university of groningen / faculty of science and engineering

Inequality in Proof-of-Stake Schemes: A Simulation Study.

Bachelor's Project Thesis

Luca Bandelli, s3221253, l.bandelli@student.rug.nl,
Supervisors: Prof Dr Davide Grossi

Abstract: A fundamental concern of blockchain systems is dispensing reward to nodes in the network as incentive to align individual interests with the system's purpose. This study investigates the problem of wealth compounding in 'Proof of stake' schemes implementing a model described by previous research on the topic.

A computer simulation was designed to investigate system dynamics varying number of agents, duration and total dispensed reward. Scheme variations were considered along reward function, proposer selection mechanism, and initial state distribution. Two metrics assessing the equitability of the process as defined in the literature were collected for each simulation as response variables.

For initially uniform population, duration predicted an equitability increase, while the log ratio reward/duration ('load') predicted a decrease. Several interactions were also found significant, notably that of 'load' with duration, as well as general differences in mean performance across 'scheme variation'. Non-uniform initial distributions were also tested providing interesting insights.

Results aligned with previous research, and exposed critical relations between parameters of the simulation algorithm and equitability of the process.

1 Introduction

Since 2008 (Nakamoto, 2008) a new technology emerged known as 'Blockchain'. This technology allows the implementation of transactional software systems whose execution is fault tolerant and decentralized. The most common design for blockchains involves 'Proof of Work' (POW) as consensus protocol, which is by design vastly inefficient in terms of energy *consumption* and processing speed. The present study focuses on a promising alternative to POW, known as 'Proof of Stake' (POS), which solves the `energy` waste issue, but requires careful design to preserve the desirable properties of POW; in particular the fairness of the reward mechanism in POS and the phenomenon of wealth compounding is analyzed implementing the model described by Fanti, Kogan, Oh, Ruan, Viswanath, and Wang (2019) attempting to replicate their mathematical findings in a computer simulation.

Transactional systems revolve around the concept of 'a source of truth' about the state of some domain of discourse, in the form of a ledger describing all the steps (transactions) leading from the initial state to the current one (Xu et al., 2017; Nakamoto, 2008; Buterin, 2014). *Fault tolerance* on the other hand denotes systems designed without potential single points of failure, providing provably safe mechanisms for counteracting anticipated failure scenarios whose underpinnings cannot be directly eradicated (Koren and Krishna, 2020). Lastly *decentralized* refers to systems comprising a multitude of components which are not collectively controlled (synchronized nor monitored) by a central entity (Narayanan, Bonneau, Felten, Miller, and Goldfeder, 2019). These can be contrasted with centralized, transactional and fault tolerant systems, e.g. PayPal (Narayanan et al., 2019) or other escrow service providers. These actors mediate the exchange of some form of currency for goods or services between mutually untrusting parties, which entrust, instead of each other, the central third-party to oversee the transaction. In this scenario both the main parties delegate the fulfillment of their respective interest to the central third-party, so the latter must enjoy of an established endorse-

ment guaranteeing it does not collude with either of the main parties. Keeping a 'clean track' in these terms is in the interest of the escrow party itself in order for the main parties to endure their trust in the service.

Centralization in these systems is key in several respects: from an infrastructural standpoint, while exposing a potential single point of failure, a clear locus of responsibility makes it easier to ensure the unaltered unfolding of the transaction: PayPal servers are only directly accessible by employees trusted with sufficient clearance (system administrators), which are dis-incetivized by clauses in their employment contract to tamper with the system threatened by the legal consequences of their infringement including the loss of their income source and associated social position.

In economic terms the central position is the one affording a novel profit opportunity. The service provider can obtain profits from every successful transaction (e.g. from fixed or proportional fees), which stimulates the emergence of these actors in the first place, offering an incentive to take on the responsibility of performing the transaction with its associated risk and potential burden of refunding damaged parties.

Finally one has to consider the psychological aspects involved when establishing the trustworthiness of a system: the general public is acquainted to 'institution-based trust' (Pavlou and Gefen, 2004) typically and historically associated with centralized system such as banks investigated by Fungáčová, Hasan, and Weill (2019), and marketplaces by Wingreen and Baglione (2005). While the decentralized alternatives promise 'trust-less trust' (Werbach, 2019; Harz and Boman, 2018), that refers to the objective cryptographical properties of a complex digital system and not to the subjective feeling of *perceived trust* and *perceived security* (Khazaei, 2020) the customer ascribes to the entity or technology, factors which influence their adoption.

1.1 `Bitcoin`, relating to definition

A natural domain of application for this technology is currency. Nakamoto's Bitcoin allows untrusting peers to safely exchange the homonymous digital currency without the need for a central third party administering the transaction. All the peers keep an up-to-date copy of the transaction ledger, which is a monotonically increasing, immutable 'chain' of blocks, from which the name 'Blockchain'. Blocks are data-structures emitted at regular intervals, each storing at least the set of new transactions, a timestamp and the id of the previous block, thus forming the 'chain'.

While Bitcoin was the first implementation of this concept on digital computers, relying on the original notion of *computer*, a person carrying out instructions, allows us to frame (Friedman, 1991; Lanchester, 2016) the monetary system of the Micronesian island of Yap documented in 1903 (Furness, 1910), as a precursor of modern Blockchain technology. Due to the lack of minerals on the island, the Yapese decided to mint their currency form a nearby island in the form of large stones (up to 4 meters diameter). Due to the inherent transportation problems, physical possession of the 'coin' was not required to use it, the mere acknowledgment of the islanders that the transaction happened was sufficient, and the heavy rock would remain in its original location untouched.

1.2 Beyond currency

Since 2015 a revolutionary Blockchain based platform called Ethereum emerged (Buterin, 2014). The novelty it introduces is the possibility to perform general purpose, Turing complete computation within its 'smart contracts'. This allows to implement any software in a decentralized fashion, not only currency related applications. Although in 2020 this is not yet the most practical or convenient solution, it paves the road towards a new era of software services.

While the domain of digital (cash-less) currency was the first to emerge and the most widespread, the applications of Blockchain technology are countless: KYC (identity verification as a service), Immutable Data sharing (IPFS, Filecoin), elimination of paper trail in supply chain monitoring, trust establishment in IoT systems, smart grids, accountability and transparency in Governance/lawmaking systems, digital voting, Copyright and royalties protection/enforcement, and even decentralized social media (Kim and Chung, 2019).

The landscape of Blockchain technology since 2008 developed into an heterogeneous ecosystem with more than 5000 cryptocurrencies

and an overall market capitalization larger than 257,000,000,000 USD as of may 2020 (https://coinmarketcap.com/). In this context scholars exposed the need for establishing standards in the development, classification and assessment of these systems (Xu et al., 2017; Kampakis, 2018; van Moorsel, 2018).

Xu et al. proposed a taxonomy of such systems capturing the impact of many of the possible design choices on performance and main qualitative aspects of a desirable Blockchain. van Moorsel on the other hand prompted for the need of effective benchmarks, besides model based predictions, with particular attention to energy consumption, an issue which cannot be ignored for a modern technology that in its mainstream variant (POW) competes with developed nations consuming 93.10 TWh/year (Bitcoin network) just below the Netherlands 108.80 TWh/year (data from https://www.cbeci.org/comparisons/ January 2020)

2 Consensus Protocol

A crucial part of a blockchain implementation is its consensus protocol, used for block proposer selection, a form of leader election: at each tick one of the peers needs to be chosen to append the next block to the chain. The selected proposer will typically be compensated with a combination of transaction fees and freshly minted tokens.

The choice of consensus protocol determines how the agreement on the elected proposer is reached. Bitcoin and the vast majority of active blockchains as of 2020 rely on a variations of a consensus protocol based on the concept of a 'proof of work'.

A 'proof of work' is some piece of data that an agent within the network has to present attached to the proposed block. A proof of work should always be easy to verify, but it must be hard to produce: it demonstrates that the proposer invested effort before making the proposition, typically in terms of computational power (and associated energetic consumption).

The proof has a difficulty level which controls how hard it is to produce a valid instance. The difficulty of the proof is varied to keep issuance rate approximately constant.

The text book example application of proof of work is preventing e-mail spam (Back et al., 2002): if the mail protocol required a proof of work to be submitted along with each sent email, regular emails with few recipients would proceed nearly unaffected, but mass diffusion to thousand of recipients would require large amounts of computational power (or a very long time), placing dis-incentives for spam producers.

An elegant way of producing a proof of work is to ask the worker to find a value (nonce) whose cryptographic hash (a bit array, produced by some predefined hashing algorithm) is smaller than some target integer. As by design there is no clever way to solve this puzzle, the worker is forced to approach it by trial and error, progressively varying the value until it hashes below the target. Bitcoin uses 'Hashcash' to perform this step.

In a POW Blockchain each block contains its proof of work nonce, that functions as ID of the block, as well as the hash of the preceding block, thus forming the 'chain'. This forces anyone who wants to 'fork' the chain, tampering with a block, to regenerate all blocks after that which involves recalculating all of their proofs of work.

The first peer to produce a valid nonce is elected as block proposer, thus favoring nodes with more computational power. This approach has certain desirable properties making it 'robust to security threats' (Fanti et al., 2019), but is very energy-inefficient and due to the dedicated hardware required to be a competitive POW miner has encouraged the formation of large mining pools, threatening the principles of a decentralized peer to peer network (Eyal, 2015).

In POS proposer selection is driven by the share of tokens each node holds rather than the share of computational power.

Each validator node will typically 'stake' a variable amount of tokens, which are deposited and locked. These may be seized in case of misbehavior of the node. For each node their portion of the total amount of staked tokens, determines their selection probability, in a pseudo-random sampling from the validator set. Once a validator is elected it is assigned the right to propose the next block and reap associated rewards.

Note that, contrarily to POW where any node can at any moment start competing to validate the next block, in POS an explicit staking transaction has to be performed by the node, deposit-

ing their tokens, and adding their stable address to the validator set. Knowing the size of the set and the identity of its members affords these scheme additional fault tolerance properties (Patnaik and Balaji, 1987).

The amount of actual computation a POS validator has to perform is appreciably smaller than that required for a POW validator as it involves no brute-force hash generation, and opens the possibility to participate even with consumer tier hardware.

The economic investment for a POW miner, being the mining hardware, is replaced in POS by the staked tokens: in both schemata to gain validation rights a peer must demonstrate its economic investment, indirectly for POW (via hardware and high consumption rates) and directly for POS (via deposited tokens).

POS solves the energetic inefficiency of POW, contributing to an overall faster transaction processing rate, and thanks to the reduced energy consumption the required amount of tokens to be minted in order to incentivize validators to keep working is smaller.

In terms of 'game-theoretic mechanism design' (Buterin, 2014) POS offers more opportunities to discourage the formation of mining 'cartels', moreover the possibility to effectively punish misbehaving nodes: while after carrying out a 51% attack on POW the protocol cannot physically seize the malicious hardware, a POS protocol can revoke the validator rights to a node, by 'burning' its staked tokens as a punishment.

Being relatively young compared to POW, the community is still working towards POS models which provide both a desirable security level and appropriate incentive for validator nodes. The scheme should, as for POW, make sure the expected reward for a node compensates their resource usage. Another aspect though is the fairness of the rewards: peers should not only be compensated correctly on average, the variance of the reward distribution also matters (Fanti et al., 2019). As nodes can directly reinvest their rewards as new stake, wealth compounds in these systems leading to a 'rich get richer' effect.

Inspired Fanti et al.'s (2019) notion of equitability to measure the fairness of a particular scheme, the goal of the present research is to investigate the problem of wealth compounding in POS schemes; through an experimental setup manipulating simulation parameters associated with several implementation choices of a POS scheme, we aim at determining which are the most critical for the insurgence of wealth compounding and how the phenomenon can be mitigated.

3 Method

In order to study the wealth compounding dynamics a computer simulation was designed. It follows the model from Fanti et al. (2019) with some adaptations.

The model does not need to account for nodes in the network which do not participate in the validation process. These are the nodes which did not deposit stake into the system, thus they are not involved in the reward mechanism.

For convenience we denote $[x] := \{0, 1, 2, ...x-1\}$ for a positive integer x; to indicate the sum of the entries in a vector $\vec{x} \in \mathbb{R}^n$, $\sum_{i=1}^{n} x_i$, we write the inner product of $\vec{x}$ and $\vec{1} \in \mathbb{R}^n$: $\langle \vec{1}, \vec{x} \rangle$. To indicate the mean value of the entries in a vector $\vec{x}$ we write $E[\vec{x}]$.

3.1 Model definition

The model considers m parties $A = [m]$ participating in the validation process, so the ith node is $\mathrm{A_i}$. Simulation time is discrete, and an epoch is indicated by $n \in [T]$ within a total period of length T. Then $\forall i \in [m], \forall n \in [T]$ let $S_{A_i}(n)$ indicate the total stake held by party A_i at time n, and $\vec{S}_A(n)$ the vector collecting these total stakes. We also denote the sum of absolute stakes at time n, $S(n) = \langle \vec{1}, \vec{S}_A(n) \rangle$.

