

# Decentralised Finance

**Networks of automated market makers  
arbitrage, order routing ...**

Lecture 11

Vincent Danos 23-11-20, BDL class, Sol, Edinburgh

# Outline

1. what is DeFi = decentralised finance
2. what is an automated market maker (aMM) = state machine to exchange assets; Uniswap example
3. why aMMs are so successful
4. networks of aMMs -> global price consistency -> arbitrage and order routing

1.

what is DeFi = decentralised finance

# Decentralised Finance (aka open finance)

assets being exchanged = ERC20 tokens

- users (=accounts) stay in control of their assets
- ownership = secret keys = no custody, ie no delegation of access rights, no IOUs
- financial transactions are mediated by smart contracts ~ state machines with controlled access
- smart contracts run on a neutral computational platform; confidence in code replaces trust in intermediaries
- financial functionalities are code hence are composable (unclear how much this is used; akropolis' hack)
- not clear which other substrates would be fit for DeFi

# What is being traded?

assets = tokens

- A, B, C ... represent tokens (ERC20) which people exchange/swap
- ERC20 can be freely created
- ... and hooked to any contract
- some of them are utility tokens (BAT), some security tokens (wBTC) <- "fundamental value" problem ...

# ERC20 building block

## Interface

```
interface IERC20 {  
    function totalSupply() external view returns (uint256);  
  
    function balanceOf(address who) external view returns (uint256);
```

**getters**

```
    function allowance(address owner, address spender)  
        external view returns (uint256);  
  
    function transfer(address to, uint256 value) external returns (bool);  
  
    function approve(address spender, uint256 value)  
        external returns (bool);  
  
    function transferFrom(address from, address to, uint256 value)  
        external returns (bool);
```

**transitions**

```
    event Transfer(  
        address indexed from,  
        address indexed to,  
        uint256 value  
    );  
  
    event Approval(  
        address indexed owner,  
        address indexed spender,  
        uint256 value  
    );
```

**events**

```
}
```

2.

what is an automated market maker (aMM)

# What is a market maker

an **MM** is a special kind of actor in a market always willing to trade

- MMs are liquidity providers posting offers eg Bs-for-As
- traders take offers
- MMs provide liquidity, traders consume it
- w/o MMs there is **less liquidity**

-liquidity means: possibility to always trade, at low "price impact"

