA drops, whenever you swap 1 UST for $1 worth of LUNA, you steadily need more and more LUNA to hit that $1 mark (which means minting even more LUNA).

At a certain point, the price of LUNA could drop so low that there simply isn't enough liquidity to provide an escape hatch for all that UST coming in.

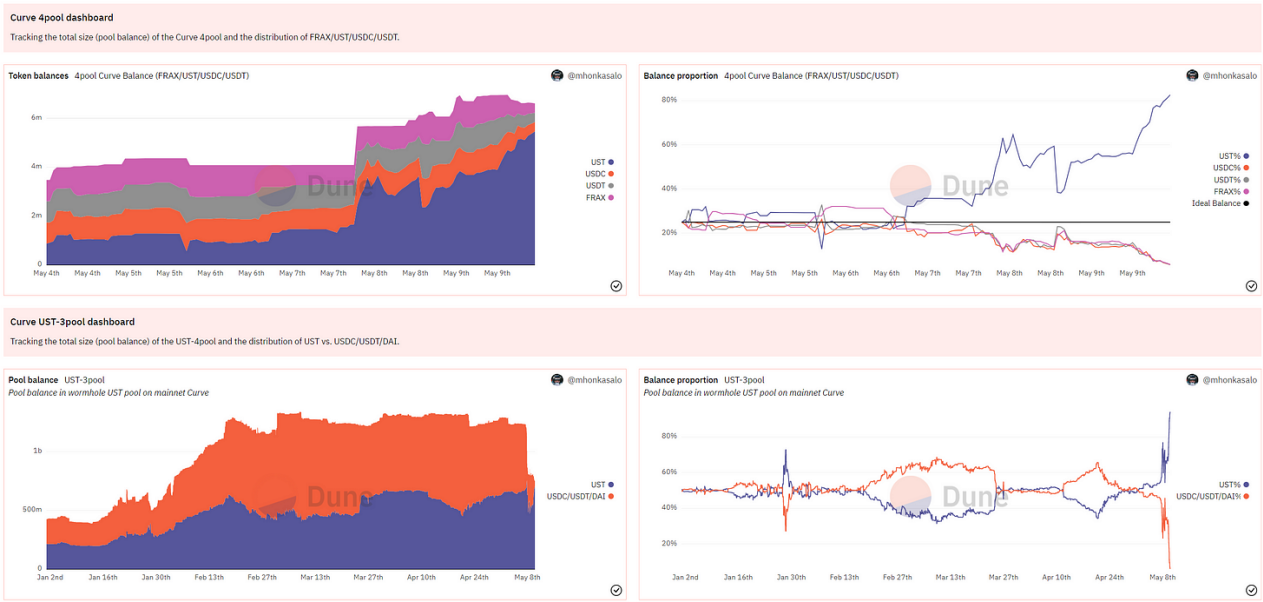
Option 2 using Curve

UST was in a Curve pool with DAI, USDT and USDC.



UST depegged by a little less than $0.02 over the weekend as Anchor departees swooped in and began flipping UST for any other stablecoin, be it Tether's USDT or Circle's USDC.

Eventually, the specific pool that allowed for these trades (called the "UST + 3Crv" pool, which also pools all the major stablecoins) became unbalanced, meaning there was far more UST than the other stablecoins in the pool.



If you sell UST for USDC on Curve, you will add more UST to this pool and remove USDC. Eventually, the pool will have more UST than USDC. In order to correct course, the pool then begins to offer that UST for a discount in hopes of getting arbitragers to make the opposite trade (and rebalance the pool).

This is in part why we began to see a slight depegging beginning on the weekend—Curve was doing what Curve is meant to do.

The problem in this specific case was that the opposite trade, the one that would rebalance the pool, wasn't happening. It appeared that, despite the relatively lucrative arbitrage trade, no one wanted to be holding UST, due on part to the waning interest in Anchor.

And at this time, at least one investor dumped more than 85 million UST tokens in exchange for 84.5 million USDC tokens in this pool. This, of course, put even more pressure on UST's dollar peg as Curve continued to create the discount hoping to incentivize arbitrage traders to rebalance the pool.