At time n the state vector of the system is $\vec{v}_A(n) \in \mathbb{R}^m$ where:

$$\vec{v}_A(n) = \frac{\vec{S}_A(n)}{S(n)} \quad \text{with} \quad \langle \vec{1}, \vec{v}_A(n) \rangle = 1 \quad \textit{and}$$

and

$$0 < S_i(n) \leq 1 \; \forall \; i \in [m]$$

representing the vector of fractional stakes, where each absolute stake is scaled by the total stake $S(n)$. Hence the for node A_i at time n its fractional stake is denoted $v_{A_i}(n)$ for $i \in [m]$. The

simulation can be framed as a random process that moves the state vector in $\mathbb{R}^m_{(0,1]}$.

Note that by construction the mean fractional stake for every epoch n is constant: $E[\vec{v}_A(n)] = \frac{1}{m}$, this is because the sum and number of parties stay fixed. This implies we get no information from looking at the mean fractional stake when assessing *equitability* we identify alternatives metrics capturing the construct in section 3.2.

The following assumption was made: $S(0) = 1$, therefore for $\vec{S}_A(0) = \vec{v}_A(0)$, that is initially the absolute stake for a party is equal to their fractional stake. This assumption does not imply a loss of generality as the random process is invariant to scaling (Fanti et al., 2019).

In order to generate initial state $\vec{v}_A(0)$ for m nodes we use a function of two arguments: the node index and the total number of nodes mapping to a positive real number in $(0, 1)$. Any function $stake_f$ for which

$$\sum_{i=0}^{m-1} stake_f(i, m) = 1$$

$$\text{such that} \quad \forall i \quad 0 < stake_f(i, m) \leq 1$$

is a valid generator for the initial population.

We can initialize all the m nodes to have each a fraction of the total stake equal to $\frac{1}{m}$ (setting labeled *eq*) thus producing an initial state with null variance $Var(\vec{v}_A(0)) = 0$.

Another choice is to sample m values from some distribution, and then normalize the vector of fractional stakes. The *beta* and *Pareto* distributions were considered for this purpose; *beta* is defined for x ∈ [0,1] and it was chosen since it introduces some degree of variance in the initial population, it has 2 parameters α and β controlling its shape. Setting $\alpha = \beta = 2$ yields a symmetry with respect to the vertical line $x = 0.5$, in this case the initial population variance amounts to

$$Var(\vec{v}_A(0)) = \frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)} = \frac{4}{16 * 5} = 0.05$$

The *Pareto* distribution was chosen for its relation to wealth distribution in general; it has a single shape parameter α, and introduces high asymmetric variability with a long right tail. If the α parameter is set to 1.16 the notorious *80-20* rule holds, thus 80% of the wealth is held by 20% of the population. Because of the heavy right tail, the mean, variance, and other moments are finite only if the shape parameter a is sufficiently large (i.e. $\alpha > 2$ for the variance to converge, $\alpha > 1$ for the mean).

Given that we fix the shape parameters for the distributions, we treat the choice as a categorical $stake_f \in \{eq, beta, pareto\}$.

3.1.1 The reward mechanism

1. Selection

   At each $n \in [T]$ the system elects a proposer node denoted $W(n) \in A$ so that:

   $$W(n) = \begin{cases} A_0 & \text{w.p.} \quad selection_f(\vec{v}_A(n))_0 \\ ... & \\ A_{m-1} & \text{w.p.} \quad selection_f(\vec{v}_A(n))_{m-1} \end{cases}$$

   where $selection_f$ is a function $\mathbb{R}^m_{(0,1]} \to \mathbb{R}^m_{(0,1]}$ transforming the discrete probability distribution over A, maintaining $\langle \vec{1}, selection_f(\vec{v}_A(n)) \rangle = 1$.

   The simulation accounts for *linear* and *log* as choices for $selection_f$ (figure B.1 in appendix):

   - *random*: $\vec{y} = \frac{\vec{1}}{dim(\vec{x})}$
   - *linear*: $\vec{y} = \vec{x}$
   - *log*: $y_i = \frac{log(1+x_i)}{\sum_{j=1}^m log(1+x_j)}$

   As for $stake_f$ parameters are fixed for $selection_f$ so its choice is treated as categorical variable with 3 possible labels.

2. Reward

   The system can dispense a total of R tokens during the T epochs. $W(n)$ is compensated with a reward $r(n)$ which is added to their stake:

   $S_{W(n)}(n + 1) = S_{W(n)}(n) + r(n)$

   The reward function $y = r(n) : \mathbb{N} \to \mathbb{R}$ yields for each epoch n a scalar value representing reward to emit for block n, satisfying: $\sum_{n=1}^T r(n) = R$.

   We define the *load factor* c as the mean epoch reward $c = \frac{R}{T}$

   The choice of reward function constitutes the last categorical variable; the simulation accounts for constant and geometric reward functions (plot in appendix figure: B.2):

- *const*: $r_c(n) = c$
- *geom*: $r_g(n) = (1+R)^{\frac{n}{T}} - (1+R)^{\frac{n-1}{T}}$

Table 3.1: Summary of independent parameters

name	type	desc
m	Int	number of staking nodes
T	Int	total epochs
R or c	Float	Total reward (R) or load (c)
stake_f	categ	initial stake distribution
$\text{selection}_\text{f}$	categ	proposer selection function
reward_f	categ	reward function

The combination of selection_f and reward_f leads to 5 conditions we refer as: *random* which uses random selection and constant reward, used as control condition; *const* which uses linear selection and constant reward; *geom* with linear selection and geometric reward; *log_const* with logarithmic selection and constant reward; *log_geom* that uses logarithmic selection and geometric reward. We refer to this composite label (with 5 possible values) as *sim*.

3.2 Response variables

Several metrics can be chosen to measure the equitability of a given system. A valid metric for equality is a function $f : \mathbb{R}^m \to \mathbb{R}$ of the state vector that has a lower bound at 0, capturing perfect equality where each of the m nodes has exactly $v_{A_i}(n) = \frac{1}{m}$, and some defined upper bound capturing perfect inequality where a single node has all the stake and the rest share nothing.

Both the variance (and therefore the standard deviation) and the Gini coefficient (Gini, 1912; Dorfman, 1979) satisfy these requirements.

3.2.1 Metrics

The Standard Deviation of the state is given by

$$\sigma : \mathbb{R}^m \to \mathbb{R}$$

$$\sigma(\vec{s}) = \sqrt{Var(\vec{s})} = \sqrt{E[(\vec{s} - \bar{s})^2]}$$

$$= \sqrt{\frac{\sum_{i=1}^{m}(s_i - \bar{s})^2}{m}}$$

So the portion under the square root is alwyas non-negative being a paraboloid in $\mathbb{R}^{m+1}$ with a single minima where $s_i = \bar{s}\ \forall i \in [m]$ and no maxima. If we consider the square root then we get an $m+1$ dimensional cone with still a single minima and no maxima. Therefore there is no upper bound in the general case, but our system constraints impose that $\vec{s}$ must satisfy

$$0 < s_i \leq 1\ \forall\ i \in [m]$$

and

$$\langle \vec{1}, \vec{s} \rangle = 1$$

to be a valid state vector of the system ($\vec{v}_A(n)$).

When $\vec{s}$ respects the constraints then the upper bound for $\sigma(\vec{s})$ is $\frac{1}{m}(1 - \frac{1}{m})$ (proved in appendix A, lemma A.1).

The Gini coefficient is more traditionally used to assess inequality of wealth

$$G(\vec{s}) = \frac{\sum_{i=1}^{m}\sum_{j=1}^{m}|s_i - s_j|}{2m^2\bar{s}}$$

for m nodes the value is always in $[0, 1 - \frac{1}{m}]$. When wealth is equally shared the value for the coefficient is 0. When all the wealth is concentrated in the hands of single party the upper bound $1 - \frac{1}{m}$ is reached and it has the natural interpretation of being the complement of the share under perfect equality.

An important difference is that the Gini coefficient measures the **mean distance** between data-points while variance (and standard deviation) measure the average **distance from the mean**; this has consequences on the response of the metrics to particular conditions.

3.2.2 Metrics response

While both metrics assign a value of 0 for perfect equality situations, their behavior is different under maximal inequality when the number of nodes m is varied. In fact when looking at the variance under maximal inequality for $\vec{s} \in \mathbb{R}^m$ the limit for the upper bound is

$$\lim_{m\to\infty} \frac{1}{m}(1 - \frac{1}{m}) = 0$$

.

On the other hand the upper bound in the same scenario for the Gini coefficient:

$$\lim_{m\to\infty} 1 - \frac{1}{m} = 1$$

a plot of this effect is presented in appendix B figure B.4.

Another difference in metric response regards the influence of outliers. To show this we consider a scenario where the number of parties m is fixed and the stake of a single party i, s_i is varied while the rest shares equally the remaining $1 - s_i$.

So when s_i is varied, decreasing starting from 1, both metrics start dropping. Both reach 0 when $s_i = \frac{1}{m}$. The interesting divergence is when $s_i < \frac{1}{m}$, where while the Gini coefficient reflects the 'injustice' towards the only outlier s_i with high scores, the variance does not punish as much those scenarios, lowering even more as m grows.

This is intuitive since variance measures the average squared distance from the mean, which in those scenarios is very low as most have a value immediately close to the mean and only one outlier has a large distance which does not influence the average much.

This behavior of the variance (and standard deviation) is desirable in some situations, as it makes it outlier 'tolerant' but not when evaluating equitability.

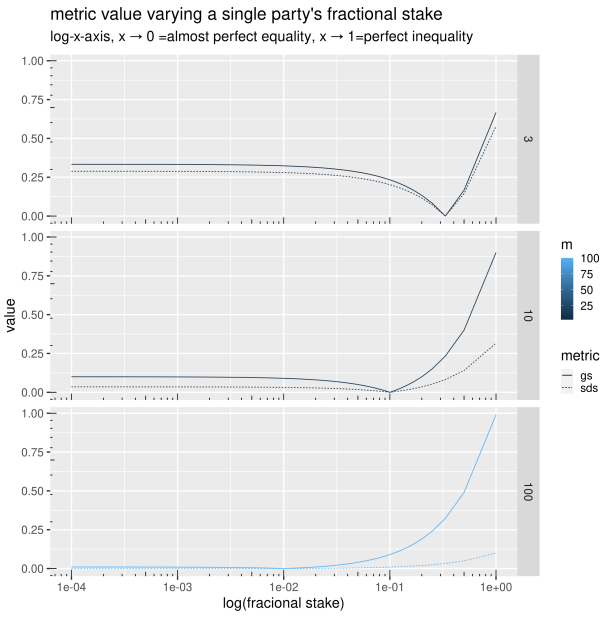


Figure 3.1: Metrics varying single party stake with fixed m

3.2.3 Scores

For a metric $f(\vec{s})$ we denote the normalized metric

$$f_r(\vec{s}) = f(\vec{s})/f_{max}$$

where f_{max} denotes f's value in the maximal inequality situation. The value $f_r(\vec{s})$ indicates the current fraction of the maximal inequality in the system state. By measuring and normalizing the value of a metric at $n = 0$ and $n = T$ we define the *score* for that simulation as the difference

$$\Delta_f = f_r(\vec{s}_T) - f_r(\vec{s}_0)$$

which denotes the gain in relative inequality that the system introduced. It is expressed as a signed fraction of the maximal value of the metrics. Hence a positive value indicates that the system amplified the inequality (e.g. 0.5 means 50% of the maximal inequality was introduced), while a negative value implies a reduction of the initial inequality. An optimal system has score 0, meaning it terminates with no inequality amplification, positive scores are not desirable as they favor some parties more than others. Negative scores are undesirable too since they would discourage larger stakeholders, as the minor stakeholders would get wealthier at their expense.

For a system initialized with perfect equality $f_r(\vec{s}_0) = 0$, the score will always be $\Delta_f \geq 0$. On the other hand for initial states with some inequality already present, negative scores can be achieved if the final metric value is smaller than the initial one.

The heat-map in figure B.3 (appendix B) shows the value of Δ_f (as the color) for any valid metric f as a function of two variables: initial (x_0) and final (x_1) normalized metric value. Optimal systems are located on the diagonal $x_0 = x_1$.

The following table shows what values were chosen for each simulation parameter. This generates 720 combinations each repeated 100 times, for a total of 72000 simulations.