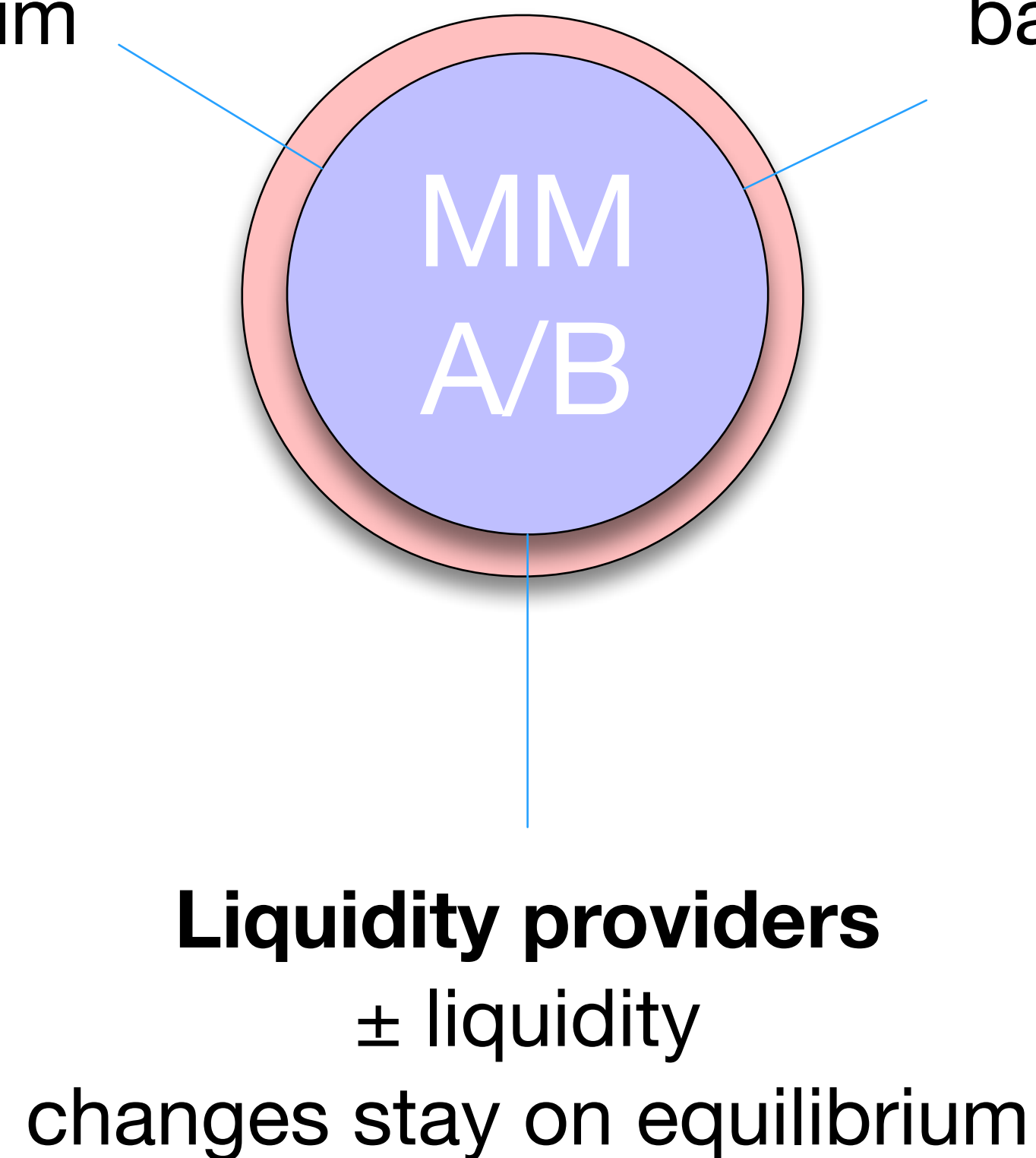


# Interacting with an MM

3 roles

**Traders**  
A's-for-B's  
move price  
off equilibrium

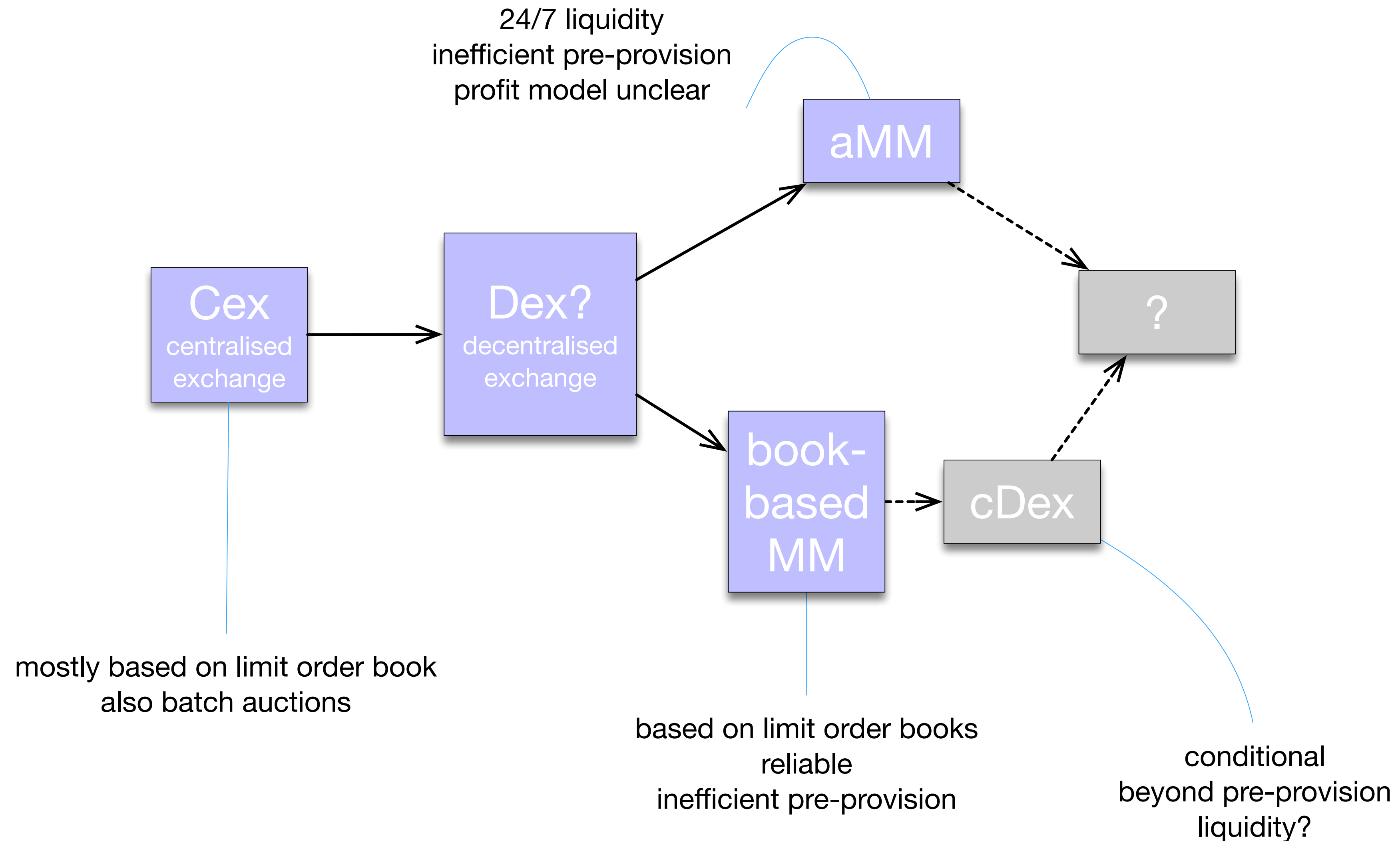
**Arbitrageurs**  
correct price  
back to equilibrium



equilibrium in this context means  
**no-arbitrage**  
(to be defined later)

# Types of markets

## evolution of market structures

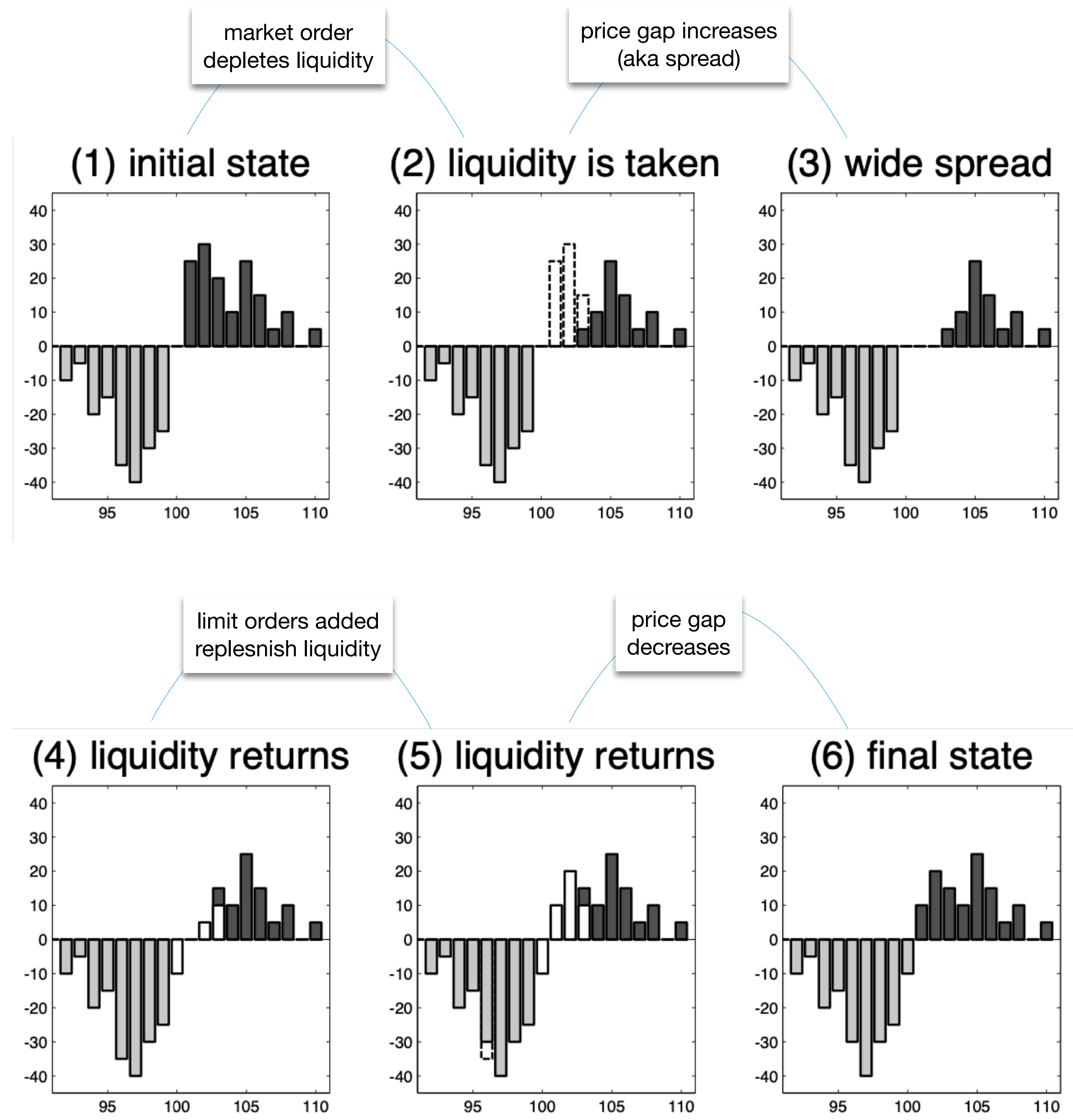


# A/B Order book

## market order vs limit order

can think of the OB as a continuous auction mechanism:

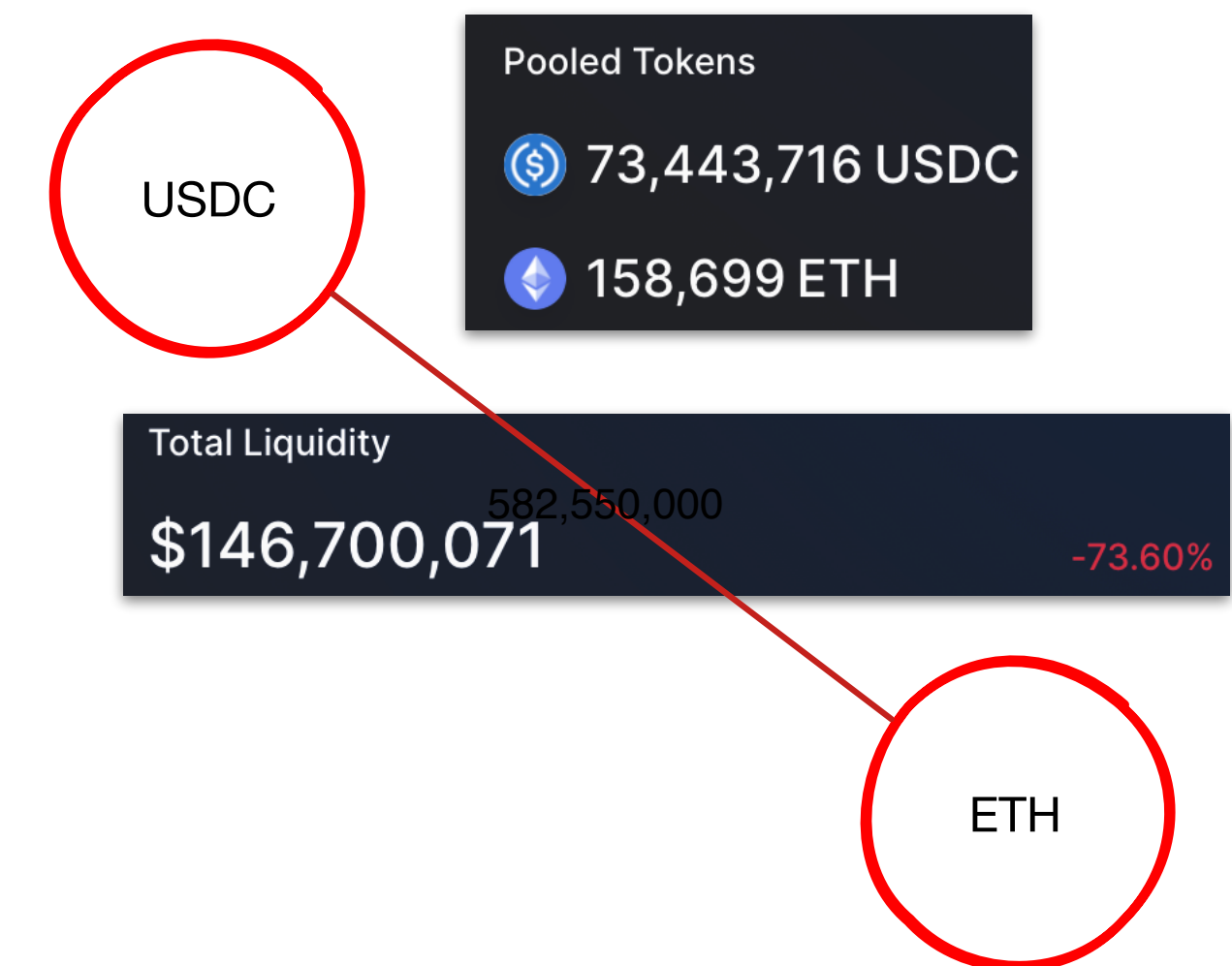
1. **Liquidity providers (LPs)**  
place and cancel offers (aka limit orders): price in **A** and amount of **B**; provide liquidity
2. **Traders** issue market orders:  
amount of **B**; consume liquidity
3. LPs are competing



# aMMs

## comparison with an order book

- no order book or other type of auction, no price oracle
- instead:
  - Liquidity providers are not competing = they are passively providing liquidity and share fees
  - liquidity provided by LPs feed reserves (aka "pools")
  - price is computed by a **price function** from current reserves = state machine (described in the next slides)
- in Cex'es books liquidity is "promised", here it is captive



2.1

The Uniswap example

# Uniswap state machine 1

## Trader action

**State of an edge**  $\theta = ([A], [B], \gamma)$  in  $\mathbb{R}_+^2 \times [0, 1]$  with  $[A]$  the amount of  $A$  in the pool,  $[B]$  the amount of  $B$ , and  $0 \leq 1 - \gamma \ll 1$  the fee.

**Trader action 1** The price function gives the amount  $y$  of  $B$  paid-out for an amount  $x$  of  $A$  paid-in:

$$f_\theta(x) = \gamma[B] \frac{x}{\gamma x + [A]} = ([B]/[A]) \frac{\gamma(x/[A])}{\gamma(x/[A]) + 1}$$

**NB1** -  $f_\theta$  factorises through  $\phi(x) = x/(1+x)$ , meaning it works in relative sizes of trades.

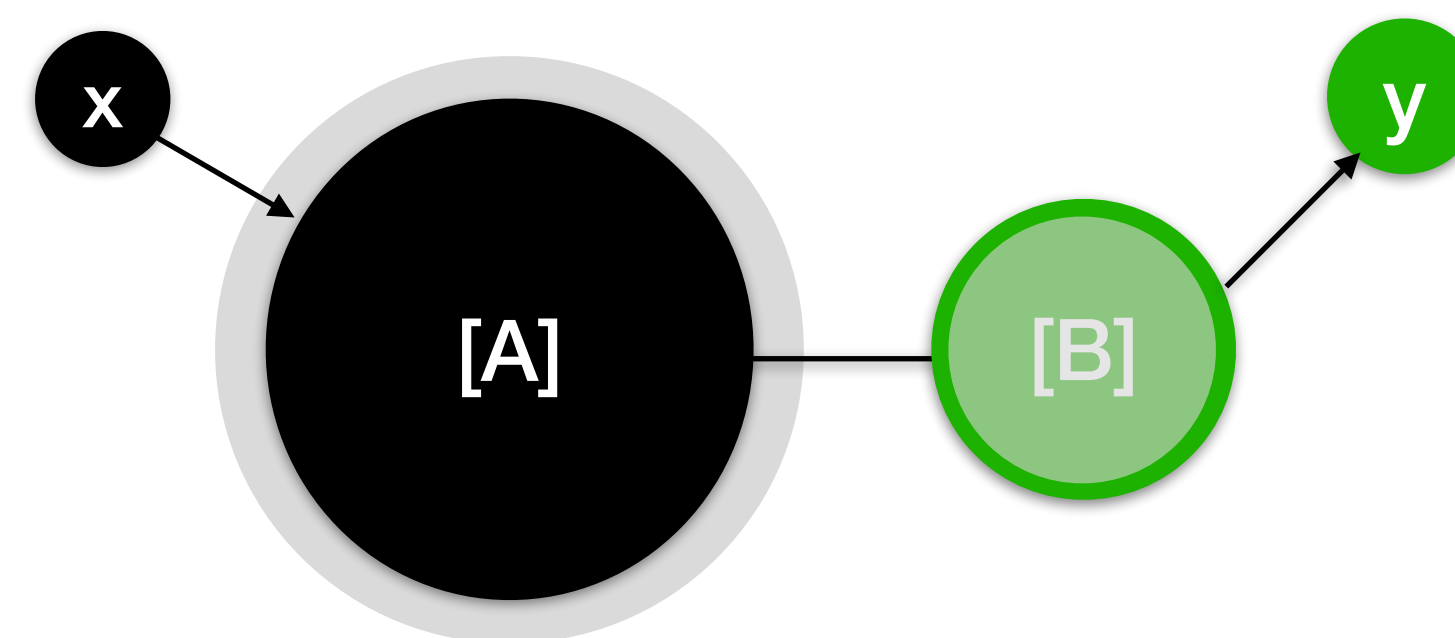
**NB2** - We see  $f_\theta(0) = 0$ ,  $f_\theta$  is non-decreasing, and strictly concave:

$$f'_\theta(x) = \gamma[A][B] \frac{1}{(\gamma x + [A])^2} > 0$$

$$f''_\theta(x) = -2\gamma^2[A][B] \frac{1}{(\gamma x + [A])^3} < 0$$

**NB3** - Marginal (or linear) price  $f'_\theta(0) = \gamma([B]/[A])$ :

$$f_\theta(x) = f'_\theta(0)x + o(x) = \gamma([B]/[A])x$$



# Abstract description 1

qualitative properties of the price function

Reasonably:

$$f_{\theta}(0) = 0$$

(no money for nothing)

$$f_{\theta} < [B]$$

(never dry)