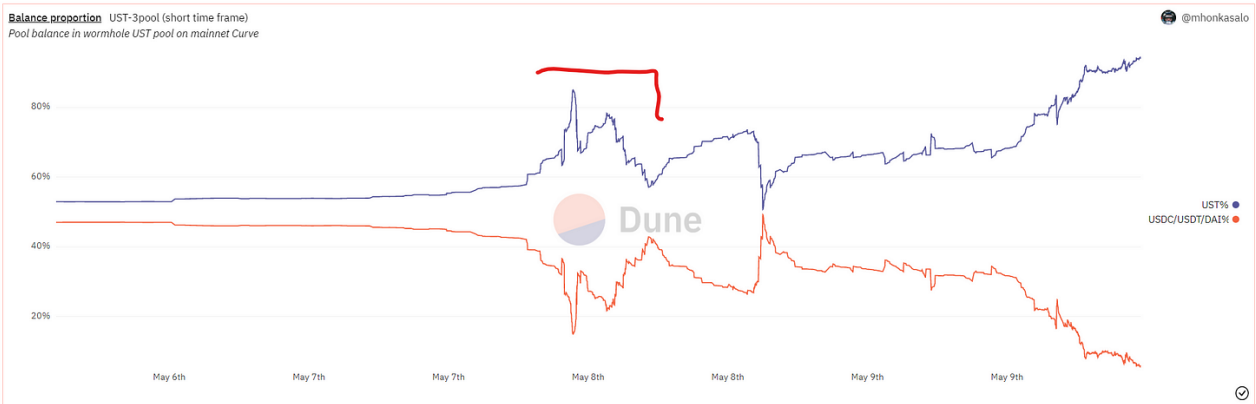
> 🚀Swap 85,001,010 $UST to 84,509,387 $USDC ($84,969,985)
> LP & veCRV Holder Fee: $33,988
> 💰Tx Hash: https://t.co/rbLEl4IndA 🦙🦙🦙
>
> — Curve Whale Watching (@CurveSwaps) May 7, 2022

What was just a $0.02 depeg on Sunday became a whopping $0.32 by Tuesday. At the same time, the $64 LUNA token fell below $30.

It was also around this point that the market capitalization of UST edged toward overtaking LUNA's, which would mean the latter would no longer be able to absorb the former, creating a death spiral.

The Luna Foundation Guard (LFG) stepped in.

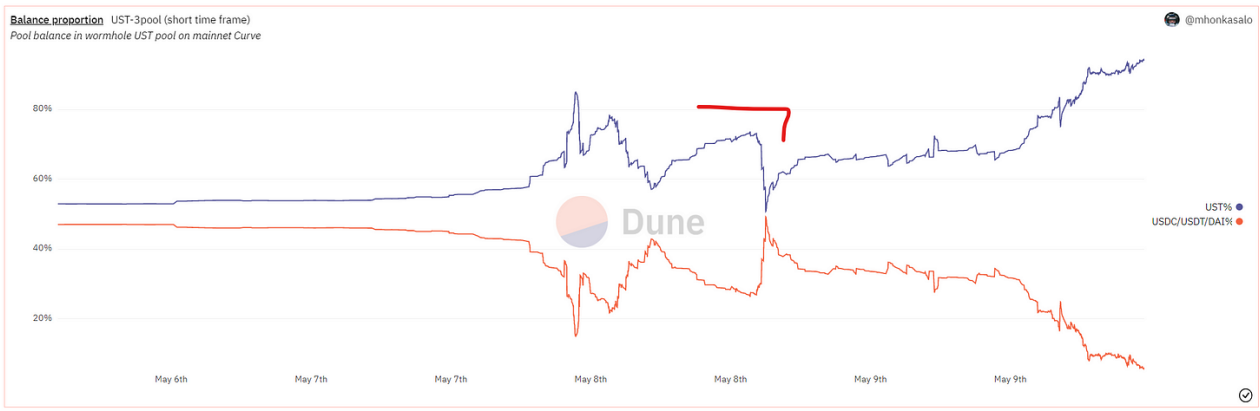
It dumped a ton of non-UST stablecoins (approximately $216 million came from Jump Crypto, whose president is also on the LFG Council) into the Curve pool to help the stablecoin find its peg. It then reportedly began deploying the Bitcoin holdings that it's been stockpiling to a "professional market maker" who was essentially told to spend BTC when UST is below the peg (and vice versa if it ever trades above the peg).



