Trader Contagion: Agent-based Stochastic Model of Markets

Name: Tsigemariam Assegid (UNI: tya2104)

November 27, 2023

Overview

In this project, I implemented a parallel version of the agent based and stochastic model described in "Linking agent-based models and stochastic models of financial markets". The paper focuses on linking agent-based and stochastic models to understand financial market dynamics. The paper investigates the emergence of fat tails and long-term memory in financial returns, suggesting that these characteristics can be attributed to the collective behavior of market participants. It emphasizes the importance of agent heterogeneity and the interaction between different types of traders. The research demonstrates how agent-based models can provide valuable insights into complex market phenomena and supports the idea that market dynamics are deeply rooted in the actions and strategies of individual traders.

Implementation

The agent based model simulates the actions of individual agents in a financial market, incorporating randomness(noise) to reflect real-world unpredictability.

The model calculates the probability of trading based on market velocity (V), differentiating between fundamental (V_f) and technical traders (V_c). Fundamental traders are assumed to hold a majority of the shares 83%$, basedonhistoricaldatafrom1997 - 2006$. Trading probability is derived from the velocities, with multiple choices for V_f, including the best-fit value of 0.4 used in the paper.

Agents decide whether to buy, sell, or hold based on the calculated trading probability. They are also distributed into opinion groups, with the number of groups determined by ω. The model sets a logical minimum of one opinion group (where all agents share the same opinion) and a maximum equal to the number of agents. The diversity of opinions affects market dynamics, with a higher number of groups reducing herd behavior. At each timestep, the model updates based on agents' decisions and market changes. This includes recalculating trading probabilities and adjusting agent behaviors according to new market conditions. The boundaries on returns is set according to the guidelines from Feng et al. 2012's Appendix 5.

In my implementation, I mainly focused on testing sensitivity of the model return's on the number of opinion groups to []. I simulated 10 runs for each ω (11 different ω values) listed on the paper. In each run, I used the following parameters:

· number of agents (n) : 1024

· probability of trading (p) : 0.2178

· steps: 1000

For each value of ω, I collected key statistics: daily returns, daily trading volume, total trading volume. Based on the paper, I implemented hill estimator and linear regression model to understand the relationship between the returns and number of opinion group.

Hill estimator is used in the paper to primarily to assess the tail heaviness of a distribution. The Hill estimator provides a measure of the "tail thickness" of the distribution, with higher values indicating a "heavier" tail, which implies a higher risk of extreme price movements. Hill estimators are used in financial modeling to to evaluate the risk of extreme price movements. I implemented linear regression to to model the relationship between the omega parameter (representing the number of opinion groups) and the Hill estimator values of returns (representing market extremities) derived from market simulations. The linear regression model is fitted to these values and calculates and returns the slope, intercept, the coefficient of determination (R^2), and p-value, which indicate how much of the variability in the Hill estimator can be explained by omega.

After the simulations and analysis, the calculated p-value (0.00037123621) is less than 0.05 rejecting the null hypothesis and showing a significant relationship between the variables. A positive correlation is also observed between omega and the Hill exponent as shown in the paper. Higher omega values which means higher number of opinion groups therefore decreased probability of herd effect correlate with a steeper slope of the distribution. For instance, if all market participants converge into a single opinion group and consequently execute identical trading actions, it would lead to high fluctuation in return, reflecting extreme market movements

I also implemented the stochastic model detailed on the paper. It involved allocating agents across different time horizons, informed by their trading strategies and market behaviors. This model captures the randomness inherent in financial markets. Agents are distributed based on an exponential decay function, which accounts for the diminishing influence of past market events over time.

Parallel Implementation

I parallelized the simulations and analysis related to different omega values described in the section above. The sequential version took over 60s, I was able to get to around 6s in the parallel version.