$f_{\theta}$  non-decreasing

$f_{\theta}$  concave



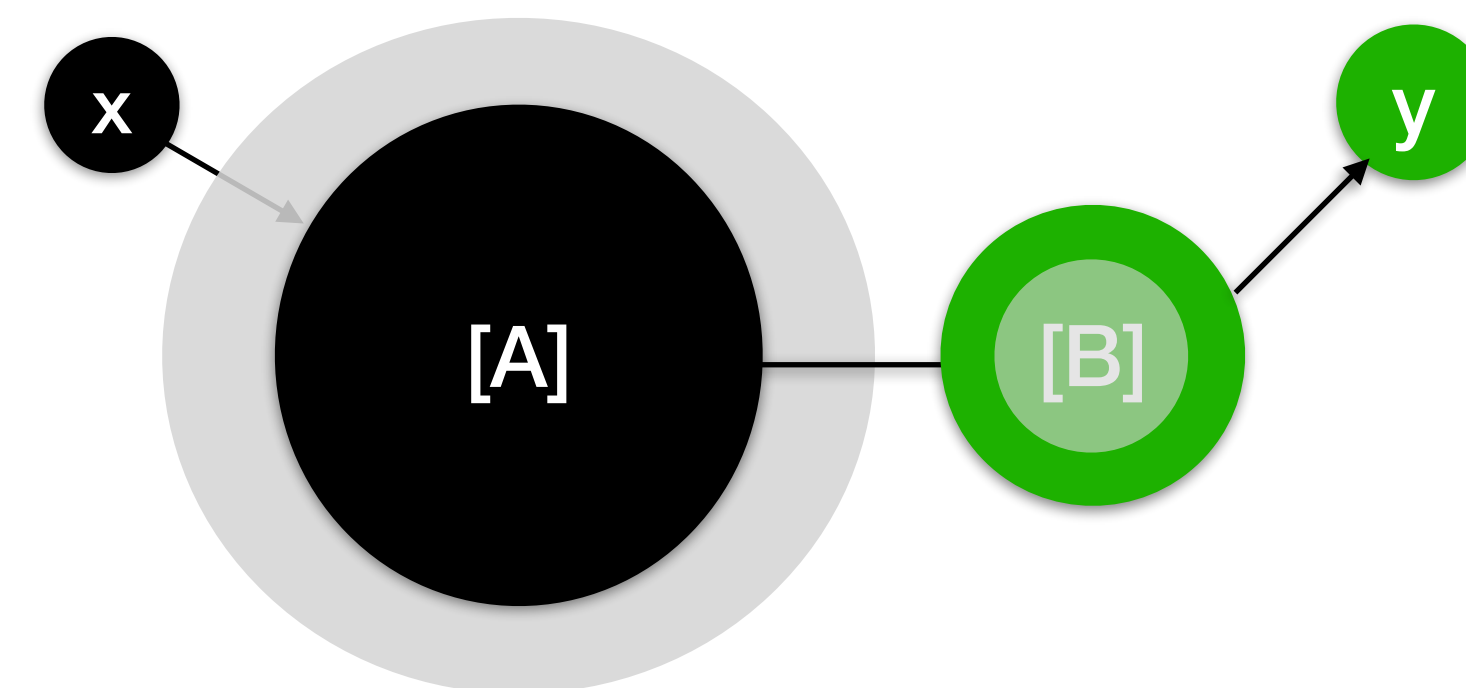
# Uniswap state machine 2

## Trader action 2

**Trader action 2** Trader action updates state as follows:

$$x \cdot ([A], [B], \gamma) = [A] + x, [B] \frac{[A]}{[A] + \gamma x}, \gamma$$

Has impact on price (scarcer token appreciates:  $f_{x,\theta} \leq f_\theta$ ).



**Example** Suppose the pool is  $100A + 100B$ , hence  $\rho = [B]/[A] = 1$ ;

[linear regime] suppose  $a = 1$ , then  $\alpha = a/[A] = \frac{1}{100}$ ,  $\beta = b/[B] = \frac{1}{101}$ , so  $b = \frac{100}{101}$ , ie very nearly a mean price of 1;

[mid regime] suppose  $a = 100$ , then  $\alpha = 1$ ,  $\beta = \frac{1}{2}$ , hence the actual mean price is  $p(B|A) = 1/2$ , to compare with the linear regime of 1/1

[saturated regime] suppose  $a = 10000$ , then  $\alpha = 100$ ,  $\beta = \frac{1}{1.01}$ ,  $b = \frac{100}{1.01}$ , with terrible mean price is  $p(B|A) = \frac{1}{101}$ .



# Uniswap state machine 3

## LP action

**LP action** The LP action is defined for  $x \in [\max(-[A], -[B]), \infty)$  and updates as follows:

$$x : ([A], [B], \gamma) = [A] + x, [B] + x([B]/[A]), \gamma$$

No impact on marginal price:

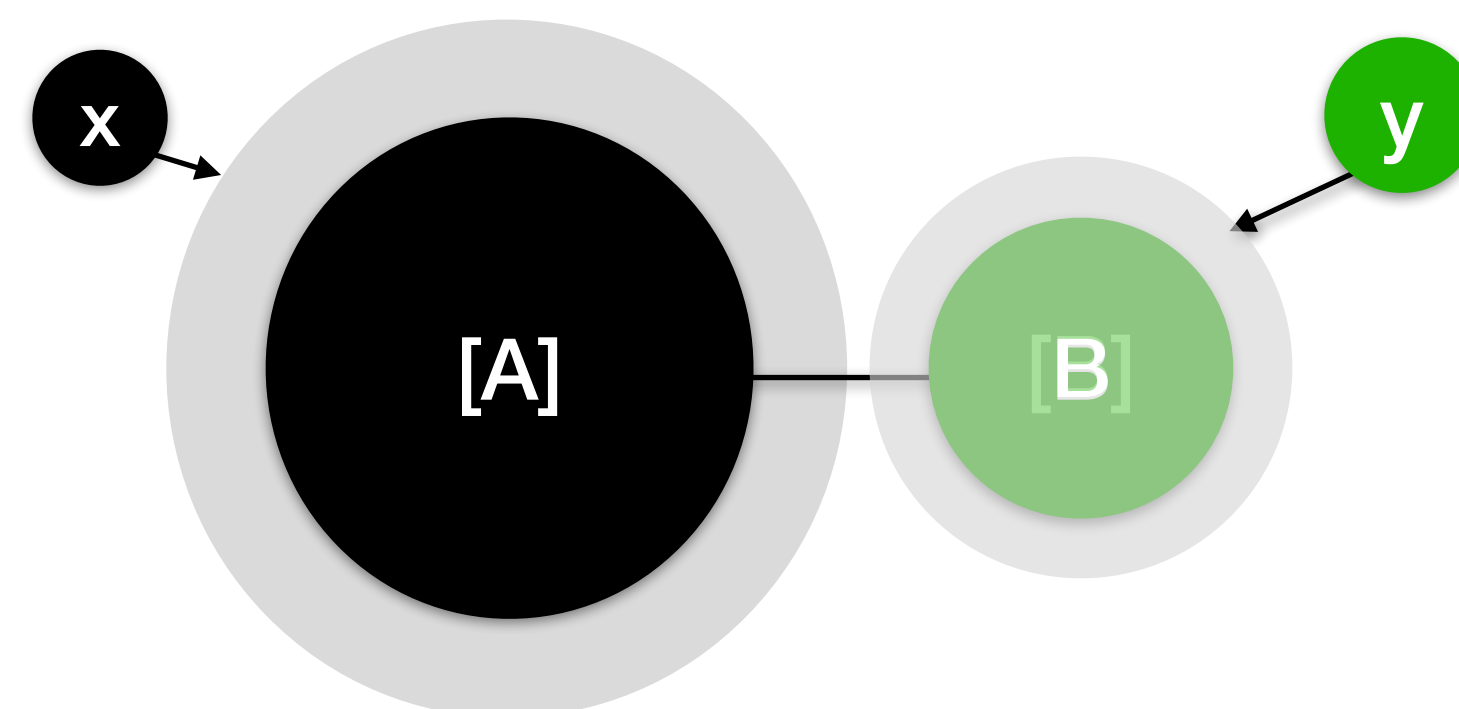
$$f'_{x:\theta}(0) = \gamma([B] + x[B]/[A])/([A] + x) = \gamma([B]/[A]) = f'_\theta(0)$$

One can show *no slicing*

$$f_\theta(x) + f_{x:\theta}(y) \leq f_\theta(x + y) \leq f_\theta(x) + f_\theta(y)$$