And UST jumped from $0.64 back to $0.93.

Unfortunately, it was a brief reprieve. Exits through Curve ate through the bailout liquidity. It's also unclear whether that BTC was ever actually used to defend the peg.

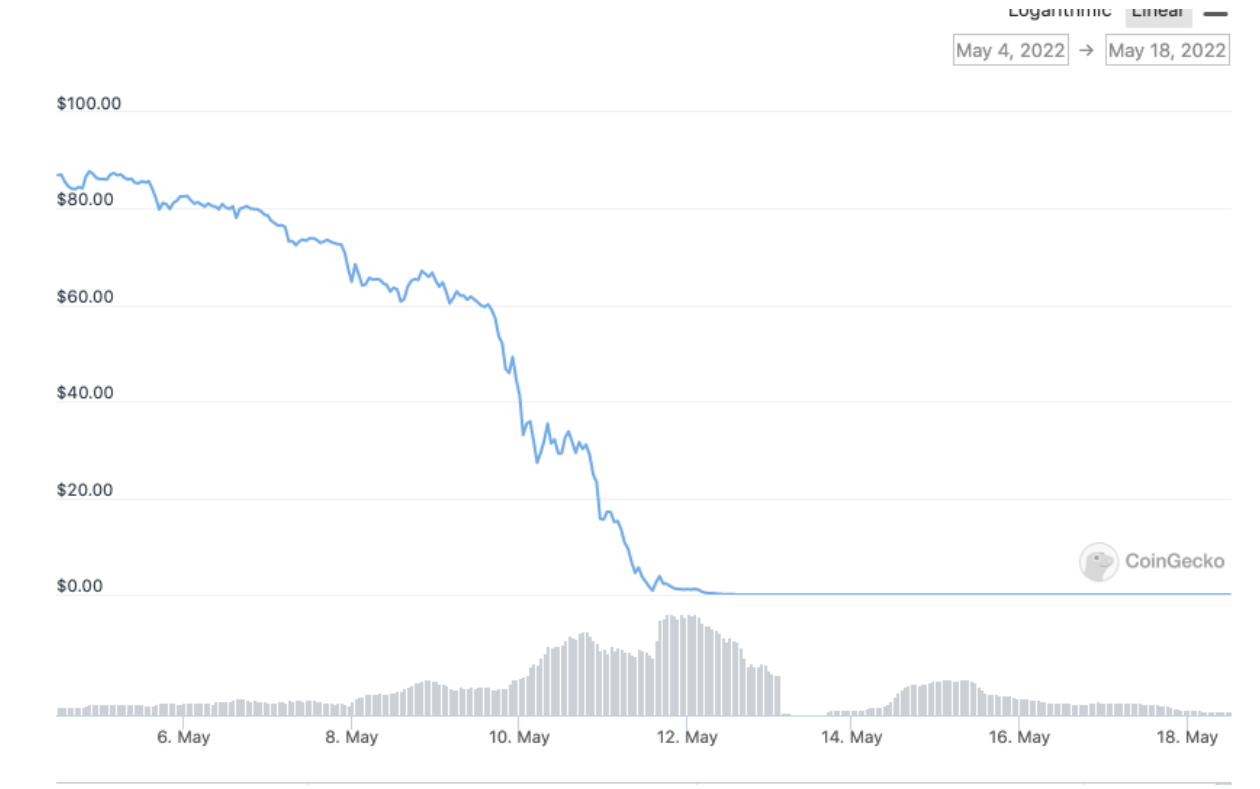
The second stage was similar to the first. Constant swaps on Curve — mostly $300k at a time and often from addresses that didn't look "malicious" but some random NFT trader.