Here are the threadscope results:

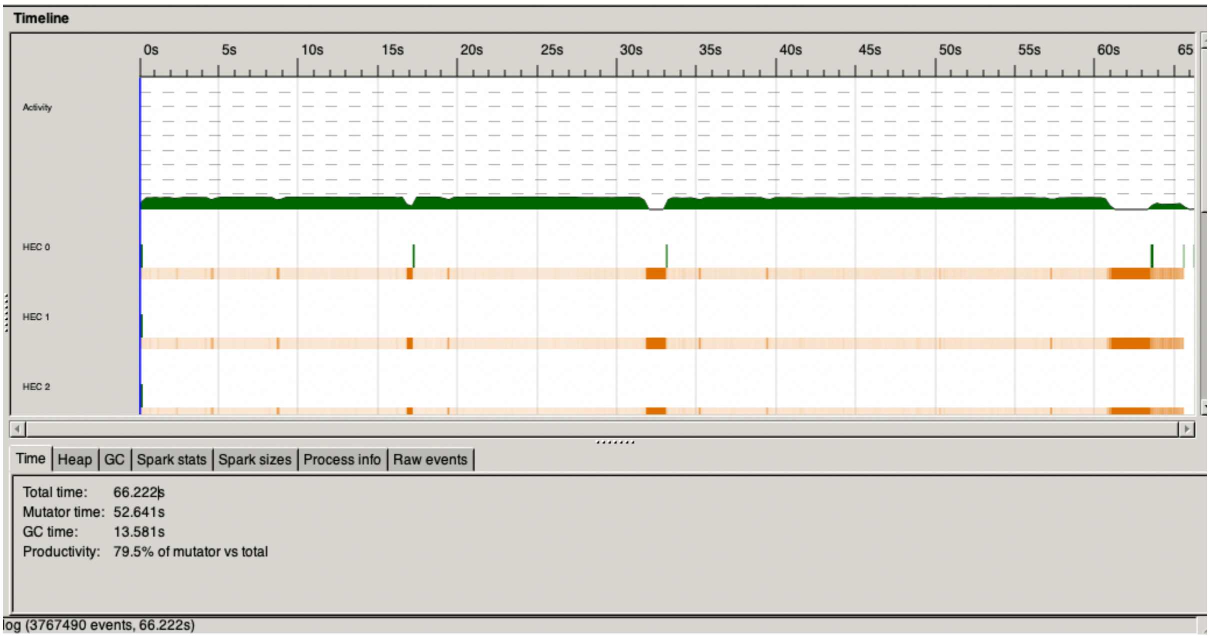

Figure 1: Sequential Simulation

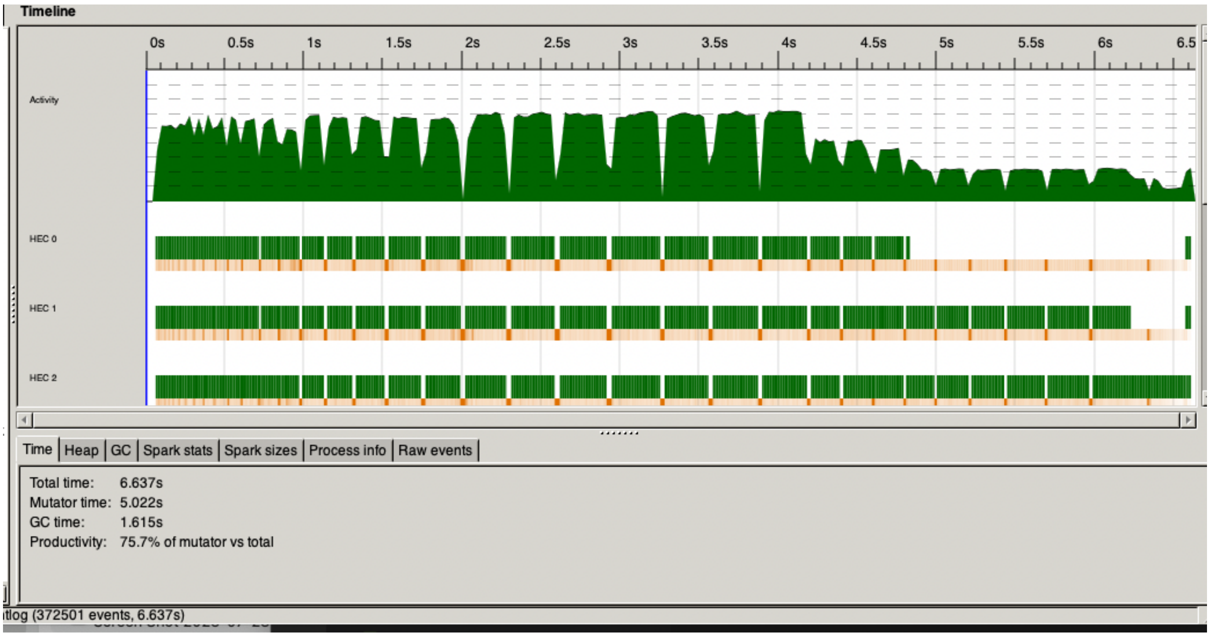

Figure 2: Parallel Simulation

Code Listing

Main.hs

```haskell
module Main (main) where

import Lib
import DifferentOmega
import ParallelDifferentOmega
import LinearRegression
main :: IO ()
main = diffomega
```