In order to perform all the simulation runs in a sensible amount of time a distributed executor was built, based on the Pyhton framework 'Celery' (celeryproject.org). The executor allows to define the experiment as a dictionary data structure, from which the combinations are computed and then executed in parallel on several processes potentially on different machines. The execution environment

Table 3.2: Chosen values for independent parameters

param	values
m	10, 100, 1000
T	100, 1000
c	0.001, 0.01, 0.1, 0.5, 1, 2, 10, 100
sim	random, const, geom, $\log_{\text{const}}$, $\log_{\text{geom}}$
stake_{f}	eq, beta, pareto

to run on each machines is bundled as a Docker image, and the details regarding its setup are reported in appendix B.

4 Results

The output of a full experiment provides estimates of how our *score* statistic Δ_f is distributed conditionally on the values of independent parameters.

First we look at the scenario with initial perfect equality; here achieving negative scores is by construction impossible. All schemes achieve therefore positive scores, meaning there was an inequality gain. The errors reported in the following paragraph represent standard deviation of the measure, with a sample size of 100, therefore one can read them divided by ($\sqrt{100}$ =)10 to get standard errors.

When looking at the *eq* scenario, leaving all but the scheme type uncontrolled (see figure 4.1 and 4.4), the random scheme achieves the lowest average score (0.334 ± 0.277), while the constant reward scheme with linear selection (*const*) scores the highest (0.762 ± 0.335); the geometric scheme with linear selection (*geom*) mitigates the compounding effects scoring below *const* (0.658 ± 0.331). Using *log* selection further mitigates the amplification for both constant and geometric reward (log_{const} 0.75 ± 0.335; log_{geom} 0.648 ± 0.332)

Clearly though the variance of these averages can be decomposed by controlling for our numeric parameters m, T and c too. The multiple modes visible in the distributions in figure 4.1, lead us to suppose some parameters have drastic influence on their shape. For example let us look at the *eq* scenario fixing the number of nodes $m = 10$ and $T = 1000$, varying c in figure 4.2:

The histograms in figure 4.2 show that when c is low, score values are all gathered around 0, while

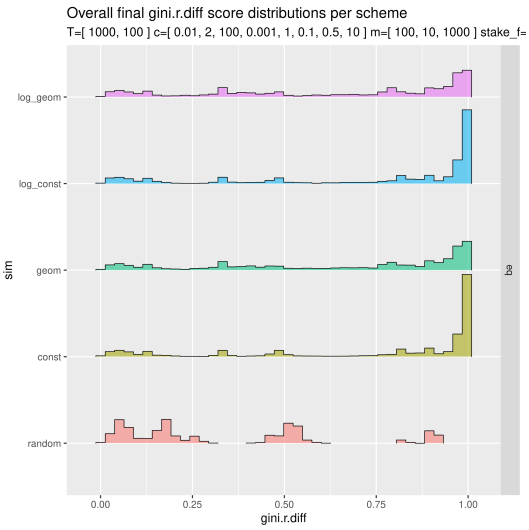


Figure 4.1: score distributions for *eq*

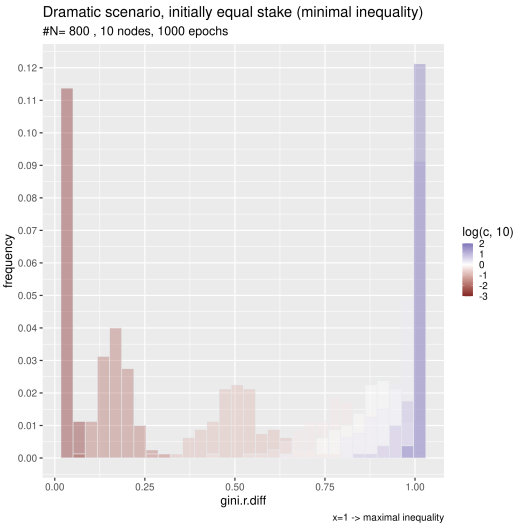


Figure 4.2: an example of variance decomposition varying c in *eq* scenario and *const* scheme

as c grows gradually the center of the distribution moves toward 1 (i.e. maximal inequality gain).

Similarly for the number of nodes (see figure 4.3) an increase of m results in score distributions shifted further right towards undesirable performance where the inequality gain is maximal.

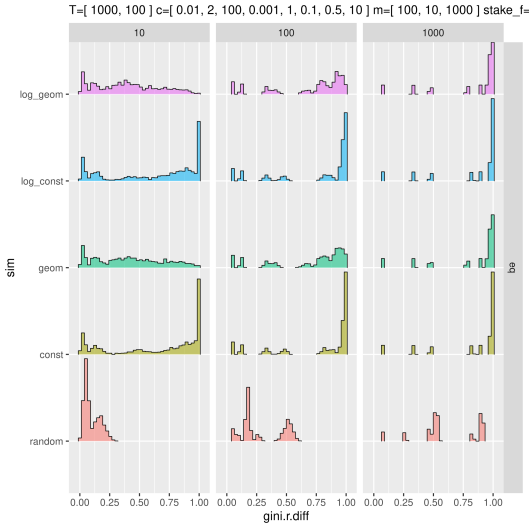


Figure 4.3: an example of variance decomposition varying m in *eq* scenario and *const* scheme

When inspecting the other initial scenarios (*beta* and *pareto* see figure 4.4) which have non null initial variance and Gini coefficient, the average amplification scores are overall lower but still positive except for the *random* scheme with *pareto* scenario (-0.18 $\pm$ 0.244); This negative score for *random* in *pareto* scenario can be attributed to the scheme rewarding equiprobably the many poor nodes of the Pareto distribution and the few rich ones, thus effectively reducing inequality.

The ranking of the schemes remains the same with constant being the worst (i.e. highest score) followed by geometric and the inadmissible random, with the *log* selection alternatives lowering slightly the scores (full table C.1 in appendix C for brevity).

4.1 Main effect

Since the number of nodes is not something fully under control when implementing a peer to peer network, we concentrate on c and T when looking

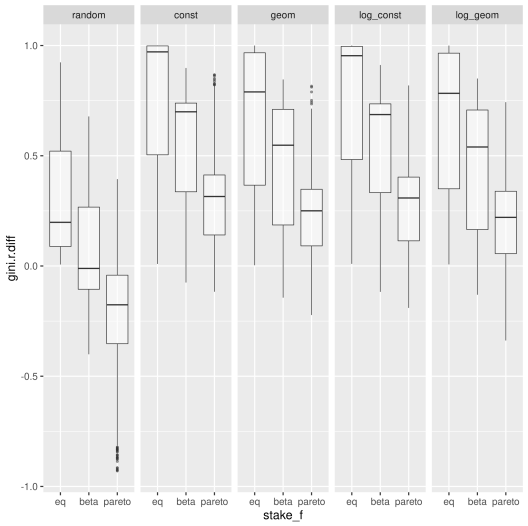


Figure 4.4: score distributions for each scheme (column) and scenario (x-axis)

for critical parameter dynamics, averaging across levels of m.

The plots in figure 4.5 show how the increase of $log(c)$ (x-axis) influences the inequality amplification *gini.r.diff* (y-axis), across levels of T (color) and scenario (row) for each scheme (column). The dashed lines represent the (log-)linear estimation of the effect.

The effect of c is clear and positive in all of the settings except for random in initially Pareto distributed populations. Interestingly the positive effect also seems to concern random schemes (see discussion). The effect is almost perfectly (log-)linear in the area preceding $log(c) = 0$ (i.e. $c = 1$: an important level discussed below), after which it inflects toward an asymptote at an height dependent on how much inequality can be gained for each initial scenario, determined by the expected initial metric value for each initial distribution (e.g. $y = 1 - 0 = 1$ for *eq*, roughly $y = 1 - 0.25 = 0.75$ for beta and $y = 1 - 0.6 = 0.4$ for pareto). Geometric schemes seem to enjoy smaller amplifications in general, which decrease even more if T is large.

An effect of T is also observed which varies across conditions: for random schemes the effect appears negative, sensibly lowering the score when increased 10 fold (from 100 to 1000); for non *random* schemes the effect interacts with that of c: in schemes with constant reward function the effect is

positive up to $log(c) = 0$ after which it becomes almost null; for schemes with geometric reward function the interaction is stronger, with a positive effect of T when c is low, which becomes negative as soon as $log(c)$ grows above -1 (i.e. $c \geq 0.1$)

To test the observed differences a linear model was fitted that predicts the *score* based on our independent variables as predictors. A complete model was constructed accounting for interactions, which was then restricted via the Akaike information criterion. The full table of regression coefficients is included in the appendix, along with assumptions diagnostic plots.

The linear model considers the score ($Delta_f$: *gini.r.diff*) modelled by scheme configuration (*random, const, geom, log_const, log_geom*) as first categorical factor with random being the reference level; the $stake_f$ factor with 3 levels (*eq, beta, pareto*) where *eq* is the reference level; and 2 numeric predictors being $log(c)$ and T.

The model explains 75.5% of the observed variance (multiple R-squared 0.7552, adjusted R-squared 0.755; F-statistic 3761 on 59 and 71940 degrees of freedom, $p < 2 * 10^{-16}$). The most relevant coefficient are presented here in the text, see appendix C for the full regression table.

The grand intercept (global reference, i.e. *random* scheme, *eq* scenario, $log(c) = 0$, $T = 0$) is established at 0.495 ($t = 115.836$, $p < 2 * 10^{-16}$), its interpretation is not intuitive as $T = 0$ does not represent a realistic scenario, but whose unit is to be read as a proportion of the maximal inequality like all the following coefficients; the baseline effect of $log(c)$ is 0.0455 ($t = 37.962$, $p < 2 * 10^{-16}$), that is from the grand intercept for each order of magnitude of c (an increase of 1 in the $log(c)$ an additional 4.5% of the maximal inequality is gained.

The baselines for each of the other schemes, being the quantity to add to the grand intercept for each condition, are as follows: for *const* 0.316 ($t = 52.271$, $p < 2 * 10^{-16}$), *geom* 0.278 ($t = 46.029$, $p < 2 * 10^{-16}$), *log_const* 0.308 ($t = 51.019$, $p < 2 * 10^{-16}$), *log_geom* 0.268 ($t = 44.337$, $p < 2 * 10^{-16}$). Thus the baseline for the *const* scheme is 1.13 times larger than that for *geom* and 1.17 times that of *log_geom*.

The baselines for *beta* and *pareto* initial distributions are respectively -0.235 ($t = -38.87$, $p < 2 * 10^{-16}$) and -0.518 ($t = -85.73$, $p < 2 * 10^{-16}$), meaning that inequality amplifications are smaller for these scenarios using random scheme.

The baseline effect for T is $-2.542 * 10^{-4}$ ($t = -42.21$, $p < 2 * 10^{-16}$). Table 4.1 shows how the effect of $log(c)$ varies when switching from *random* scheme to each of the others and when switching from *eq* scenario to *beta* and *pareto*. The estimates in table 4.1 should be added to the baseline effect of $log(c)$, thus we see that while the growth of the amplification associated with an increase of c is larger for non *random* schemes, its slope is slightly lower when the initial population already presents some degree of inequality (*beta, pareto*).

No significant interaction is associated with switching from *eq* to *beta* from each of the non *random* schemes ($p > 0.1$ see table for the individual p-values). Moreover a negative interaction of $-3.66 * 10^{-5}$ was found between $log(c)$ and T ($t = -21.7$, $p < 2 * 10^{-16}$). Finally several other minor but significant interactions were found, which are not presented in the text for brevity (see full regression table). The 4 way interaction was discarded by the Akaike information criterion as redundant for the model.

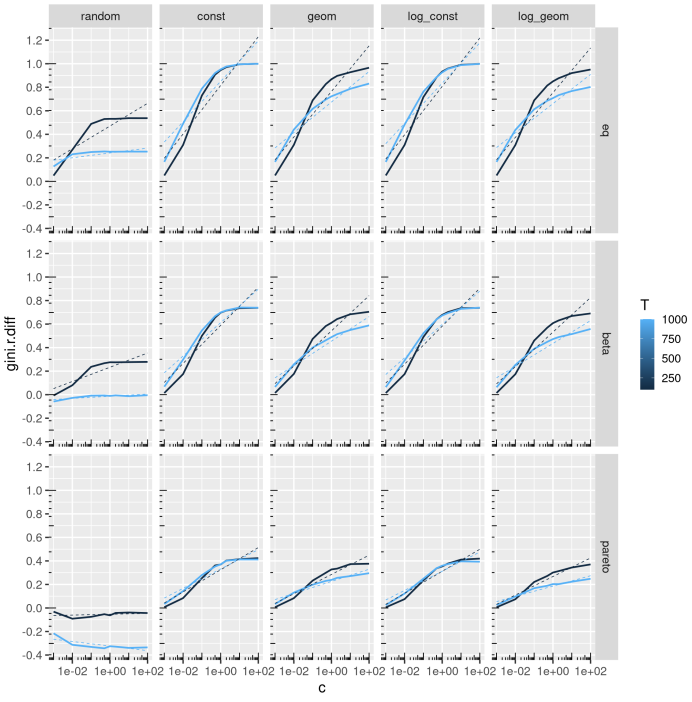


Figure 4.5: Effect of c and T for each scheme (column) and scenario (row)

Table 4.1: Interactions of $log(c)$ with *sim*

interaction	estimate	t	p
log(c) - const	0.0458	27	$< 2*10^{-16}$
log(c) - geom	0.0416	24.5	$< 2*10^{-16}$
log(c) - $\log_{const}$	0.0454	26.8	$< 2*10^{-16}$
log(c) - $\log_{geom}$	0.0401	23.6	$< 2*10^{-16}$
log(c) - beta	-0.0168	-9.907	$< 2*10^{-16}$
log(c) - pareto	-0.0428	-25.252	$< 2*10^{-16}$

The diagnostic plots for the linear model are presented in appendix B. The quantile plot (figure B.7) shows the assumptions are well respected except for the last part where our curves inflect phenomenon the linear model cannot account for; A similar observation can be made for the distribution of the residuals in figure B.8, where the last segment of fitted values is the only one breaking the assumption of normality. Finally as for the leverage plot no point falls outside of the Cook's distance limit (figure B.9).