Ultimately, LUNA's price continued to plummet as folks ditched UST and then sold their LUNA until, eventually, LUNA's price was so low that there wasn't enough runway for UST, creating an enormous amount of bad debt.

Do Kwon and the Terra community doubled down, and opened up how much LUNA can be minted at a time. But all this did was accelerate the spiral.
On May 8, LUNA had a 343 million circulating supply.
By May 12, that figure had ballooned to 32.3 billion (and counting).

Despite the negative feedback loop, the Terra community proposed three more emergency actions, which boil down to simply lighting as much UST on fire as possible (without having to mint LUNA on the other end).



Eventually the Terra Blockchain halted to prevent governance attacks given the low costs of LUNA

Knock on effects

$200 billion of crypto value vaporized in 24 hours - more than the market cap of BTC in 2020

Stablecoin Theory

Paper

There are two classes of stablecoin: custodial, which require trust in a third party, and non-custodial, which replace this trust with economic mechanisms.
Major custodial examples such as Tether, Binance USD, USDC, and TrueUSD have a combined market capitalization of over USD 10bn.
On the non-custodial side, of the USD 1bn of value locked in so-called Decentralized Finance (DeFi) protocols, more than 50% are allocated to Maker's Dai stablecoin

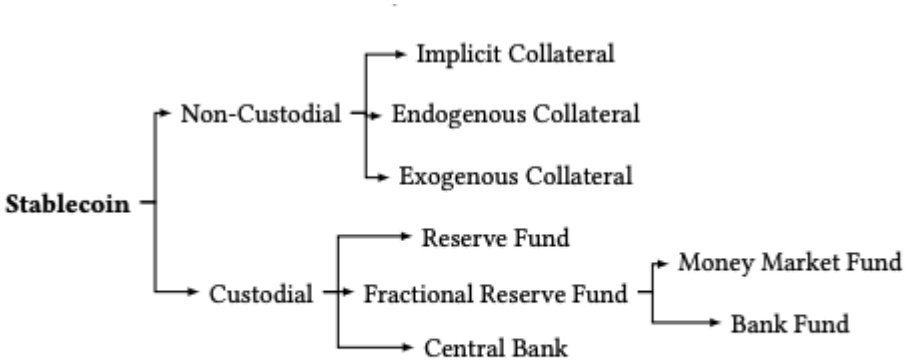


Figure 1: Risk-based overview of stablecoin design space.

Custodial

- Reserve Fund - TUSD, USDC, Libra v2
  In Reserve Fund stablecoins, the stablecoin maintains a 100% reserve ratio–i.e., each stablecoin is backed by a unit of the reserve asset (e.g., 1 USD) held by the custodian
- Fractional Reserve Fund
  A Fractional Reserve Fund stablecoin is backed by a mixture of reserve assets and other capital assets, and has a target price.
  - Bank Fund - Tether
  - Money Market Fund - Libra v1
- CDBC - Digital Yuan

Non Custodial

The collateral for these is of the following types

- Exogenous when the collateral has primary outside use cases
  Stablecoins are issued against this collateral subject to a collateral factor that dictates the minimum overcollateralization allowed in the system.
- Endogenous when the collateral is created for the purpose of being collateral This means that it has few, if any, competing uses outside of the stablecoin system
- Implicit when the design lacks explicit collateralization.

Examples

- Exogenous - DAI
- Endogenous - Synthetix, Terra
- Implicit - Nu-Bits

There is something of a spectrum of exogenous and endogenous collateral, e.g. Steemit and Celo

A further dimension is the type of collateral used, this may be

- Fiat currency, such as USD
- Cryptocurrency
- Commodity - such as gold

2 Fundamental questions

1. (Incentive Security). Is there mutually profitable continued participation across all required parties?
   If not, then the mechanism cannot work as no one will participate. This question also includes incentives around attacks; in particular, if incentives lead to profitable attacks, then rational agents will be less inclined to participate.

2. (Economic Stability). Do the incentives actually lead to stable outcomes?