AgentBased.hs

```haskell
module AgentBased
    ( Model
    , prunModel
    , initializeModel
    , step              -- Function to perform one step of the model
    , dailyReturns      -- Function to get daily returns from the model
    , dailyTradingVolumes -- Function to get daily trading volumes
    , runModel
    ) where

import System.Random
    ( newStdGen, randomRIO, uniformR, Random(randomR), RandomGen )
```

```haskell
import System.Random.MWC ( create )
import System.Random.MWC.Distributions (normal)
import Control.Monad ( replicateM, replicateM_ )
import Control.Monad.State
    ( MonadState(put, state, get),
      MonadIO(liftIO),
      execStateT,
      runState,
      StateT )
import Statistics.Sample (mean, stdDev)
import Data.Vector (fromList)
import Graphics.Gnuplot.Simple ( plotList )
import Graphics.Gnuplot.Advanced ()
import Debug.Trace ()

-- Model data type
data Model = Model {
    n :: Integer,
    p :: Double,
    dailyReturn :: Double,
    tradingVolume :: Int,
    k :: Int,
    omega :: Double,
    dailyReturns :: [Double],
    ct :: Int,
    b :: Int,
    dailyTradingVolumes :: [Int]
} deriving (Show)

boxMuller :: (Double, Double) -> (Double, Double)
boxMuller (u1, u2) = (z0, z1)
  where
    r = sqrt (-2 * log u1)
    theta = 2 * pi * u2
    z0 = r * cos theta
    z1 = r * sin theta

-- Generate a normally distributed number
generateNormal :: RandomGen g => Double -> Double -> g -> (Double, g)
generateNormal mean stddev gen =
  let scale = sqrt stddev
      (u1, gen1) = randomR (0, 1) gen
      (u2, gen2) = randomR (0, 1) gen1
      (z0, _) = boxMuller (u1, u2)
  in (mean + z0 * scale, gen2)