Note that these results are relative to using the Gini coefficient as inequality metric; this choice was motivated in the method section, but a plot equivalent to figure 4.5 using standard deviation, more akin to Fanti et al.'s variance metric is attached in appendix B (figure B.5).

5 Discussion

5.1 Shift limitation

An inherent limitation of our metric is that it does not capture wealth shifts, for example if a system starts in a scenario where node 0 has 50% of the stake while nodes 1 and 2 have 25% each, and terminates with node 1 holding 50% and 0 and 2 having 25%, the metric value is the same, yielding a score of 0, as the inequality level has not changed, even if the system was unfair to 0 and favored 1 for no reason.

5.2 Effects in *random*

We observed that for *eq* and *beta* scenarios, *random* schemes still experience the positive effect of c, and the negative effect of T on the final score. One may have the opposite intuition, i.e. *random* schemes should always achieve the same score, whereas both c and T seem to have an influence. The negative effect of T can be explained as follows: we expect the random scheme (in *eq* scenario) to produce final wealth distributions that approach a binomial; smaller T implies the system has less time to converge to that binomial resulting in higher scores. The effect c for *random* schemes can be similarly justified looking at the implication of a larger c: larger reward payloads; if T was infinite, regardless of the size of the rewards, the binomial wealth distribution we expect from *random* would gradually get tighter, with progressively smaller spread, more and more indistinguishable from a round robin selection. If the time is finite though the probability that each node is rewarded the same number of times is extremely low. Thus on average some node will be rewarded a few times more than the others, and the larger the payload the more those few times weigh on the overall inequality introduced. If nodes a and b both have for example 1.8 tokens (Gini coeff. 0) and a gets compensated once more than b, if the single compensation is 0.1 the system ends up with a Gini coefficient of 0.0135, while if the compensation is ten times larger (1.0) the final coefficient is roughly 8 times larger (0.1086).

5.3 Role of $c \geq 1$

An interesting value for c is 1 ($log(c) = 0$), as 1 is the original stake pool total $c = 1$ means that the expected reward per epoch is equal to the total original pool. As we saw in figure 4.5 after this point the curves inflect and the behavior is not predicted well by the log-linear estimation after that point. In realistic scenarios no blockchain will have rewards close to or larger than the initial pool value as the first node to reap such reward immediately has more than 50% of the total stake, exposing the network to the appetites of potential attackers.

5.4 Beta and Pareto initial stake distributions

Lastly we observed that the initial stake scenario has an impact on the system's dynamics; Nevertheless exception made for the inadmissible *random* scheme, while the magnitude of the effect varies, the trend as function of c and T is very similar

across scenarios (figure 4.5). This suggests that, while the careful system designer should take into account the wealth distribution of the stakeholders, the main findings presented here generalize to several investor populations.

6 Conclusion

Several steps can be taken to account for more complex aspects of a full fledged blockchain (Xu et al., 2017), including adding the mechanism of *coin age* (Li, Andreina, Bohli, and Karame, 2017) that takes in to account holding time the tokens for their weight in the selection process; accounting for intermittent parties, that is considering that there is no guarantee that every node is always active; simulating attacks like the stake-bleeding (Gaži, Kiayias, and Russell, 2018); strategic behaviors (Fanti et al., 2019) and finally once the first large scale POS networks are deployed tweaking parameter ranges and wealth distributions to reflect the ones observed in real life scenarios. A recent paper by Wang, Yang, Bracciali, Leung, Tian, Ke, and Yu (2020) also extended the work of Fanti et al. focusing on the incentive compatibly of a POS scheme using geometric rewards. Like the present work, they also use the Gini coefficient to measure the fairness of a scheme. Wang et al. propose a variation of the geometric reward where the trade-off between incentive compatibility and equitability can be controlled by adding random 'salts' (positive or negative bonuses) to each reward. These salts are exponentially distributed with parameters chosen to have 0 expectation. The fact that geometric reward functions provide exponentially smaller rewards at the beginning of the process constitutes a disincentive for the prospect miner. The random bonuses provided by Wang et al. variation allow to fine tune by controlling the bonuses distribution the incentive compatibility of the scheme. This variation could also be easily implemented in this project's simulation.

This study builds on Fanti et al. (2019) model of a POS system, implementing an extensible computer simulation based on their analytic work. For initial stake distribution where the stake is equally divided among parties Fanti et al. derive formulas for the maximal quantity of reward that can be dispensed by some reward function in order to guarantee that the final normalized variance remains smaller or equal than a desired level $0 < \epsilon < 1$. Our work also considers initial stake distributions which already bear some degree of inequality, and we argue that a desirable system will preserve rather than increase or decrease the inequality of the initial population. This requires that we look at difference in inequality amplification rather than just its final value, as we cannot assume it is initially 0.

While Fanti et al. measure the inequality of stake with standard deviation we also introduce the Gini coefficient and argue its superiority in capturing the construct. Their work proves that the geometric reward function allows to reduce the effects of wealth compounding, our simulations empirically confirm the analytic finding with respect to our metric.

They also conclude that besides the choice of reward function the effects of compounding can be reduced by ensuring that the reward dispensed for each block (i.e. epoch) is small with respect to the initial stake pool size. We formalized the expected reward for each block as $c = \frac{R}{T}$ and denoted it as 'load factor', then since Fanti et al. show how the initial pool size $S(0)$ can be considered 1 without loss of generality, the ratio they refer to becomes simply $\frac{c}{S(0)} = \frac{c}{1} = c$. We experimentally assessed the influence of the load factor on the processes under different scheme variations and confirmed their claim by exposing its effect on our inequality amplification metric. For realistic values ($0 < c < 1$) c predicts a log-linear increase of the inequality amplification due to compounding. The effect is weaker when using geometric rewards. A plot restricted to the realistic interval averaged across T and m is presented in figure B.6 in appendix B.

While Fanti et al. also extend their work considering stake pools and strategic behavior of the participants, we left those aspects as future extensions of the simulation program, in favor of the introduction of different initial stake distribution and variations of the selection function.

In the context of the current wide array of blockchain designs, besides mathematical modeling (Fanti et al., 2019), game-theoretic design of the scheme (Liu, Luong, Wang, Niyato, Wang, Liang, and Kim, 2019), and benchmarking of the implementation (van Moorsel, 2018), this study shows the power of large scale simulations, grounded in mathematical models, when it comes to tweaking,

testing and studying the dynamics of these stochastic systems which will gradually appear in ever more aspects of society.

References

Adam Back et al. Hashcash-a denial of service counter-measure. 2002. URL `https://bit.ly/3ay9w9j`.

Vitalik Buterin. A next-generation smart contract and decentralized application platform-ethereum whitepaper.(2014). 2014.

Robert Dorfman. A formula for the gini coefficient. *The review of economics and statistics*, pages 146–149, 1979.

I. Eyal. The miner's dilemma. In *2015 IEEE Symposium on Security and Privacy*, pages 89–103, 2015. doi: https://doi.org/10.1109/SP.2015.13.

Giulia Fanti, Leonid Kogan, Sewoong Oh, Kathleen Ruan, Pramod Viswanath, and Gerui Wang. Compounding of wealth in proof-of-stake cryptocurrencies. In *International Conference on Financial Cryptography and Data Security*, pages 42–61. Springer, 2019. doi: https://doi.org/10.1007/978-3-030-32101-7_3.

Milton Friedman. *The island of stone money*. Hoover Institution, Stanford University Stanford, CA, 1991.

Zuzana Fungáčová, Iftekhar Hasan, and Laurent Weill. Trust in banks. *Journal of Economic Behavior & Organization*, 157:452–476, 2019. doi: https://doi.org/10.1016/j.jebo.2017.08.014.

William Henry Furness. *The island of stone money, Uap of the Carolines*. JB Lippincott Company, 1910.

Peter Gaži, Aggelos Kiayias, and Alexander Russell. Stake-bleeding attacks on proof-of-stake blockchains. In *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 85–92. IEEE, 2018. doi: https://doi.org/10.1109/CVCBT.2018.00015.

Corrado Gini. Variabilità e mutabilità. *vamu*, 1912.

Dominik Harz and Magnus Boman. The scalability of trustless trust. In *International Conference on Financial Cryptography and Data Security*, pages 279–293. Springer, 2018. doi: https://doi.org/10.1007/978-3-662-58820-8_19.

Stylianos Kampakis. Three case studies in tokenomics, 2018. ISSN 2516-3949.

Hamed Khazaei. Integrating cognitive antecedents to utaut model to explain adoption of blockchain technology among malaysian smes. *JOIV: International Journal on Informatics Visualization*, 4(2):85–90, 2020. doi: http://dx.doi.org/10.30630/joiv.4.2.362.

M. S. Kim and J. Y. Chung. Sustainable growth and token economy design: The case of steemit. *Sustainability*, 11(1):167, January 2019. doi: 10.3390/su11010167.

Israel Koren and C Mani Krishna. *Fault-tolerant systems*. Morgan Kaufmann, 2020.

John Lanchester. When bitcoin grows up. *London Review of Books*, 38(8):3–12, 2016.

Wenting Li, Sébastien Andreina, Jens-Matthias Bohli, and Ghassan Karame. Securing proof-of-stake blockchain protocols. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 297–315. Springer, 2017. doi: https://doi.org/10.1007/978-3-319-67816-0_17.

Z. Liu, N. C. Luong, W. Wang, D. Niyato, P. Wang, Y. Liang, and D. I. Kim. A survey on blockchain: A game theoretical perspective. *IEEE Access*, 7:47615–47643, 2019. doi: https://doi.org/10.1109/ACCESS.2019.2909924.

Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2008.

Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. Bitcoin and cryptocurrency technologies. *Curso elaborado pela*, 2019.

L. M. Patnaik and S. Balaji. Byzantine-resilient distributed computing systems. *Sadhana : Academy Proceedings in Engineering Sciences*, 111-2(11):pp. 81–91, 1987. doi: http://doi.org/10.1007/BF02811312.

Paul A Pavlou and David Gefen. Building effective online marketplaces with institution-based trust. *Information systems research*, 15(1):37–59, 2004. doi: https://doi.org/10.1287/isre.1040.0015.

Aad van Moorsel. Benchmarks and models for blockchain. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 3–3, Berlin, Germany, 2018. ACM. doi: https://doi.org/10.1145/3184407.3184441.

Yilei Wang, Guoyu Yang, Andrea Bracciali, Hofung Leung, Haibo Tian, Lishan Ke, and Xiaomei Yu. Incentive compatible and anti-compounding of wealth in proof-of-stake. *Information Sciences*, 2020.

Kevin Werbach. Summary: Blockchain, the rise of trustless trust? 2019.

Stephen C Wingreen and Stephen L Baglione. Untangling the antecedents and covariates of e-commerce trust: Institutional trust vs. knowledge-based trust. *Electronic Markets*, 15 (3):246–260, 2005.