Composite Stablecoins

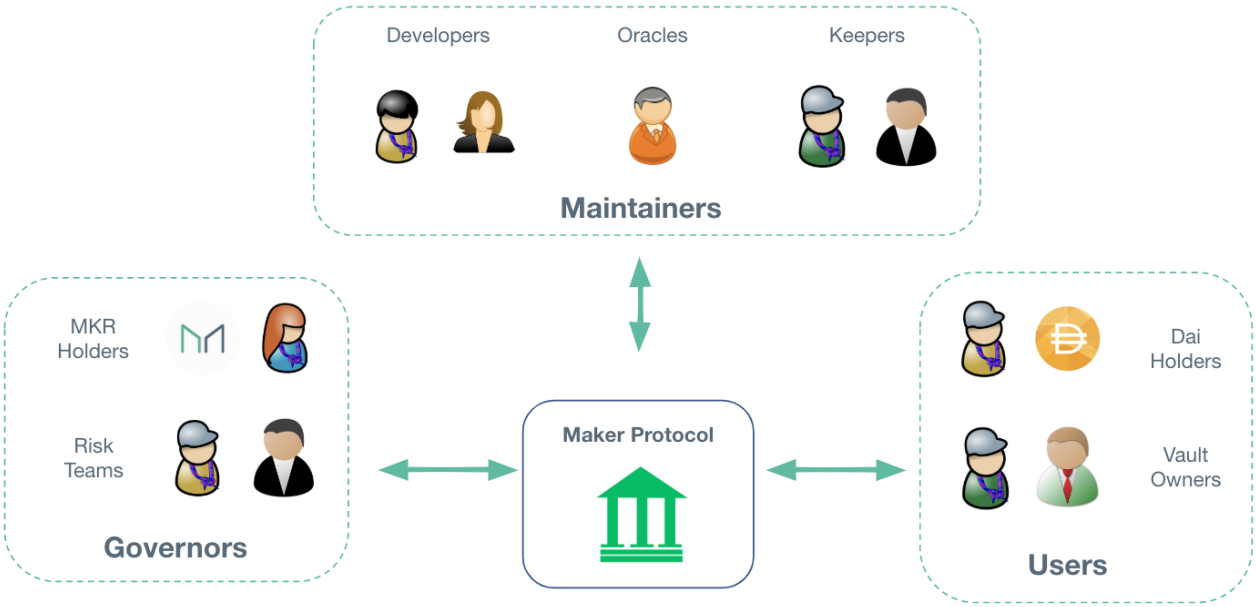
1. ETF stablecoins
   An ETF type arbitrage mechanism is used to redeem the stablecoin against a basket of assets
2. DEX stablecoin
   This aims to spread risk over the basket while providing an exchange service between the constituents, and so the basket weights change with exchange demand.
3. CDO stablecoin
   This segregates stablecoin risk into tranches
4. A rainy day fund (RDF) stablecoin
   This holds a basket of assets that accrues value to a safety buffer over time through arbitrage, fees, and other collateral uses.