```

```haskell
-- Pure version of buySellHold with explicit random number generator state
buySellHoldPure :: RandomGen g => Double -> Int -> g -> ([Int], g)
buySellHoldPure p amountTimes gen =
    let (diceRolls, gen1) = generateDiceRolls amountTimes gen
        (coinFlips, gen2) = generateCoinFlips amountTimes gen1
        indices = filter ((<= (2 * p)) . snd) $ zip [0..] diceRolls
        psis = zipWith (\(idx, _) coin -> (idx, if coin == 0 then 1 else -1)) indices coinFlips
        result = foldr (\(idx, val) acc -> take idx acc ++ [val] ++ drop (idx + 1) acc) (replicate amountTimes 0) psis
    in (result, gen2)

-- Helper function to generate a list of dice rolls
generateDiceRolls :: RandomGen g => Int -> g -> ([Double], g)
generateDiceRolls n = runState $ replicateM n (state $ uniformR (0.0, 1.0))

-- Helper function to generate a list of coin flips
generateCoinFlips :: RandomGen g => Int -> g -> ([Int], g)
generateCoinFlips n = runState $ replicateM n (state $ randomR (0, 1))

-- buy_sell_hold function
buySellHold :: Double -> Int -> IO [Int]
buySellHold p amountTimes = do
    diceRolls <- replicateM amountTimes (randomRIO (0.0, 1.0))
    let indices = filter ((<= (2 * p)) . snd) $ zip [0..] diceRolls
    psis <- mapM (\(idx, _) -> do
                    coin <- randomRIO (0, 1 :: Int)
                    return (idx, if coin == 0 then 1 else -1)
                 ) indices
    return $ foldr (\(idx, val) acc -> take idx acc ++ [val] ++ drop (idx + 1) acc) (replicate amountTimes 0) psis



mean' :: Model -> Double
mean' model = (fromIntegral (n model) / abs (dailyReturn model)) ** (omega model)

pdistributeOpinionGroups :: RandomGen g => Model -> g -> (Int, g)
pdistributeOpinionGroups model gen
    | b model == 0 = (round $ mean' model, gen)
    | abs (dailyReturn model) >= fromIntegral (n model) = (1,gen)
    | otherwise =
        let mean = mean' model
```

```
            bVal = b model
            (c, newGen) = generateNormal mean (fromIntegral bVal) gen
            d = max 1 (round c)
        in (min d (fromIntegral (n model)), newGen)


distributeOpinionGroups :: Model -> IO Int
distributeOpinionGroups model
    | b model == 0 = return $ round $ mean' model
    | abs (dailyReturn model) >= fromIntegral (n model) = return 1
    | otherwise = do
        let mean = mean' model
            stdDev = sqrt (mean * fromIntegral (b model))
            minValue = mean - stdDev
            maxValue = mean + stdDev
        g <- create
        c <- normal mean stdDev g
        -- liftIO $ putStrLn $ "c: " ++ show c ++ show mean ++ show stdDev
        let d = max 1 (round c)
        return $ min d (fromIntegral (n model))

applyBoundaries :: Double -> Double -> Double -> Double
applyBoundaries dailyReturn minReturn maxReturn =
    let sign = if dailyReturn < 0 then -1 else 1
    in sign * min maxReturn (max minReturn (abs dailyReturn))

pstep :: RandomGen g => Model ->  g -> (Model, Int, g)
pstep model gen =
    let (c, gen1) = pdistributeOpinionGroups model gen
        (psis, gen2) = buySellHoldPure (p model) c gen1
        averageAgentsPerGroup = fromIntegral (n model) / fromIntegral c
        returnMatrix = map ((* averageAgentsPerGroup) . fromIntegral) psis
        -- Other calculations
        tradingVolume = round $ sum $ map abs returnMatrix
        dailyReturn' = sum returnMatrix
        minimumReturn = fromIntegral (n model) ** ((omega model - 1) / omega model)
        dailyReturn'' = applyBoundaries dailyReturn' minimumReturn (fromIntegral (n model))
        newModel = model { dailyReturn = dailyReturn'',
                           dailyReturns = dailyReturns model ++ [dailyReturn''],
                           dailyTradingVolumes = dailyTradingVolumes model ++ [tradingVolume],
```

```haskell
                                ct = ct model + 1 }
    in (newModel, ct model + 1, gen2)

step :: StateT Model IO Int
step = do
    model <- get
    c <- liftIO $ distributeOpinionGroups model
    psis <- liftIO $ buySellHold (p model) c
    let averageAgentsPerGroup =  fromIntegral (n model) / fromIntegral c
        returnMatrix = map ((* averageAgentsPerGroup) . fromIntegral) psis
    -- liftIO $ putStrLn $ "c: " ++ show c ++ ", avgAgentsPerGroup: " ++ show averageAgentsPerGroup ++ ", returnMatrix: " ++ show returnMatrix
    let
        tradingVolume = round $ sum $ map abs returnMatrix
        dailyReturn' = sum returnMatrix  -- Should be Double now
        minimumReturn = fromIntegral (n model) ** ((omega model - 1) / omega model)
        dailyReturn'' = applyBoundaries dailyReturn' minimumReturn (fromIntegral (n model))
    put model { dailyReturn = dailyReturn'',
                dailyReturns = dailyReturns model ++ [dailyReturn''],
                dailyTradingVolumes = dailyTradingVolumes model ++ [tradingVolume],
                ct = ct model + 1 }
    return $ ct model + 1

prunModel :: RandomGen g => Int -> Model ->  g -> (Model, g)
prunModel 0 model gen = (model, gen)
prunModel t model gen =
    let (updatedModel, _, newgen) = pstep model gen
    in prunModel (t - 1) updatedModel newgen

runModel :: Int -> Model -> IO Model
runModel t model = execStateT (replicateM_ t step) model

standardScale :: [Double] -> [Double]
standardScale xs = map (\x -> (x - m) / s) absXs
  where
    absXs = map abs xs  -- Take the absolute value of each element
    vXs = fromList absXs  -- Convert the list to a Vector
    m = mean vXs  -- Calculate the mean
    s = stdDev vXs  -- Calculate the standard deviation
```

```haskell
initializeModel :: Integer -> Double -> Double -> Int -> Int -> Model
initializeModel nVal pVal omegaVal bVal kVal = Model {
        n = nVal,
        p = pVal,
        dailyReturn = 1.0,
        dailyReturns = [],
        dailyTradingVolumes = [],
        omega = omegaVal,
        b = bVal,
        k = kVal,
        tradingVolume = 0,
        ct = 0
        }

main :: IO ()
main = do

    let initialmodel = Model {n = 1024,  p = 0.02178, dailyReturn =
    1.0, dailyReturns = [], dailyTradingVolumes = [], omega = 1, b = 1,
    k = 1, tradingVolume = 0, ct = 0}
    gen <- newStdGen
    finalmodel1 <- runModel 20 initialmodel
    let (finalmodel, _) = prunModel 10000 initialmodel gen
        y = standardScale (dailyReturns finalmodel)
        y2 = standardScale (dailyReturns finalmodel1)
        points = zip ([1..] :: [Int]) y
    plotList [] points
```