X. Xu et al. A taxonomy of blockchain-based systems for architecture design. Number 7930224, pages 243–252, 2017. URL

A Appendix

A.1 Max var proof

Lemma A.1. *Given $\vec{s} \in \mathbb{R}^m$ such that $\langle \vec{1}, \vec{s} \rangle = 1$ and $0 \leq s_i \leq 1 \quad \forall i \in [m]$ the maximal variance $Var(\vec{s})$ amounts to $\frac{1}{m}(1 - \frac{1}{m})$ and is achieved when $\vec{s}$ has a single non-zero component with a value of 1.*

Proof. We prove the claim with two sub-proofs: the first shows that $\frac{1}{m}(1-\frac{1}{m})$ is the variance of $\vec{s}$ if it has a single non-zero component with a value of 1. The second part shows that the obtained value is maximal under the constraints stated in the lemma.
Consider $\vec{s} \in \mathbb{R}^m$, we want to show that if all entries of the vector are 0 except for one, then the

$$Var(\vec{s}) = \frac{1}{m}(1 - \frac{1}{m})$$

.
Let $\bar{s}$ mean value of $\vec{s}$ then $\bar{s} = \frac{1}{m}$
By its definition

$$Var(\vec{s}) = \frac{\sum_{i=1}^{m}(s_i - \bar{s})^2}{m}$$

Let m be the index of the one entry being 1 then:

$$Var(\vec{s}) = \frac{\sum_{i=1}^{m-1}[(0 - \bar{s})^2] + (1 - \bar{s})^2}{m}$$

if we substitute $\bar{s} = \frac{1}{m}$

$$Var(\vec{s}) = \frac{\sum_{i=1}^{m-1}[(0 - \frac{1}{m})^2] + (1 - \frac{1}{m})^2}{m}$$

$$= \frac{(m-1)\frac{1}{m^2} + (1 - \frac{1}{m})^2}{m}$$

$$= \frac{(m-1)\frac{1}{m^2} + 1 - \frac{2}{m} + \frac{1}{m^2}}{m}$$

$$= \frac{\frac{m}{m^2} + 1 - \frac{2}{m}}{m}$$

$$= \frac{\frac{1}{m} + 1 - \frac{2}{m}}{m}$$

$$= \frac{1 - \frac{1}{m}}{m}$$

$$= \frac{1}{m}(1 - \frac{1}{m})$$

Now we want to show that the above situation is the maximal variance point.
By looking at the variance

$$Var(\vec{s}) = \frac{\sum_{i=1}^{m}(s_i - \bar{s})^2}{m}$$

as a function from $\mathbb{R}^m \to \mathbb{R}$ we can see its graph is a shifted and scaled (by $\frac{1}{m}$) m dimensional hyper-paraboloid that has a single zero where all the entries have equal value $\frac{1}{m}$, and whose gradient is

$$\nabla Var(\vec{s}) = \nabla[\frac{(\vec{s} - \bar{s})^2}{m}]$$

$$= \nabla[\frac{1}{m}\vec{s}^2 - \frac{2\bar{s}}{m}\vec{s} + \frac{\bar{s}}{m}]$$

$$= 2(\frac{1}{m}s - \frac{\bar{s}}{m})$$

Thus for each dimension the ascent is linear and positive.
Our assumptions

$$\langle \vec{1}, \vec{s} \rangle = 1$$

$$0 \leq s_i \leq 1 \quad \forall i \in [m]$$

pose 2 constraints on the domain:

- the first one means $s_1 ... + s_m = 1$ which defines the hyper-plane where the sum of fractional stakes add up to 1,
- The second one ensures only the region where all entries are non negative and smaller than 1 is considered.

Under these constraints the region is an $m - 1$ dimensional hyper-paraboloid whose maxima are in the 'corner' as the linear gradient indicates for each dimension (An example in $\mathbb{R}^2$ in figure A.1). Therefore we showed that under our assumptions the situation of maximal inequality coincides with maximal variance $= \frac{1}{m}(1 - \frac{1}{m})$

□

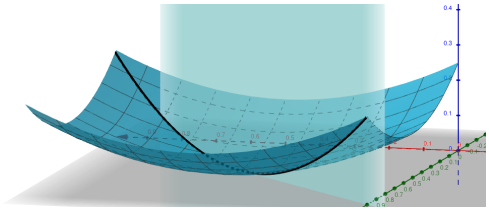


Figure A.1: Example in $\mathbb{R}^2$

B Additional Images

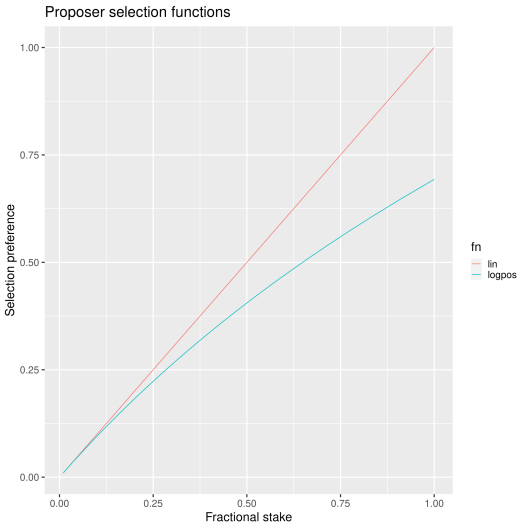


Figure B.1: Proposer selection functions

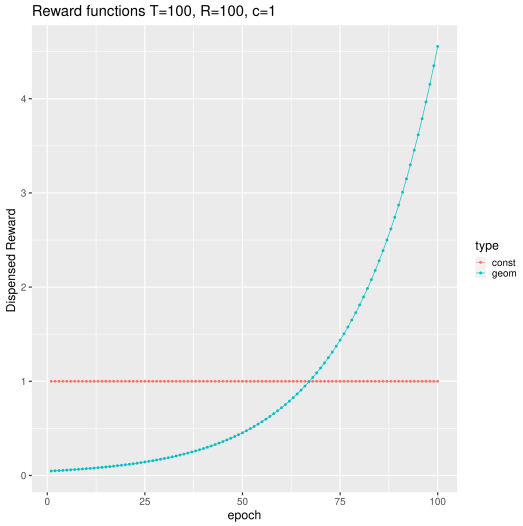


Figure B.2: Reward functions

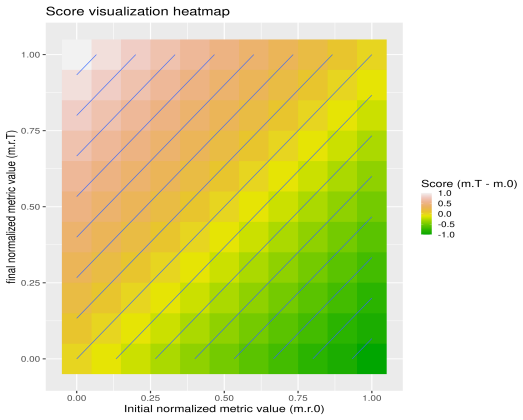


Figure B.3: Score contour

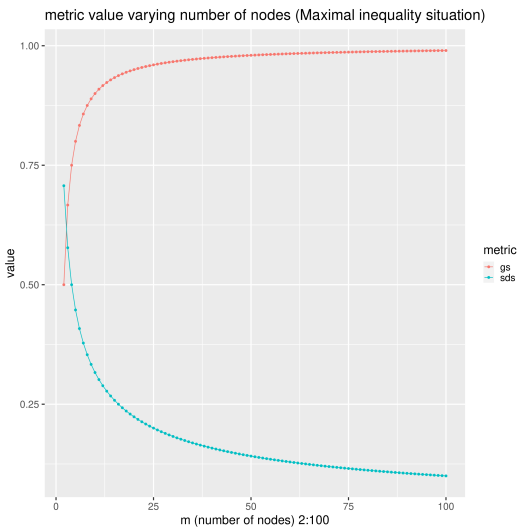


Figure B.4: Metrics under maximal inequality varying m

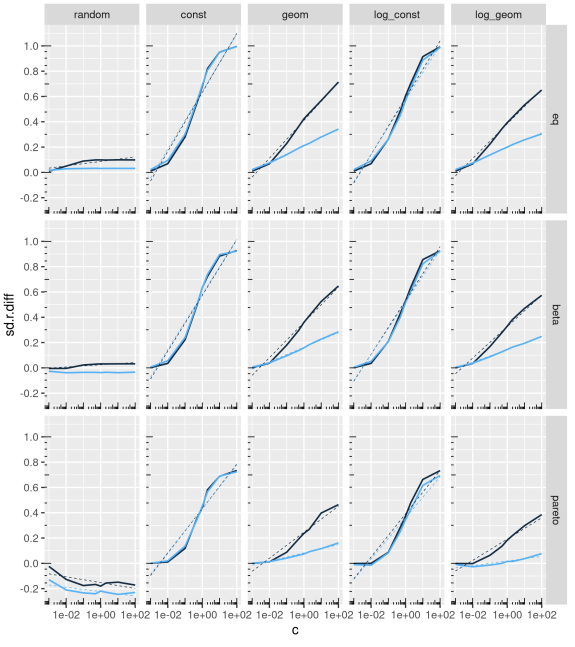


Figure B.5: Plot of the main effect equivalent to 4.5 using standard deviation as metric.

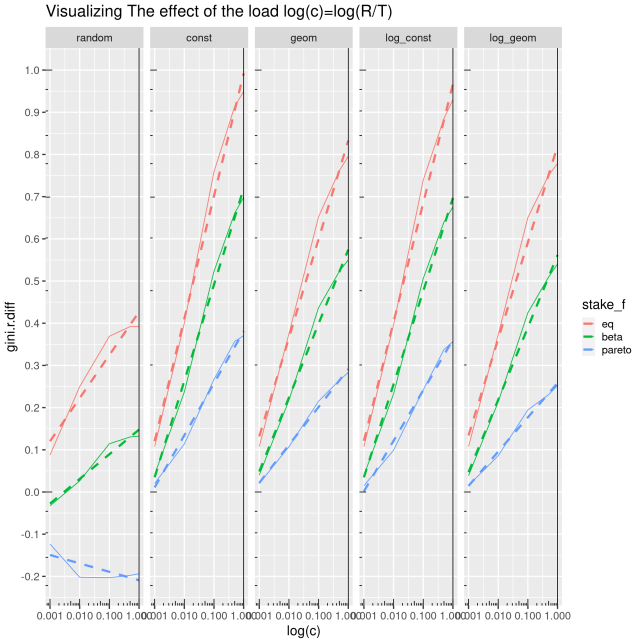


Figure B.6: Plot of the main effect, restricted to realistic values of $log(c)$, averaged across T and m

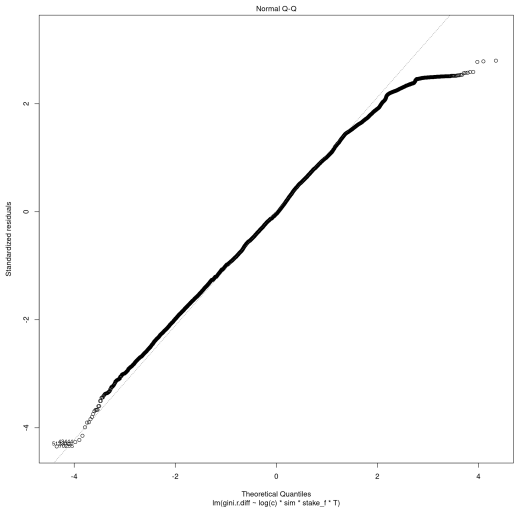


Figure B.7: Regression diagnostic plot, residuals quantile plot

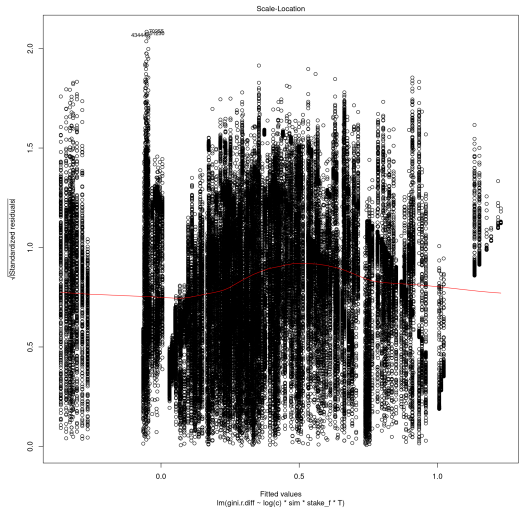


Figure B.8: Regression diagnostic plot, residuals distribution plot

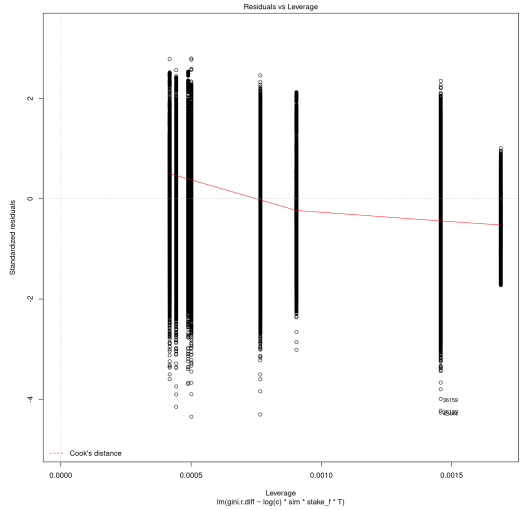


Figure B.9: Regression diagnostic plot, residuals leverage plot

C Full regression table

Appears on the next page due to absence of space here.