The second inequality is by sub-additivity, which follows from concavity and  $f(0) = 0$ . So we also get for free the weaker:

$$f_{x:\theta} \leq f_\theta$$



# Peek at the code

code is open, transitions are logged as events

```
contract UniswapExchange {
    using SafeMath for uint256;

    /// EVENTS
    // transitions
    event EthToTokenPurchase(address indexed buyer, uint256 indexed ethIn, uint256 indexed tokensOut);
    event TokenToEthPurchase(address indexed buyer, uint256 indexed tokensIn, uint256 indexed ethOut);
    event Investment(address indexed liquidityProvider, uint256 indexed sharesPurchased);
    event Divestment(address indexed liquidityProvider, uint256 indexed sharesBurned);
```

```
contract UniswapFactory is FactoryInterface {
    event ExchangeLaunch(address indexed exchange, address indexed token);
```

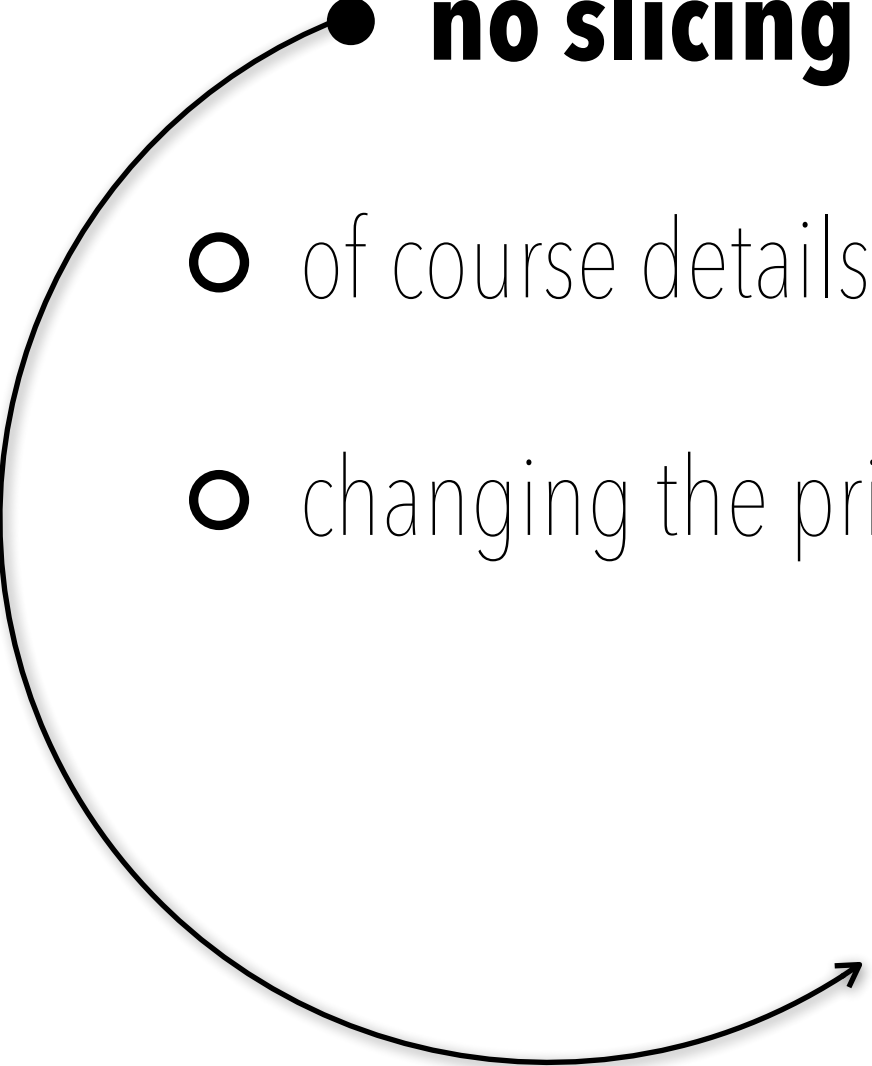
2.2

Uniswap variants

# Abstract description 2

details of the price function do not matter qualitatively

- unclear what exact class is the right one but:
  - any map which is (strictly) **concave**, **0@0**, and **non-decreasing** will “work”
  - **no slicing** seems natural to ask and entails “price sensing”
- of course details matter for finding closed forms, Uniswap is algebraically simple
- changing the price function is the source of many variants (next slide)


$$\begin{aligned}f_{\theta}(x) + f_{x \cdot \theta}(y) &\leq f_{\theta}(x + y) \leq f_{\theta}(x) + f_{\theta}(y) \\ f_{x \cdot \theta} &\leq f_{\theta}\end{aligned}$$

# A convenient representation

implicit description of the price function via invariant

Supposing  $\gamma = 1$  (no fee):  $y$  the amount of  $B$  paid out is related to  $x$  the amount of  $A$  received by the *constant product rule*. That is to say  $x$  and  $y$  have to be such that:

$$([A] + x)([B] - y) = [A][B]$$

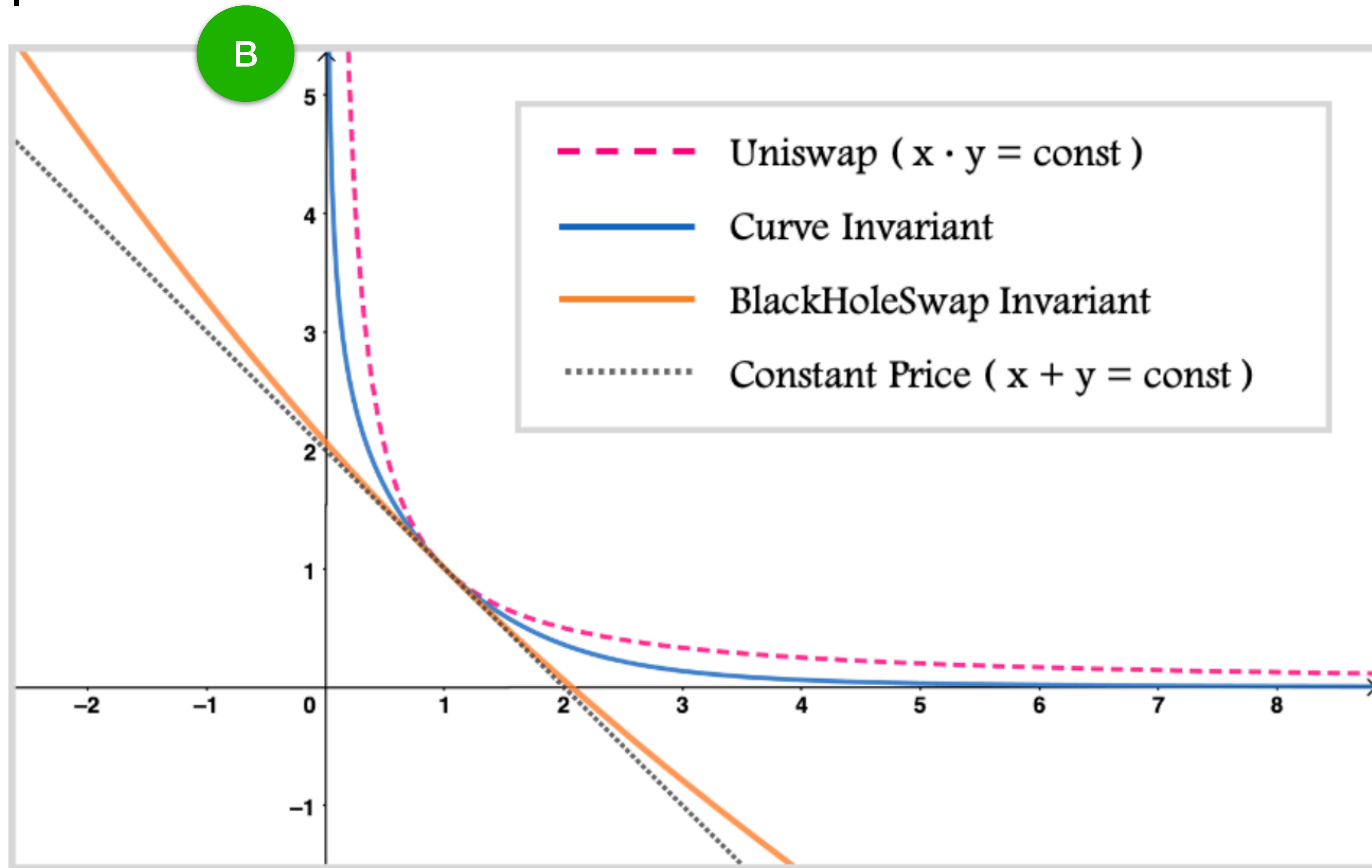
If we introduce relative changes  $\alpha = x/[A]$ ,  $\beta = y/[B]$  we get the intensive form of the invariant:

$$(1 + \alpha)(1 - \beta) = 1$$



# Make your own curve!

implicit description

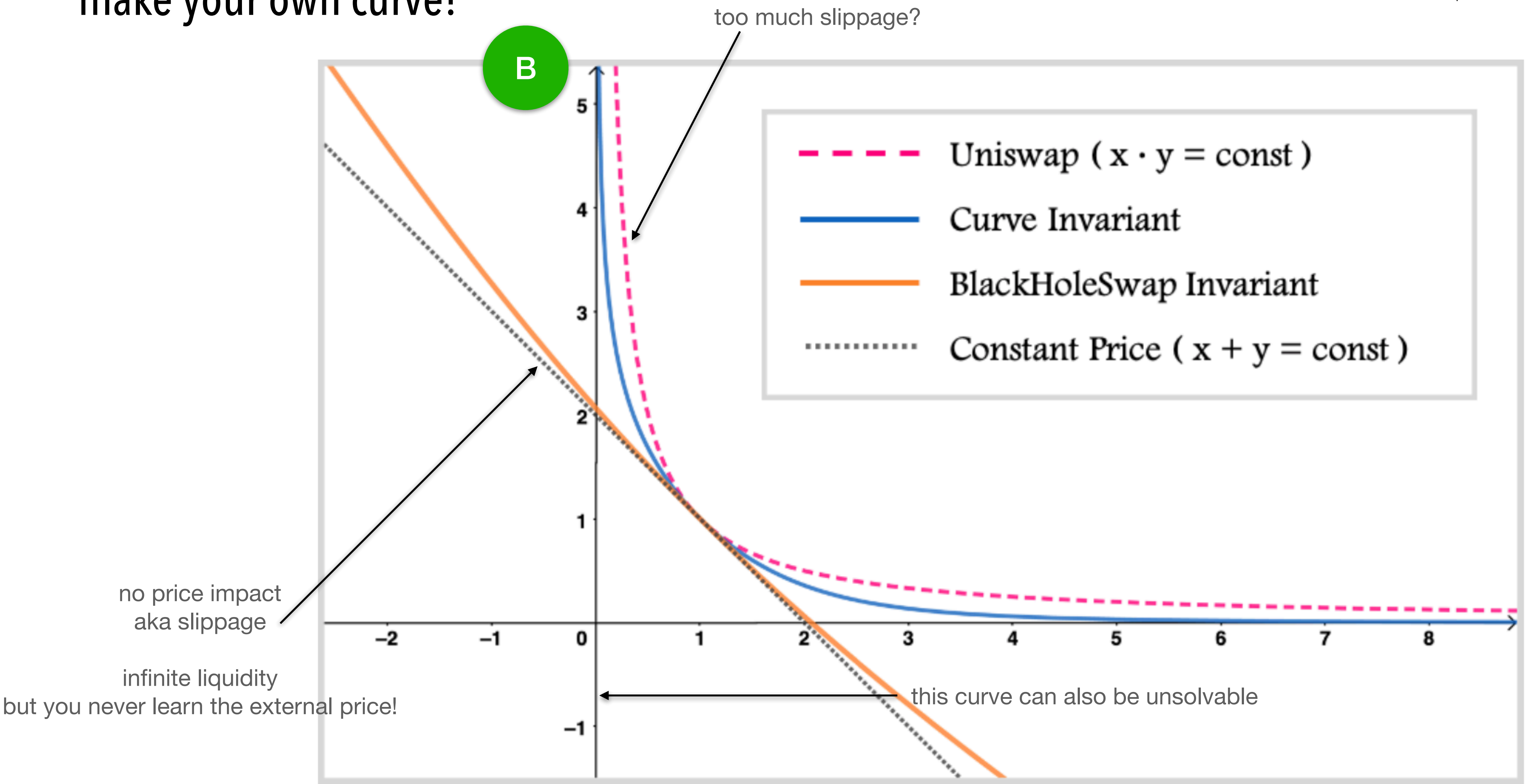


A

# The slippage vs arb trade-off

make your own curve!

**caveat:** implicit description



**A**

3.

why aMM are so successful and whether it will last ...



# Reasons for aMM dominance

may not last ...

- low resources <- good for gas price
- simple code <- good for confidence
- programmable lego brick - can create a token and hook it up somewhere to form a new pair
- TKRs are happy because it is 24/7 liquid and the price is arb'ed -so good
- LPs are happy because they get fees on each swap
- ARBs are happy, who doesn't like a free lunch



# Confidence in the code v1 (300 lines) - eg **LP** action

```
function investLiquidity(uint256 _minShares) external payable exchangeInitialized
{
    require(msg.value > 0 && _minShares > 0);
    // user injects x = msg.value ETH to buy a bunch of shares Sp
    // why bother to check if x > 0 and min_share > 0?
    uint256 ethPerShare = ethPool.div(totalShares);
    // price of share in ETH: [ETH]/[S]
    // what do you do with the remainder -> it goes in the pocket of the pool!
    // division-by-zero! div must contain some test
    // should price of share not be in the permanent state of the contract
    // rather than be a temporary variable?
    // what if ethPool = 0? ethPerShare = 0 and then revert?
    require(msg.value >= ethPerShare);
    // this "require" is not needed nor desirable
    // it costs gas for every user not just idiots who do not provide enough money
    // besides if you have not enough money for 1 share you will get zero which is perfect
    uint256 sharesPurchased = msg.value.div(ethPerShare);
    // Sp = x / ([ETH]/[S]) = x [S]/[ETH]
    // solve Sp.{x = [ETH]/[S] * Sp}
    // division-by-zero again; remainder is pocketed (no change!)
    // would return zero for idiots in the absence of the "require" right before
    require(sharesPurchased >= _minShares);
    // this is a limit order, it is OK to put it
    // suppose require-2 is not there, and msg.value < ethPerShare
    // then sharesPurchased = 0 (.div means Euclidean quotient) as intended
    uint256 tokensPerShare = tokenPool.div(totalShares);
    // price of share in DAI: [DAI]/[S]
    // same remark - should be a state variable
    uint256 tokensRequired = sharesPurchased.mul(tokensPerShare); // Sp * [DAI]/[S]
    // series of updates
    shares[msg.sender] = shares[msg.sender].add(sharesPurchased); // add Sp to user account
    totalShares = totalShares.add(sharesPurchased); // update total Shares
    ethPool = ethPool.add(msg.value); // update ETH pool
    tokenPool = tokenPool.add(tokensRequired); // update DAI pool
    invariant = ethPool.mul(tokenPool); // update invariant
    Investment(msg.sender, sharesPurchased); // event
    // execute a request for tokens on the sender's behalf
    require(token.transferFrom(msg.sender, address(this), tokensRequired));
    // token.transferFrom returns false if order does not succeed and "require" fails then and everything reverts
    // we rely on contract to contract communication with no authentication necessary
    // should be earlier in the sequence, right after computing: "tokensRequired"
    // if idiot user has not provisioned, updates will be paid for and then undone
}
```