ABMSimulations.hs

```haskell
module ABMSimulation
    (
        runABM,
        prunABM
    ) where
import AgentBased
    ( Model(dailyTradingVolumes, dailyReturns),
      prunModel,
      runModel,
      initializeModel )
import Control.Monad (replicateM)
import System.Random (StdGen)

-- Function to run the ABM model for a given number of runs and time
    steps
runABM :: Integer -> Double -> Double -> Int -> Int -> Int -> Int -> IO
    ([[Double]], [[Int]])
```

```haskell
runABM n p omega b k t runs = do
    results <- replicateM runs $ do
        let model = initializeModel n p omega b k  -- Initialize the model
        finalModel <- runModel t model                     -- Run the model for t steps
        let returns = dailyReturns finalModel
        let volumes = dailyTradingVolumes finalModel
        return (returns, volumes)
    let (returns, volumes) = unzip results
    return (returns, volumes)


prunABM :: Integer -> Double -> Double -> Int -> Int -> Int -> Int -> StdGen -> ([[Double]], [[Int]])
prunABM n p omega b k t runs gen =
    let results = replicate runs $
            let model = initializeModel n p omega b k  -- Initialize the model
                (finalModel, newGen) = prunModel t model gen  -- Run the model for t steps
                returns = dailyReturns finalModel
                volumes = dailyTradingVolumes finalModel
            in (returns, volumes)
    in unzip results

-- Function to calculate probability of trading based on the market velocity of fundamental and chartist traders
probabilityOfTrading :: Double -> Double -> Double
probabilityOfTrading vf v = vc / (250 * 2)
  where
    vc = (v - 0.83 * vf) / (1 - 0.83)
```

Sequential version of the different omega simulations

```haskell
module DifferentOmega (diffomega) where
import ABMSimulation ( runABM )
import Control.Monad
import Data.List
import HillEstimator
import qualified Data.Map as Map
import LinearRegression

diffomega :: IO ()
diffomega = do
    let omega_list = [0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.4, 1.5, 2.0]
```

```
    results <- forM omega_list $ \omega -> do
        runABM 1024 0.02178 omega 1 1 1000 10
    let (normalized_return, normalized_voulme) = processResults results
        hill_estimator_returns = applyHillEstimator 1 normalized_return
        mean_return = init $ meanReturns hill_estimator_returns
        (slope, intercept, r2, tStats, pVal) = regAnalysis omega_list
  mean_return
    print(slope, intercept,r2, pVal)
    return()

applyHillEstimator:: Double -> [[[Double]]] -> [[[Double]]]
applyHillEstimator t d = map (map (\x -> [hillEstimator t x])) d

normalise :: [Double] -> [Double]
normalise xs = map (\x -> abs (x - mean) / stdDev) xs
  where
    mean = sum xs / fromIntegral (length xs)
    stdDev = sqrt $ sum (map (\x -> (x - mean) ** 2) xs) / fromIntegral
    (length xs)

processResults :: [([[Double]], [[Int]])] -> ([[[Double]]], [[[Double
  ]]])
processResults results = (absNormalizedReturns, abmNormalisedVolumes)
  where
    absNormalizedReturns = map (map normalise . fst) results
    abmNormalisedVolumes = map (map (normalise . map fromIntegral) .
  snd) results

meanReturns :: [[[Double]]] -> [Double]
meanReturns = map (mean . concat)
    where
    mean xs = sum xs / fromIntegral (length xs)
```