Table C.1: Full regression table, parenthesized numbers are the exponent of the scientific notation. Refrence levels: *sim*=random, *stake*=eq

Coefficient	Estimate	Std-Error	t-value	p-value
(Intercept)	4.958 (-01)	4.280 (-03)	115.836	< 2(-16)
log(c)	4.557 (-02)	1.200 (-03)	37.962	< 2(-16)
simconst	3.164 (-01)	6.053 (-03)	52.271	< 2(-16)
simgeom	2.786 (-01)	6.053 (-03)	46.029	< 2(-16)
$\text{simlog}_{\text{const}}$	3.088 (-01)	6.053 (-03)	51.019	< 2(-16)
$\text{simlog}_{\text{geom}}$	2.684 (-01)	6.053 (-03)	44.337	< 2(-16)
$\text{stake}_{\text{fbeta}}$	-2.353 (-01)	6.053 (-03)	-38.870	< 2(-16)
$\text{stake}_{\text{fpareto}}$	-5.189 (-01)	6.053 (-03)	-85.729	< 2(-16)
T	-2.542 (-04)	6.023 (-06)	-42.210	< 2(-16)
log(c):simconst	4.585 (-02)	1.698 (-03)	27.006	< 2(-16)
log(c):simgeom	4.163 (-02)	1.698 (-03)	24.521	< 2(-16)
log(c):$\text{simlog}_{\text{const}}$	4.549 (-02)	1.698 (-03)	26.793	< 2(-16)
log(c):$\text{simlog}_{\text{geom}}$	4.017 (-02)	1.698 (-03)	23.663	< 2(-16)
log(c):$\text{stake}_{\text{fbeta}}$	-1.682 (-02)	1.698 (-03)	-9.907	< 2(-16)
log(c):$\text{stake}_{\text{fpareto}}$	-4.287 (-02)	1.698 (-03)	-25.252	< 2(-16)
simconst:$\text{stake}_{\text{fbeta}}$	9.953 (-03)	8.560 (-03)	1.163	0.244949
simgeom:$\text{stake}_{\text{fbeta}}$	1.298 (-02)	8.560 (-03)	1.517	0.129372
$\text{simlog}_{\text{const}}$:$\text{stake}_{\text{fbeta}}$	1.109 (-02)	8.560 (-03)	1.295	0.195278
$\text{simlog}_{\text{geom}}$:$\text{stake}_{\text{fbeta}}$	1.395 (-02)	8.560 (-03)	1.629	0.103237
simconst:$\text{stake}_{\text{fpareto}}$	3.110 (-02)	8.560 (-03)	3.633	0.000280
simgeom:$\text{stake}_{\text{fpareto}}$	3.594 (-02)	8.560 (-03)	4.198	2.70 (-05)
$\text{simlog}_{\text{const}}$:$\text{stake}_{\text{fpareto}}$	2.617 (-02)	8.560 (-03)	3.057	0.002235
$\text{simlog}_{\text{geom}}$:$\text{stake}_{\text{fpareto}}$	3.121 (-02)	8.560 (-03)	3.646	0.000267
log(c):T	-3.668 (-05)	1.689 (-06)	-21.712	< 2(-16)
simconst:T	2.926 (-04)	8.518 (-06)	34.350	< 2(-16)
simgeom:T	1.538 (-04)	8.518 (-06)	18.056	< 2(-16)
$\text{simlog}_{\text{const}}$:T	2.846 (-04)	8.518 (-06)	33.408	< 2(-16)
$\text{simlog}_{\text{geom}}$:T	1.508 (-04)	8.518 (-06)	17.699	< 2(-16)
$\text{stake}_{\text{fbeta}}$:T	-1.963 (-05)	8.518 (-06)	-2.305	0.021175
$\text{stake}_{\text{fpareto}}$:T	-4.622 (-05)	8.518 (-06)	-5.426	5.79(-08)
log(c):simconst:$\text{stake}_{\text{fbeta}}$	-3.339 (-03)	2.401 (-03)	-1.391	0.164308
log(c):simgeom:$\text{stake}_{\text{fbeta}}$	-3.273 (-03)	2.401 (-03)	-1.363	0.172813
log(c):$\text{simlog}_{\text{const}}$:$\text{stake}_{\text{fbeta}}$	-3.265 (-03)	2.401 (-03)	-1.360	0.173943
log(c):$\text{simlog}_{\text{geom}}$:$\text{stake}_{\text{fbeta}}$	-2.818 (-03)	2.401 (-03)	-1.173	0.240602
log(c):simconst:$\text{stake}_{\text{fpareto}}$	-7.055 (-03)	2.401 (-03)	-2.939	0.003298
log(c):simgeom:$\text{stake}_{\text{fpareto}}$	-7.286 (-03)	2.401 (-03)	-3.035	0.002408
log(c):$\text{simlog}_{\text{const}}$:$\text{stake}_{\text{fpareto}}$	-6.928 (-03)	2.401 (-03)	-2.885	0.003910
log(c):$\text{simlog}_{\text{geom}}$:$\text{stake}_{\text{fpareto}}$	-6.958 (-03)	2.401 (-03)	-2.898	0.003758
log(c):simconst:T	1.992 (-05)	2.389 (-06)	8.338	< 2(-16)
log(c):simgeom:T	5.883 (-06)	2.389 (-06)	2.462	0.013804
log(c):$\text{simlog}_{\text{const}}$:T	2.007 (-05)	2.389 (-06)	8.401	< 2(-16)
log(c):$\text{simlog}_{\text{geom}}$:T	4.934 (-06)	2.389 (-06)	2.065	0.038922
log(c):$\text{stake}_{\text{fbeta}}$:T	1.191 (-05)	2.389 (-06)	4.985	6.20(-07)
log(c):$\text{stake}_{\text{fpareto}}$:T	2.554 (-05)	2.389 (-06)	10.690	< 2(-16)
simconst:$\text{stake}_{\text{fbeta}}$:T	7.112 (-06)	1.205 (-05)	0.590	0.554953
simgeom:$\text{stake}_{\text{fbeta}}$:T	2.303 (-05)	1.205 (-05)	1.912	0.055923 .
$\text{simlog}_{\text{const}}$:$\text{stake}_{\text{fbeta}}$:T	6.096 (-06)	1.205 (-05)	0.506	0.612837
$\text{simlog}_{\text{geom}}$:$\text{stake}_{\text{fbeta}}$:T	1.556 (-05)	1.205 (-05)	1.291	0.196599
simconst:$\text{stake}_{\text{fpareto}}$:T	1.522 (-05)	1.205 (-05)	1.264	0.206334
simgeom:$\text{stake}_{\text{fpareto}}$:T	8.136 (-05)	1.205 (-05)	6.754	1.45(-11)
$\text{simlog}_{\text{const}}$:$\text{stake}_{\text{fpareto}}$:T	1.663 (-05)	1.205 (-05)	1.381	0.167394
$\text{simlog}_{\text{geom}}$:$\text{stake}_{\text{fpareto}}$:T	5.941 (-05)	1.205 (-05)	4.932	8.16(-07)

Project README

This document acts as technical appendix for the bachelor project pos-sim-0.1
It gives hands on instruction to use the library and reproduce the experiments.
For documentation of the actual code see `http://139.162.161.39/thesis/build/html/index.html`.

Standard mode

1. Installation

   This section applies for GNU/linux systems. It should also work on OSX (mac) if a python (>3) and pip are installed.

   Clone the repo (`https://github.com/bandoos/bachelor-project`) of the project to a location we will refer to as `$PROJECT_ROOT`

   The alternative is using the provided Docker images, which will work on GNU/linux, OSX and windows. See the **Docker mode** section for instructions.

   NOTE about using virtualenv

   Using virtualenv to avoid python libraries versions conflicts is encouraged.

   if virtualenv is not installed on your system yet it can be installed with pip

   ```
   $ pip install --user virtualenv
   ```

   check installation

   ```
   $ virtualenv --version
   ```

   create an env directory

   ```
   $ mkdir sim-core-env
   $ cd sim-core-env
   $ virtualenv venv
   ```

   then from sim-core-env activate the environment

   ```
   $ source ./venv/bin/activate
   ```

   The terminal prompt should now display the name of the environment to signal it is active.

   The code is bundled as a python package so, from the root folder of the project run:

   ```
   $ pip install --user -e ./
   ```

   This will use `pip` to install (for the current user) the dependencies of the project from PyPI, and add this project to the `$PYTHONPATH` so it can be executed on your system.

   Thought this is not necessary for the basic functionality some features rely on `$HOME/.local/bin` to be in the `$PATH`. This is typically the case with standard linux distribution, but the install command will warn if this is not the case for you.

   If using virtualenv once the environment is activated the `$PATH` is manipulated appropriately automatically.

   If `.local/bin` is in your `$PATH` then the main entry point for the simulation executable is available as:

   ```
   $ sim-stake [OPTIONS]
   ```

   If `.local/bin` is not in your $PATH then from the `$PROJECT_ROOT` run:

   ```
   $ python -m sim.core.main [OPTIONS].
   ```

   In that case remember that you should always be in `$PROJECT_ROOT` and substitute `sim-stake` with `python -m sim.core.main` in the following sections.

2. Usage You can see its usage information with:

   ```
   $ sim-stake --help
   ```

```
usage: sim-stake [-h] [--id ID]
                 --m M --T T
                 --c C --times TIMES
                 --stake_f {eq,beta,pareto}
                 --sim {random,const,geom,log_const,log_geom}

Run a sim-stake-batch

optional arguments:
  -h, --help            show this help message and exit
  --id ID               unique id for the experiment

required arguments:
  --m M                 INTEGER: Indicate the number of nodes [m] (valid if >= 2)
  --T T                 INTEGER: Indicate max epoch time [T] (valid if >= 2)
  --c C                 FLOAT: Indicate total load factor "c" [R=cT] (valid if > 0)
  --times TIMES         INTEGER: Redudancy factor (valid if > 0)
  --stake_f {eq,beta,pareto} STRING: Generator function for inital stake distrib.
  --sim {random,const,geom,log_const,log_geom} STRING: Indicate simulator class
```

So the program requires a flag for each simulation parameter, plus an optional id argument. The id argument is not necessary for basic usage, and i suggest using the docker setup for batch execution anyways (which manages experiment ids independently) so it may be removed in subsequent releases.

3. Test the installation

   An example of well formed command would be:

   ```
   $ sim-stake --m 3 --T 200 --c 0.5 --stake_f eq --sim random --times 10
   ```

   Which would run the simulation with:

   - $m = 3$ nodes
   - for $T = 200$ epochs,
   - with a load factor $c = 0.5$
   - initial stake $stake_f = eq$
   - $sim = random$ scheme,
   - repeating the experiment 10 times ($times = 10$)

   A csv is produced on standard output which looks like (may overflow page on pdf):

```
m,T,c,R,sim,stake_f,var_0,var_T,gini_0,gini_T,under_target,avg_loss,over_target,avg_gain
3,200,0.4,80.0,random,eq,0.0,0.0001354807,0.0,0.0164609053,0.6666666667,-0.0082304527,0.3333333333,0.01
3,200,0.4,80.0,random,eq,0.0,0.0013222917,0.0,0.0592592593,0.6666666667,-0.0230452675,0.3333333333,0.04
3,200,0.4,80.0,random,eq,0.0,0.0007207573,0.0,0.0427983539,0.6666666667,-0.0181069959,0.3333333333,0.03
3,200,0.4,80.0,random,eq,0.0,0.0008833342,0.0,0.046090535,0.6666666667,-0.0205761317,0.3333333333,0.041
3,200,0.4,80.0,random,eq,0.0,0.0006557266,0.0,0.0362139918,0.6666666667,-0.0181069959,0.3333333333,0.03
3,200,0.4,80.0,random,eq,0.0,0.0008508188,0.0,0.0427983539,0.6666666667,-0.0205761317,0.3333333333,0.04
3,200,0.4,80.0,random,eq,0.0,5.4192e-06,0.0,0.0032921811,0.3333333333,-0.0032921811,0.6666666667,0.0016
3,200,0.4,80.0,random,eq,0.0,0.0003305729,0.0,0.0296296296,0.3333333333,-0.0230452675,0.6666666667,0.01
3,200,0.4,80.0,random,eq,0.0,0.0002655422,0.0,0.0263374486,0.6666666667,-0.0106995885,0.3333333333,0.02
3,200,0.4,80.0,random,eq,0.0,0.0005581805,0.0,0.0362139918,0.3333333333,-0.0329218107,0.6666666667,0.01
```

   All simulation parameters are reported for each row along with the observed result metrics, so that each result is fully characterized by its csv output (i.e. 2 outputs can merged in a single dataframe without loss of information)

   Use output redirection to save the results to a file for later inspection:

   ```
   $ sim-stake --m 3 --T 200 --c 0.5 --stake_f eq --sim random --times 10 > some_name.csv
   ```

(a) NOTE Running the simulation as saw above works for simple tests with a single parameters combination. For a full fledged experiment with parameter manipulation see either section 4 (using as library) or section (docker mode).