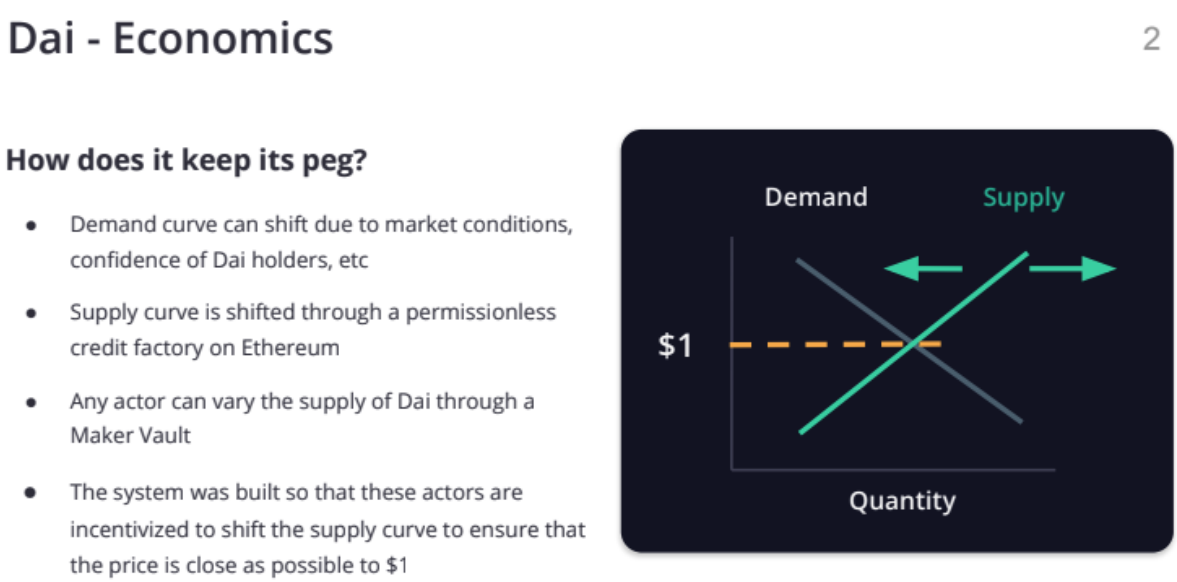
Other stablecoins

DAI

Dai is created from an overcollateralized loan and repayment process facilitated by MakerDAO's smart contracts.
Users who deposit Ether (or other cryptocurrencies accepted as collateral are able to borrow against the value of their deposits and receive newly generated Dai.
The minimum collateralization ratio for Ether is currently set at 150%, or in other words, depositing $150 worth of Ether allows one to borrow up to 100 Dai

Maker Protocol in detail





Dai - Vault

Maker Vault

- Borrow Dai through locking up crypto assets as collateral
- Repay Dai + fee to retrieve collateral
- Safe, over-collateralized Vault >

Dai - Vault

Liquidations

1. A Vault is automatically liquidated if the collateral value (in USD) falls too low
2. Part of the collateral is auctioned off by the Protocol to cover the outstanding debt + penalty fee
3. Dai is then burned by the Protocol to decrease the supply

   Vault owner receives the leftover collateral

System Lines of Defense

At any point, the system must have more value in collateral than value of Dai supply.
The following mechanisms help maintain system solvency:

1. Supply and Demand Supply and demand of Vaults (and thus Dai) is influenced by the Stability Fees, Dai Savings Rate, and Debt Ceiling adjustments.

2. Liquidation Any time the collateral value of a Vault gets closer to its debt, it becomes “risky-er”. The system liquidates Vaults that get too risky.

3. MKR Minting/Burning If MKR holders govern the Maker Protocol successfully, the Protocolwill accrue Surplus Dai as Dai holders pay Stability Fees. On the other hand, if liquidations are inadequate, then the Protocol will accrue Bad Debt. Once this Surplus Dai / Bad Debt amount hits a threshold, asvoted by MKR holders, then the Protocol will discharge Surplus Dai /Bad Debt through the Flapper / Flopper smart contract by buying and burning / minting and selling MKR, respectively.

4. Emergency Shutdown This is a process that is used as a last resort in cases of extreme market irrationality, attacks, or coordinated upgrades. Emergency Shutdown gracefully settles the Maker Platform while ensuring that all users, both Dai holders and Vault users, receive the net value of assets they are entitled to.

In March 2020, as a result of extraordinary market volatility, Dai experienced a deflationary deleveraging spiral that, at its peak, caused it to trade for up to $1.11 before returning to its intended $1.00 valuation
Article - $8.32 million was liquidated for 0 DAI

Frax

From Frax documentation
Frax is the first fractional-algorithmic stablecoin protocol. Frax is open-source, permissionless, and entirely on-chain – currently implemented on Ethereum and 12 other chains. The end goal of the Frax protocol is to provide a highly scalable, decentralized, algorithmic money in place of fixed-supply digital assets like BTC. The Frax ecosystem has 2 stablecoins: FRAX (pegged to the US dollar) & FPI (pegged to the Consumer Price Index).