event





# Many clones

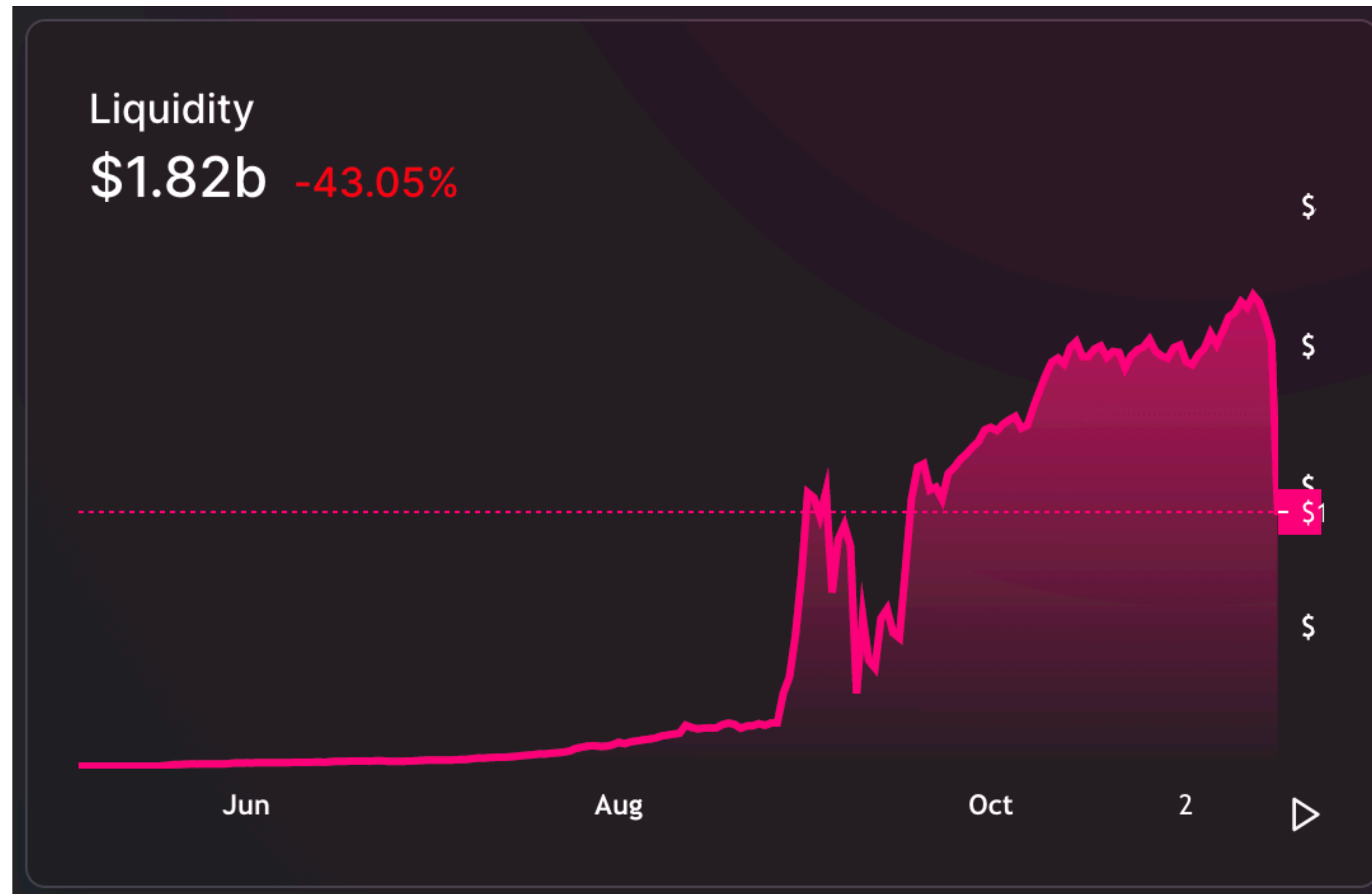
an example of price performance for a txn: ETH 100 -> x DAI



differ by price curve

# Many clones -> competition

vampire attacks - another macro-variable to monitor



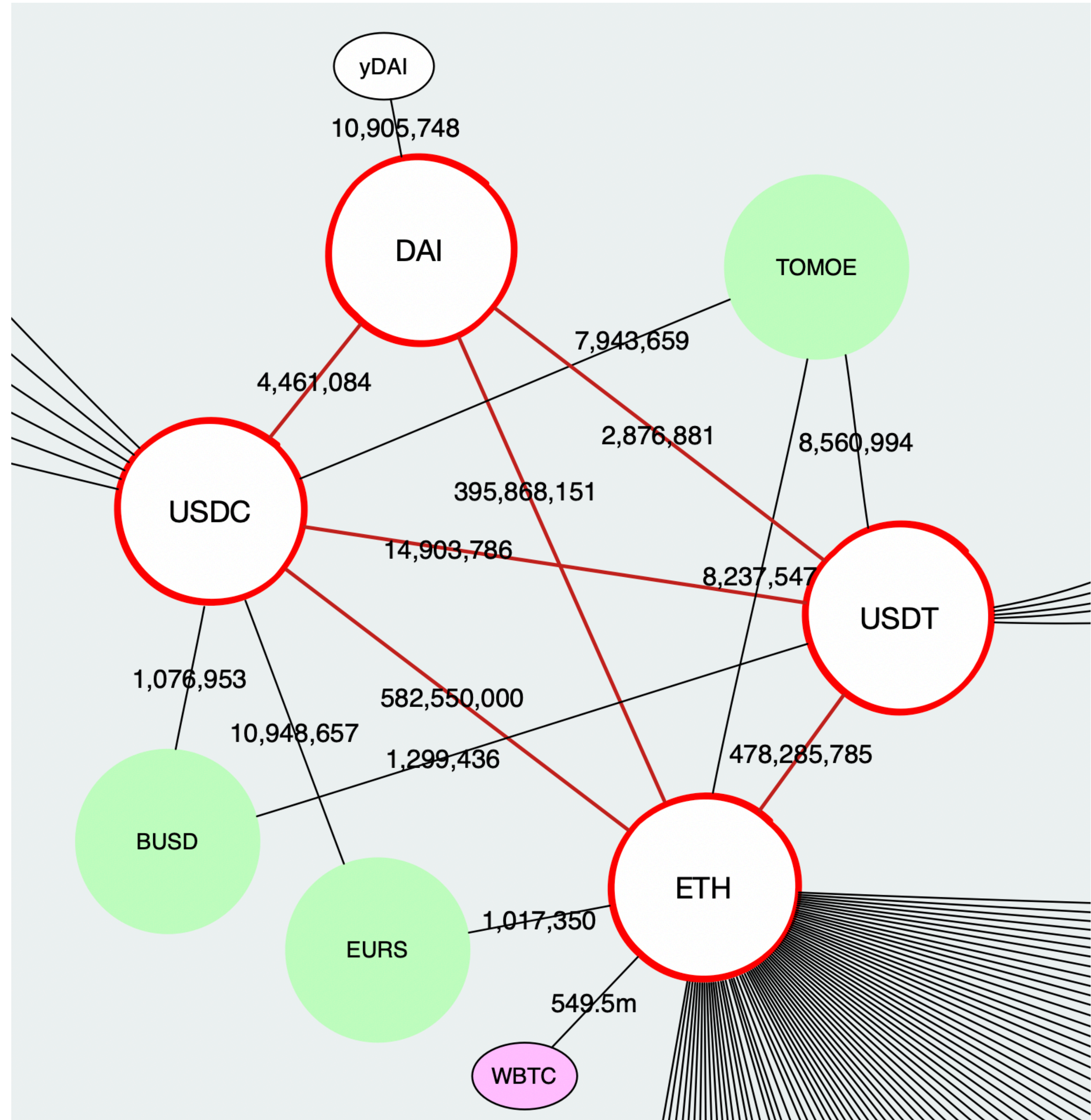
4.  
networks of aMMs



# The aMM network

superimpose all Uniswap edges

- real-time adjustment at freq = 1/15s
- notice the cycles
- should superimpose also: all the other swap machines (including Cexes)
- aggregators do that (1inch, paraswap, etc)





# Network specific questions

- **arbitrage**: looking for cyclic sequences of swaps which guarantee risk-less profit
- **order routing**: best combination of paths from **A** to **B** for a given amount **a**
- **liquidity migration**: LPs move from a pool to another

# Arbitrage

## abstract approach

**NB:** non-decreasing concave 0@0 functions compose; eg the **price function** of a **cycle**

cycle  $\rightarrow$  arbitrage condition  $f(x) \geq x$

include all swap contracts (not just aMMs) indeed OBs are also non-dec concave and 0@0!