4. Using as library

   Once installed the code can also be used as library. In the module **sim.core.main** exposes a `run` function that accepts the parameters you would provide on the command line as a dictionary (without the `--` prefix on parameters name).

   In a python script of your choice:

```
import sim.core.main as simulation

params = {'m':3,
          'T':300,
          'c':0.5,
          'stake_f':'eq',
          'sim':'random',
          'times':10}

simulation.run(params)
```

   The run function accepts 2 other optional named parameters:

   - `out_fn` (default = sys.stdout.write)
   - `header` (default = True)

   The `out_fn` will be called for each simulation repetition passing a string being the comma separated values (parameter + response metrics) i.e. `times` times once per line of the output csv.

   The `header` boolean controls whether the header of the csv should be produced before the first run results.

   (a) Simple experiment

   A simple experiment can be conducted by writing a procedure that runs several simulations:

   Let's say we want to manipulate the number of nodes m:

```
import sim.core.main as simulation

ms = range(2,10)

params = {'m':None,
          'T':300,
          'c':0.5,
          'stake_f':'eq',
          'sim':'random',
          'times':10}

header = True
for m in ms:
    params['m'] = m
    simulation.run(params,header=header)
    if header:
        header=False
```

   Note that we ensure that the header is only produced on the first parameter combination so we get a valid csv as output.

5. Experiment definition grammar

   Although the above is sufficient for simple experiments, relying on procedural code may hide the essence of the experiment in complex scenarios, rendering difficult to infer what is tested. A more declarative approach ensures readability and clarity.

In order to define experiment in a pleasant way a module was defined to provide a definition grammar for complex experiments.

The fundamental idea is providing a callable data structure that represents the Cartesian product of named sets. Once called the ds will expand to a list of dictionaries where each key assumes one of the values of its set.

The `sim.executor.batch.ibatch` module provides the constructor `P` for these Cartesian expansions.

```
from pprint import pprint
from sim.executor.batch.ibatch import P

p1 = P({'a':{True,False},
        'b':{True,False}})

pprint(p1())
```

Which produces the following output:

```
[{'a': True, 'b': True},
 {'a': True, 'b': False},
 {'a': False, 'b': True},
 {'a': False, 'b': False}]
```

Typically the values of the dictionary provided to the P constructor will be sets (thus ensuring no duplicates) but any iterable or callable that returns an iterable is fine, so the following is acceptable:

```
from pprint import pprint
from sim.executor.batch.ibatch import P

def i_could_be_a_very_complex_function():
    "...complex compute..."
    return {True,False}

p2 = P({'n': range(1,4),
        'b': i_could_be_a_very_complex_function})

pprint(p2())
```

Which produces:

```
[{'b': False, 'n': 1},
 {'b': True, 'n': 1},
 {'b': False, 'n': 2},
 {'b': True, 'n': 2},
 {'b': False, 'n': 3},
 {'b': True, 'n': 3}]
```

If we only desire a segments of the product (i.e. some value should only be matched with specific ones) then chaining 2 separate P constructor suffices. To chain constructors just use the + operator:

```
from pprint import pprint
from sim.executor.batch.ibatch import P

p3 = P({'mode': {"a"},
        'sub_mode': {"a1","a2"}})

p4 = P({'mode': {"b"},
        'sub_mode':{"b1","b2"}})

p5 = p3 + p4
```

```
pprint(p5())

[{'mode': 'a', 'sub_mode': 'a1'},
 {'mode': 'a', 'sub_mode': 'a2'},
 {'mode': 'b', 'sub_mode': 'b1'},
 {'mode': 'b', 'sub_mode': 'b2'}]
```

A real experiment definition for the simulation could be:

```
from sim.executor.batch.ibatch import P
REPETITIONS=10
REDUNDANCY=2
batch = P({'m':  [10 ** i for i in range(1,4)], # 3 elems
           'T':  [10 ** i for i in range(2,4)], # 2 elems
           'c':  [0.001, 0.01, 0.1, 0.5, 1, 2, 10, 100], # 8
           'sim':        ['const','geom','log_const','log_geom','random'], # 5 elmes
           'stake_f':    ['eq','beta','pareto'], # 3 elems
           'times':      [REPETITIONS],
           'redundancy': range(REDUNDANCY) })
```

Which will generate $3*2*8*5*3 = 720$ unique parameters configurations, which are replicated `REDUNDANCY` times (thus 1440 runs) each of which tests the configuration `REPETITIONS` times (thus 14'400 total simulations).

'redundancy' in this case is a dummy key, the actual simulation will not read its value, but it still multiplies the number of generated parameter dictionaries. The reason for having both 'times' and 'redundancy' should become clear when the distributed multiprocess facility is introduced; in a single process environment one should just use 'times'.

the above experiment could be run as follows:

```
import sim.core.main as simulation
header = True
for params in batch():
    simulation.run(params,header=header)
    if header:
        header=False
```

A large experiment like the one above may take very long to terminate which is why the software is meant to be run in a distributed multiprocess fashion thanks to celery `https://github.com/celery/celery`.

6. Experiment definition convention We adopt the following convention to define experiments:

   create a python file in `$PROJECT_ROOT/executor/experiments/`

   define the experiment via arbitrary code or using the above presented grammar and assign the callable or iterable that generates the configurations to a toplevel variable called `batch`.

   Note that you can define experiments wherever you want as long as the file is in the `$PYTHONPATH` and a `batch` callable or iterable is present.

   The main experiment presented in the paper is located in module `sim.executor.experiments.exp_0`.

   This convention will be important later on in section 3.

Docker mode

If not already present on your system install docker: `https://docs.docker.com/get-docker/`

On linux you may want to use your usual package manager. On linux, after installation, you need to add your user to the `docker` group to be able to run docker images without root privileges. (This is strongly encouraged rather than using sudo!!)

```
# usermod --append -G docker <your-user>
```

On macos and windows (using the desktop version of docker) the docker-compose utility ships by default. On linux you will have to install it separately: `https://docs.docker.com/compose/install/`

It quiet intuitively allows to compose docker images/containers.

1. Ensuring docker installation

   Test the docker installation

   ```
   $ docker run --rm hello-world
   ```

   This can take a while the first time, but it should then produce some useful information about docker and exit.

2. Installing the project's image

   The docker image for this project ships with a fully functional archlinux system with all the necessary requirements installed plus some packages and tweaks to make the experience pleasant like tab-completion on the project's commands (see `$PROJECT_ROOT/Dockerfile`).

   Using a pre-built image is suggested; download it from `https://zenodo.org/record/4543831/files/pos-sim-core-latest.tar.gz?download=1` (to check the sha sums see section .)

   The compressed image is about 1 GB.

   once downloaded load it to the docker engine with

   ```
   $ docker load < pos-sim-core-latest.tar.gz
   ```

   ** Launch the system Once the image is successfully loaded enter the `$PROJECT_ROOT/compose` folder and run:

   ```
   $ docker-compose up
   ```

   This will start the container and mount the `$PROJECT_ROOT/compose/data` directory to the container's `~/data` dir. This location can be used as a (persitent) bridge between your system and the container.

   The above command will hang until you decide to stop it, when so hit CTRL-C to send the shutdown signal, the system will process it and shutdown gracefully.

   Note this is named a container so only one instance at a time can run, that is more than sufficient to run many simulations in parallel within the container though!

3. Start a session You can start a terminal session within the running system (from another terminal) with

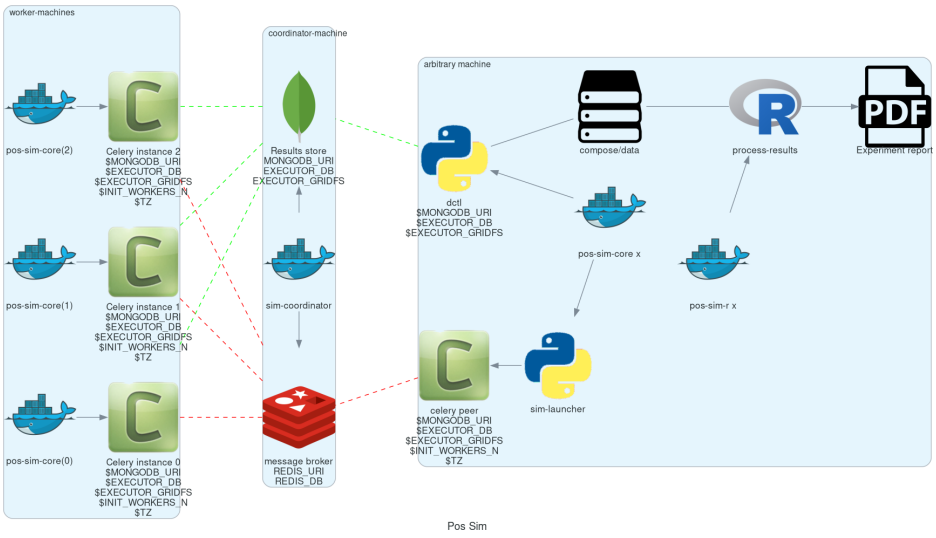
   ```
   $ docker exec -it pos-sim-core /bin/zsh
   ```

   This will open a terminal within the container.

   Inside you find a copy of `$PROJECT_ROOT`.

   All of the project commands are in the `$PATH` there so they can be called directly. If in doubt you can list them with `$ ls ~/.scripts`

Multiprocess distributed execution



To allow for large scale simulations facilities are provided to run multiple simulations in parallel on multiple machines thanks to Celery (v4.4.3) `https://docs.celeryproject.org/en/4.4.3/getting-started/resources.html` coordinated by Redis `https://redis.io/` and storing results on Mongodb `https://www.mongodb.com/`.

While a setup without docker for this use case is possible it involves installing the project, mongodb and redis to your system, and since the purpose of this facility is to deploy easily on several possibly heterogeneous systems the easiest and more reliable solution is to just have a docker engine on each machine and rely on the provided images.

Note that no knowledge about redis or mongodb is required to carry out the experiments as utilities are provided for the necessary interactions.

1. Coordination

   On one machine the **sim-coordinator** system should be run. Assuming docker and docker-compose are available on the machine simply enter `$PROJECT_ROOT/sim-coordinator` and run

   ```
   $ docker-compose up
   ```

   Note that this uses the official redis and mongodb images so no `docker load` is needed in this case.

   This will start the database and redis instances on predefined ports (see section 5 if you want to change the port numbers for any reason.)

   the above command will hang until CTRL-C is pressed which will start the graceful shutdown.

   The workers running the project's code will receive jobs to execute from redis and produce results to the database.

   Inside of `$PROJECT_ROOT/sim-coordinator` 2 folders are present:

   - `$PROJECT_ROOT/sim-coordinator/mongo-volume`
   - `$PROJECT_ROOT/sim-coordinator/reids-data`

   Similarly to `$PROJECT_ROOT/sim-coordinator/compose/data` these act as bridges with your host system. The database will persist the data the **mongo-volume** dir and redis (which by default is not persistent) will do so in the **redis-data** dir if configure to be persistent.

   No further actions need to be taken with regard to the coordination system.

2. Workers

   On each machine that should be targeted by the job distribution mechanism follow sections 2 and 3 to boot the worker environment.

   Once you have a session terminal ensure that the system configuration is correct for your needs (see section 5), and then simply run:

   `$ run-worker`

   to have the machine join the distributed system. This will hang until you hit CTRL-C, and will print information about the system and then log events.

3. Launcher

   A launcher is provided in the module `sim.executor.launcher` which is linked in `.scripts/sim-launcher` for convenience.

   You can use the launcher from any of the machines that have a running (and correctly configured) instance of the project's docker image. Another option is launching from a machine (e.g. a laptop) that will not have a worker running so long as it is properly configured to contact the distributed system coordinator (see section 5).

   it synopsis is as follows:

   ```
   usage: sim-launcher [-h] [--exp-module EXP_MODULE] [--async]

   optional arguments:
     -h, --help            show this help message and exit
     --exp-module EXP_MODULE
     --async, -a
   ```

   The `--exp-module` option controls which experiment will be loaded and distributed on worker machines. If not provided a small default experiment is chosen to test the system. The value provided for the experiment module should be a fully qualified python module name such as `sim.executor.experiments.exp_0` just like in an **import** statement, pointing to a module in the `$PYTHONPATH`. The `batch` variable within that module will be looked up according to the convention presented in section 6.

   If `--async` is not provided then the launcher will block until the experiment completes. If `--async` is provided then the launcher will exit as soon as the dispatching completes, you can then monitor the progress as explained in section 6

   Once an experiment is successfully launched the coordinator will distribute the necessary jobs to complete the experiment to the available workers.