Frax is a new paradigm in stablecoin design. It brings together familiar concepts into a never before seen protocol:

- **Fractional-Algorithmic** – Frax is the first and only stablecoin with parts of its supply backed by collateral and parts of the supply algorithmic. This means FRAX is the first stablecoin to have part of its supply floating/unbacked. The stablecoin (FRAX) is named after the "fractional-algorithmic" stability mechanism. The ratio of collateralized and algorithmic depends on the market's pricing of the FRAX stablecoin. If FRAX is trading at above $1, the protocol decreases the collateral ratio. If FRAX is trading at under $1, the protocol increases the collateral ratio.

- **Decentralized & Governance-minimized** – Community governed and emphasizing a highly autonomous, algorithmic approach with no active management.

- **Fully on-chain oracles** – Frax v1 uses Uniswap (ETH, USDT, USDC time-weighted average prices) and Chainlink (USD price) oracles.

- **Two Tokens** – FRAX is the stablecoin targeting a tight band around $1/coin. Frax Shares (FXS) is the governance token which accrues fees, seigniorage revenue, and excess collateral value.

- **Crypto Native CPI** – Frax's end vision is to build the first crypto native version of the CPI called the Frax Price Index (FPI) governed by FXS holders (and other protocol tokens). FRAX is currently pegged to USD but aspires to become the first decentralized, permissionless native unit of account which holds standard of living stable.

Price History from CoinGecko


May 4, 2022 → May 18, 2022
$1.03
$1.02
$1.02
$1.01
$1.01
$1.00
$0.995000
$0.990000
$0.985000
CoinGecko
6. May
8. May
10. May
12. May
14. May
16. May
18. May

Float

Unlike most stablecoins, whilst FLOAT is designed to be stable and have significantly lower volatility than most cryptocurrencies, **it is not designed to hold its price at $1.00**. Instead, **it is designed to float and change value over time.**

FLOAT is stabilised through frequent auctions to expand or contract the supply. It is also partly supported by a Basket of cryptocurrencies that are used as part of the auction process.

FLOAT is also supported by a second token, BANK .
This token serves three purposes:

- to take the profit created in times of excess demand for FLOAT,
- to support the price of FLOAT from time to time and
- to govern the Float protocol.

Iron

From Rekt :

On Binance Smart Chain, IRON uses BUSD and their native token STEEL as collateral to maintain a peg at $1.
On Polygon, IRON used USDC and their native token TITAN to maintain the peg.

The incident started when TITAN became overpriced, perhaps due to users purchasing the token in order to farm TITAN pairs at ~50,000% APY.

Some large TITAN sales were made and the price became volatile, making investors nervous, and leading them to also sell their tokens.

The IRON stablecoin then lost it's peg due to TITAN dropping so rapidly.
This created a situation in which users could now redeem a token worth 90 cents, for 75 cents of stablecoin and 25 cents of TITAN. An incredible arbitrage opportunity which required minting new TITAN tokens each time.

The market was flooded with freshly minted TITAN, and a panic sale began, pushing down the TITAN price and therefore making the IRON stablecoin lose its peg even further.

Gyroscope Protocol

Gyroscope's mission is to build robust public infrastructure for DeFi. The central piece is a fully-backed stablecoin with all-weather reserves and algorithmic price bounding:

- **A fully backed stablecoin**: the Gyroscope stablecoin aims at a long-term reserve ratio of 100%, where every unit of stablecoin is backed by 1 USD worth of collateral.
- **An all-weather reserve**: the reserve is a basket of protocol-controlled assets that jointly collateralize the issued stablecoin. Initially most assets will be other stablecoins. The reserve aims to diversify all risks in DeFi to the greatest extent possible. It considers more than just price risk, but also censorship, regulatory, counterparty, oracle and governance risks.
- **Algorithmic price bounding**: Prices for minting and redeeming stablecoins are set algorithmically to balance the goal of maintaining a tight peg with the goal of long-term viability of the project in the face of short-term crises.

Further Resources

Video from Ariah Klages-Mundt
Stablecoins 2.0: Economic Foundations and Risk-based Models from Cornell