There are closed formulas for 1) no-arb, and 2) max profit arb for uniswap

**Corollary 1** Arbitrage zones are downward closed, ie  $f(x_0) \geq x_0$  implies  $f(x) \geq x$  for all  $x \leq x_0$ .

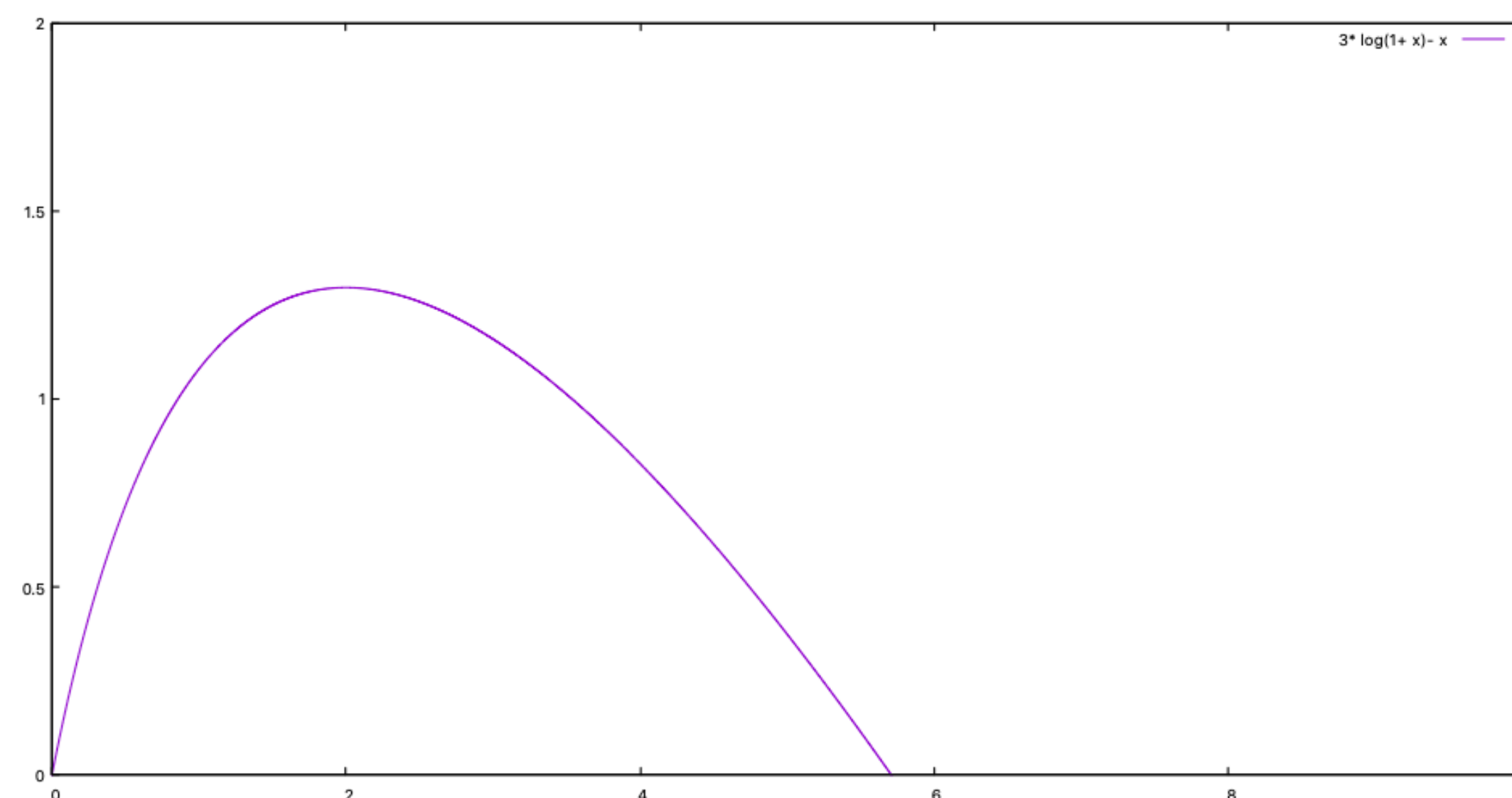


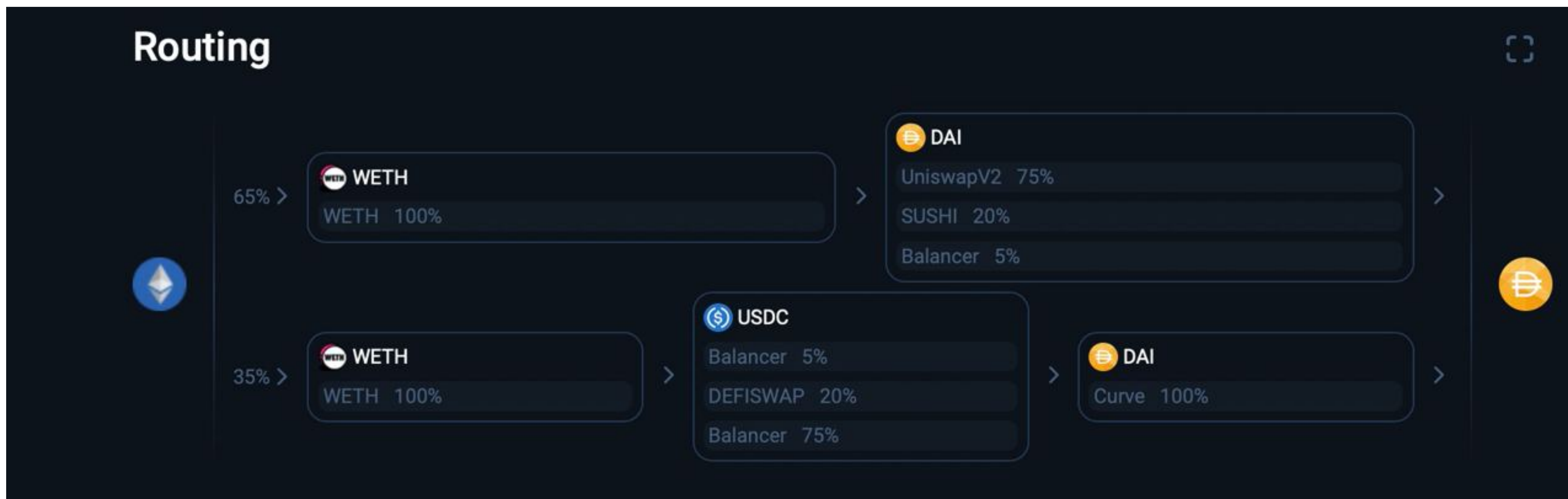
Figure 1: Plotting  $f(x) - x$  with  $f(x) = 3 \log(1 + x)$  a zz, ndec, concave function: profit is when  $f(x) - x \geq 0$ , the profit zone is a closed interval; max profit is obtained somewhere left of the middle of the profit interval.

**Corollary 2** If  $f \not\geq I$  (eg  $f$  is bounded), either  $K = (f - I)^{-1}[0, +\infty)$  is empty, or it is a compact interval and  $f - I$  has a unique maximum which belongs to  $K$ .



# Routing orders

convex combination of 6 paths: ETH -> DAI



**caveat:** noise = asynchrony + front-running

noise and malicious noise ...

### **reasons for hope:**

secure multiparty computation and cryptographic techniques such as zero-knowledge

(the topics of the previous two lectures)

can be used to resolve front-running in theory ...

but it is still an open question how