Parallel Version

```
module ParallelDifferentOmega (pdiffomega) where
import Control.Monad (replicateM)
import System.Random (newStdGen)
import Control.Parallel.Strategies
    ( runEval, parList, parMap, rdeepseq, using )
import ABMSimulation ( prunABM )
import Control.Parallel ()
import HillEstimator ( hillEstimator )
import LinearRegression ( regAnalysis )

pdiffomega :: IO()
pdiffomega = do
```

```haskell
    let omega_list = [0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.4, 1.5, 2.0]
    gens <- replicateM (length omega_list) newStdGen  -- Generate a list of random number generators
    let results = zipWith (\omega gen -> prunABM 1024 0.02178 omega 1 1 10000 10 gen) omega_list gens
                            `using` parList rdeepseq
    let (normalized_return, normalized_voulme) = processResults results
        hill_estimator_returns = applyHillEstimator 1 normalized_return
        mean_return = init $ meanReturns hill_estimator_returns
        (slope, intercept, r2, tStats, pVal) = regAnalysis omega_list mean_return
    print(slope, intercept,r2, pVal)
    return()

applyHillEstimator :: Double -> [[[Double]]] -> [[[Double]]]
applyHillEstimator t = map (parMap rdeepseq (\x -> [hillEstimator t x]))


normalise :: [Double] -> [Double]
normalise xs = map (\x -> abs (x - mean) / stdDev) xs
  where
    mean = sum xs / fromIntegral (length xs)
    stdDev = sqrt $ sum (map (\x -> (x - mean) ** 2) xs) / fromIntegral (length xs)

processResults :: [([[Double]], [[Int]])] -> ([[[Double]]], [[[Double]]])
processResults results = runEval $ do
    absNormalizedReturns <- rdeepseq (map (map normalise . fst) results)
    abmNormalisedVolumes <- rdeepseq (map (map (normalise . map fromIntegral) . snd) results)
    return (absNormalizedReturns, abmNormalisedVolumes)

meanReturns :: [[[Double]]] -> [Double]
meanReturns = map (mean . concat)
    where
    mean xs = sum xs / fromIntegral (length xs)
```

Linear Regression Model

```haskell
{-# OPTIONS_GHC -Wno-identities #-}
module LinearRegression(regAnalysis)where
import Statistics.LinearRegression ( linearRegressionRSqr )
import Numeric.LinearAlgebra
```

```haskell
    ( Transposable(tr),
      fromList,
      (><),
      inv,
      (<>),
      toList,
      takeDiag,
      Linear(scale) )
import Statistics.Distribution ( Distribution(complCumulative) )
import Statistics.Distribution.StudentT ( studentT )

-- Fit the linear model  and calculate statistical measures
regAnalysis :: [Double] -> [Double] -> (Double, Double, Double, [Double
    ], [Double])
regAnalysis omega returns = (slope, intercept, r2, tStats, pVals)
  where
    xVec = fromList omega
    yVec = fromList returns
    (intercept, slope, r2) = linearRegressionRSqr xVec yVec
    predictions = map (predict (intercept, slope)) omega
    sse = sum $ zipWith (\x y -> (x - y) ** 2) predictions returns
    sampleSize = length omega
    numPredictors = 1.0
    mse = sse / (fromIntegral sampleSize - numPredictors - 1.0)
    ones = replicate (length omega) 1
    xMatrix = (length omega >< 2) (ones ++ omega)
    covarianceMatrix = scale mse $ inv (tr xMatrix Numeric.
    LinearAlgebra.<> xMatrix)
    se = toList $ sqrt $ takeDiag covarianceMatrix
    tStats = [slope / head se]
    pVals = map (\t -> 2 * complCumulative (studentT (fromIntegral
    sampleSize - numPredictors - 1)) (abs t)) tStats

predict :: (Double, Double) -> Double -> Double
predict (intercept, slope) x = intercept + slope * x
```