   The launcher program outputs some information about the dispatched experiment. In particular it outputs a python dictionary whose `batch_uuid` key is what we are interested in for fetching results later on as explained in section 4. (If the `--async` flag is on the look for `_batch_uuid`)

4. Retrieving results

   In distributed mode the database is used to store results as they are produced.

   Once an experiment is finished you can use the utility provided in `$PROJECT_ROOT/.scripts/dctl` that helps to fetch all the aggregated results of a full experiment from the database as a csv.

   Within the docker environment this is linked to `~/.local/bin` so you can use it directly

   `$ dtcl [cmd] [options]`

   in custom environment from `$PROJECT_ROOT` use it by invoking the full with path

   `.scripts/dtcl [cmd] [options]`

   It provides 2 cmd(s):

   - `dctl fs ls` List the experiment results csv that are available in the system.
   - `dctl fs get <batch_uuid>.csv` Get a result by name.

Note that from the docker environment tab-completion is available for the file name, so you just need to remember the first few characters of the `batch_uuid` and then press tab to complete.

Redirect the output of `dctl fs get` to a file in to save the results. If you are running dctl in the docker environment redirect to `~/compose/data/<filename>.csv` to have the results visible on the host system. (Remeber `compose/data` acts as bridge - so called docker volume - between the virtual system in the docker and your host system).

```
$ dctl fs get batch_uuid.csv > destination/name.csv
```

substitute `batch_uuid`, `destination` and `name` appropriately.

5. Configuring the distributed system

   The distributed system is configured via the following environment variables:

   (a) `MONGDB_URI` Defines the address of the database in the following format:
   mongdb://<ip-adress>:<ip-port>
   so for example assuming the coordinator was launched on a machine on 192.168.178.31 on the default port:
   mongodb://192.168.178.31:27020
   default **mongodb://0.0.0.0:27020**

   (a) `EXECUTOR_GRIDFS` The name of the internal database to use as distributed filesystem, the default is **executor-gridfs**

   (b) `EXECUTOR_DB` The name of the internal database to use for task metadata and partial results, default is **from-celery**

   (c) `REDIS_URI` Similar to `MONGODB_URI` but for the redis server, default is **redis://0.0.0.0:6399**

   (d) `REDIS_DB` The number (redis uses integers to identify the dbs) of the redis internal database to use. Default 2

   (e) `INIT_WORKERS` The number of workers (processes) to run concurrently if the machine is used as worker node.

   (f) `TZ` The timezone to use (must be consistent on all machines for proper coordination). Defaults to **Europe/Amsterdam**, must be a valid timezone value.

   Ideally you want to modify only the URI(s), `TZ` and `INIT_WORKERS`.

   the suggested manner of configuration is putting all the values in a .env file like the following:

```
MONGODB_URI=mongodb://0.0.0.0:27020
EXECUTOR_GRIDFS=executor-gridfs
EXECUTOR_DB=from_celery

REDIS_URI=redis://0.0.0.0:6399
REDIS_DB=2

INIT_WORKERS_N=4

TZ=Europe/Amsterdam
```

   Environment variables must be established for each running terminal session. An utility is provided in `$PROJECT_ROOT/.scripts/source-env.sh`, use it as follows from `$PROJECT_ROOT`

```
$ source .scripts/source-env.sh <path-to-env-file>
```

   The default .env file is located at `$PROJECT_ROOT/compose/defaults.env`.

   Please note that you have to source your (or the default) .env file for each session! In each session use the config doctor from section 5a to ensure the system is configured correctly.

   NOTE: To streamline configuration you can edit `$PROJECT_ROOT/compose/defaults.env` before distributing the project to your machines, the variables in this file will be loaded automatically when you start a pos-sim-core docker by following instruction in section 2. If you then still need to change them at runtime you will have to soruce the file from inside the container again as explained above.

(a) Config doctor

Another utility is provided at which will validate the configuration and verify that the coordination services are reachable.

It requires no arguments as it reads the environment vars.

You can invoke the `config-doctor` by running:

```
$ python -m sim.executor.config-doctor
```

6. Monitoring the distributed system

   The status of the distributed system can be monitored with a web-ui provided by `flower` (`https://flower.readthedocs.io/en/latest/`).

   Start a new session on one of the machines running the project's docker images (not the coordinator!)

   ```
   $ docker exec -it pos-sim-core /bin/zsh
   ```

   once the session starts run:

   ```
   $ launch-flower
   ```

   If no active worker is found this may log some warnings like: `'stats' inspect method failed`, don't worry, as soon as a worker connects the system will heal automatically.

   The docker exposes port 5555 so you can open a browser on that machine (outside of docker that is) and point it to `http://0.0.0.0:5555`

   Note that the graphs are not retroactive so keep a tab open on the graph page and do not reload.

Results analysis

Experiment results are analyzed with R code. Compiling R dependencies may take a lot of time (nearly 30 minutes for the dependencies of `analyze.Rmd` on an medium tier laptop), and errors in the process may harm the reproducibility of the analysis. Therefore a third docker image is provided which ships with all the dependencies compiled in it, and when run exposes an R-studio web interface to run (and possibly customize) the analysis.

Whether you produced results via single process code, or via the distributed system you will have one or more csv files with results to analyze.

If running via the distributed system use `dctl` utility (section 4) to retrieve form the database with the desired csv.

The analysis can be performed by the R script provided in `$PROJECT_ROOT/pos-sim-r/analyze.Rmd`.

You should run the analysis docker on the machine where you downloaded the results via `dctl` or transfer the csv files to another machine and then use that one.

Copy the results csv file to `$PROJECT_ROOT/pos-sim-r/data/exp_data/`.

NOTE: `$PROJECT_ROOT/pos-sim-r/data` is a docker volume that will be mounted when the image is run, so you can copy from your host system with `cp` or drag/drop and the changes will be reflected inside the container.

NOTE: The analysis script will merge all files that it finds in the `exp_data` directory so be careful to only have the files you desire in there later when you run the script. If you create other data folders you can control which is used by ediding the first cell of `analyze.Rmd` where **data.folder** is defined.

Download the image from `https://zenodo.org/record/4543831/files/pos-sim-r-latest.tar.gz?download=1` (to check the sha sums see section .)

Load the image to the docker engine:

```
$ docker load < pos-sim-r-latest.tar.gz
```

Enter `$PROJECT_ROOT/pos-sim-r`. Edit the **defaults.env** file to change the default password ('foobarbaz') for the R-studio server. (The username is always 'rstudio'). Now run

```
$ docker-compose up
```

As usual this will hang until you stop it with CTRL-C

Point a browser to R-studio web-ui on `http://localhost:8787`. It will ask to login with password you provided in **defaults.env**.

enter the `projects` folder and Open the analyze.Rmd file, press 'knit'.

Alternatively run from the rstudio terminal: `$ make render`.

Assuming valid data is found in `projects/exp_data` within the docker, the analyze.Rmd will produce a pdf/markdown/html (depending on kint options, defaults to html) file that presents all the results. The file will be saved in the docker `projects/` folder and is therefore also present on your host machine in `$PROJECT_ROOT/pos-sim-r/data`.

The main experiment discussed in the project's paper is at: `http://139.162.161.39/thesis/analyze.html`
Yours will be available locally: at `http://localhost:8787/files/projects/analyze.html`

Project structure

PROJECT_ROOT

```
.
|-- compose
|   |-- data
|   |   `-- .gitgignore
|   |-- defaults.env
|   |-- docker-compose.yml
|   |-- vars.env
|   `-- vars.wan.env
|-- Dockerfile
|-- .dockerignore
|-- doc_source
|   |-- conf.py
|   |-- index.rst
|   `-- _static
|-- .gitignore
|-- Makefile
|-- pipinstalls.txt
|-- pos-sim-r
|   |-- data
|   |   |-- analyze.html
|   |   |-- analyze.Rmd
|   |   |-- exp_data
|   |   |   `-- .gitkeep
|   |   |-- figure
|   |   |   |-- .gitkeep
|   |   |   `-- score_contour.jpg
|   |   |-- makefile
|   |   `-- make.r
|   |-- defaults.env
|   |-- docker.build.sh
|   |-- docker-compose.yml
|   |-- Dockerfile
|   `-- install_deps.r
|-- README.html
|-- README.md
|-- README.org
|-- README.pdf
|-- .scripts
|   |-- add-aur.sh
|   |-- dctl
|   |-- get-batch-file
|   |-- install.sh
|   |-- launch-flower
|   |-- list-batch-files
|   |-- revoke-sudo.sh
|   |-- run-worker
|   |-- sim-launcher
|   |-- sim-stake1
|   |-- source-env.sh
|   |-- tabulate.sh
|   `-- welcome.sh
```

```
|-- setup.cfg
|-- setup.py
|-- sim
|   |-- cmd
|   |   `-- ucmd.py
|   |-- core
|   |   |-- abstract_sim.py
|   |   |-- base_object.py
|   |   |-- boot_exp.py
|   |   |-- decorators.py
|   |   |-- ecdf.py
|   |   |-- implem.py
|   |   |-- __init__.py
|   |   |-- main.py
|   |   |-- node.py
|   |   |-- parser.py
|   |   |-- plot.py
|   |   |-- rew_f.py
|   |   |-- sel_f.py
|   |   |-- sim_0.py
|   |   |-- stake_f.py
|   |   `-- utils.py
|   |-- executor
|   |   |-- batch
|   |   |   `-- ibatch.py
|   |   |-- celeryconf.py
|   |   |-- config-doctor.py
|   |   |-- db
|   |   |   |-- cmd.py
|   |   |   |-- fs.py
|   |   |   |-- logger.py
|   |   |   `-- parser.py
|   |   |-- dbdriver.py
|   |   |-- experiments
|   |   |   |-- exp_01.py
|   |   |   |-- exp_0.py
|   |   |   |-- exp_365.py
|   |   |   |-- exp_const_geom_pt2.py
|   |   |   |-- exp_const_geom.py
|   |   |   |-- exp_log.py
|   |   |   `-- foo.py
|   |   |-- launcher.py
|   |   |-- logger.py
|   |   `-- tasks.py
|   `-- parser
|       `-- aparse.py
|-- sim-coordinator
|   |-- docker-compose.yml
|   |-- mongo-volume
|   |   |-- .gitignore
|   |   `-- README.txt
|   `-- redis-data
|       |-- .gitignore
|       `-- README.txt
`-- todo.org

20 directories, 86 files
```

1. Locs

Language	Files	Lines	Blank	Comment	Code
Python	41	3121	776	316	2029
./executor/dbdriver.py		381	96	23	262
./executor/tasks.py		168	44	12	112
./core/boot_exp.py		144	32	3	109
/core/test/stake-sim-0.py		159	35	15	109
executor/config-doctor.py		137	33	0	104
./executor/launcher.py		195	55	39	101
./core/sim_0.py		144	35	11	98
/executor/batch/ibatch.py		145	41	10	94
./core/abstract_sim.py		123	29	14	80
./parser/aparse.py		113	23	13	77
./core/plot.py		88	16	4	68
./core/decorators.py		78	12	1	65
./core/implem.py		94	23	10	61
./executor/db/fs.py		94	30	9	55
./core/utils.py		64	12	1	51
./core/node.py		62	13	3	46
./core/main.py		67	18	5	44
./core/base_object.py		66	16	9	41
./cmd/ucmd.py		54	11	3	40
./core/parser.py		54	14	1	39
./core/test/random1.py		46	15	0	31
./core/sel_f.py		42	8	6	28
./core/stake_f.py		50	11	12	27
./executor/celeryconf.py		37	11	0	26
./executor/db/parser.py		32	7	0	25
./executor/db/cmd.py		43	15	6	22
./core/test/batch.py		63	16	25	22
./executor/test/ctx.py		39	16	2	21
nts/exp_const_geom_pt2.py		36	10	7	19
riments/exp_const_geom.py		39	11	9	19
./core/ecdf.py		32	3	10	19
or/experiments/exp_365.py		33	9	7	17
utor/experiments/exp_0.py		34	10	7	17
tor/experiments/exp_01.py		34	10	7	17
or/experiments/exp_log.py		39	11	11	17
./core/rew_f.py		27	9	2	16
./core/test/tx.py		21	6	1	14
./core/__init__.py		10	3	0	7
./executor/db/logger.py		15	3	8	4
./executor/logger.py		17	3	10	4
ecutor/experiments/foo.py		2	1	0	1

Images sha256sum

from the folder where you downloaded the archives run `$ sha256sum *.tar.gz`

```
dbe608295d63480a21421f24b7582dea8f70613b383d783e46b0d7683e675ca  pos-sim-core-latest.tar.gz
d9d564c2b26b3df8d105235d0ae4f2fe98d45d52620021b5a7d08617b730cd78 pos-sim-r-latest.tar.gz
```