Hill Estimator

```haskell
module HillEstimator(hillEstimator)where
import Numeric.LinearAlgebra ()
import Data.List ( sort )

hillEstimator :: Double -> [Double] -> Double
hillEstimator tailPercentage dataList = alphaEst
  where
    sortedData = sort dataList
    n = fromIntegral $ length sortedData
```

```
    k = round $ (tailPercentage * n) / 100
    logXNMinusK = log $ sortedData !! (round n - k - 1)
    logXNMinusJPlus1 = map log $ take k $ reverse sortedData
    alphaEst = fromIntegral k / sum (map (\x -> x - logXNMinusK)
   logXNMinusJPlus1)

normalise :: [Double] -> [Double]
normalise array = normalized
  where
    mean = sum array / fromIntegral (length array)
    stdDev = sqrt $ sum (map (\x -> (x - mean) ** 2) array) /
   fromIntegral (length array)
    normalized = map (\x -> abs (x - mean) / stdDev) array
```

Stochastic Model

```
module Stochastic
    (
        StochasticModel(..)
        , runModel
        , initializeStochasticModel
    ) where


import System.Random.MWC
import System.Random.MWC.Distributions (normal)
data StochasticModel = StochasticModel {
    n :: Integer,
    p :: Double,
    initial :: Double,
    returns :: [Double],
    time_horizon :: Bool,
    d :: Double,
    m :: Int
} deriving (Show)

initializeStochasticModel :: Integer -> Double -> Double -> Bool ->
   Double -> Int -> StochasticModel
initializeStochasticModel nVal pVal initialVal timeHorizonVal dVal mVal
    = StochasticModel {
    n = nVal,
    p = pVal,
    initial = initialVal,
    returns = [initialVal],
    time_horizon = timeHorizonVal,
    d = dVal,
    m = mVal
```

```
}

-- Function to calculate time horizons
timeHorizons :: StochasticModel -> Double
timeHorizons model = sum timeHorizonsList / sum alphaList
  where
    returnsList = returns model
    mValue = m model
    dValue = d model
    timeHorizonsList = [ fromIntegral i ** (-dValue) * absReturn i | i <- [1..mValue] ]
    alphaList = [ fromIntegral i ** (-dValue) | i <- [1..mValue] ]
    absReturn i
      | length returnsList == 1 = abs (head returnsList)
      | i >= length returnsList = abs (head returnsList - last returnsList)
      | otherwise = abs (last returnsList - (returnsList !! (length returnsList - i)))

-- Function to perform a step
step :: StochasticModel -> IO StochasticModel
step model = do
    g <- createSystemRandom
    normalVal <- normal 0.0 1.0 g
    -- liftIO $ putStrLn $ show normalVal
    let variance = if time_horizon model
                      then 2 * p model * fromIntegral (n model) * timeHorizons model
                      else 2 * p model * fromIntegral (n model) *  abs ( last (returns model))
    let std = sqrt variance
    let value = std * normalVal
    let newReturns = returns model ++ [value]
    return model { returns = newReturns }

runModel :: (Eq t, Num t) => t -> StochasticModel -> IO StochasticModel
runModel = iterateM
  where
    iterateM 0 m = return m
    iterateM n m = step m >>= \newModel -> iterateM (n-1) newModel
```

Stochastic Simulations

```
module StochasticSimulation
(

) where
```

```haskell
import Stochastic
    ( StochasticModel(returns), initializeStochasticModel, runModel )
import Control.Monad (replicateM, forM_)

-- Function to run the stochastic model for a given number of runs and time steps
runStochasticModel :: Integer -> Double -> Double -> Bool -> Double -> Int -> Int -> Int -> IO [[Double]]
runStochasticModel n p init timeHorizon d m t runs = do
    results <- replicateM runs $ do
        let model = initializeStochasticModel n p init timeHorizon d m
        finalModel <- runModel t model
        return (returns finalModel)
    return results
```

References

1. Feng, L., Li, B., Podobnik, B., Preis, T., Stanley, H. E. (2012). Linking agent-based models and stochastic models of financial markets. Proceedings of the National Academy of Sciences of the United States of America, 109(22), 8388–8393. http://www.jstor.org/stable/41602564

2. Hill, B.M. (1975) A simple general approach to inference about the tail of a distribution. Annals of Statistics. 13, 331-341