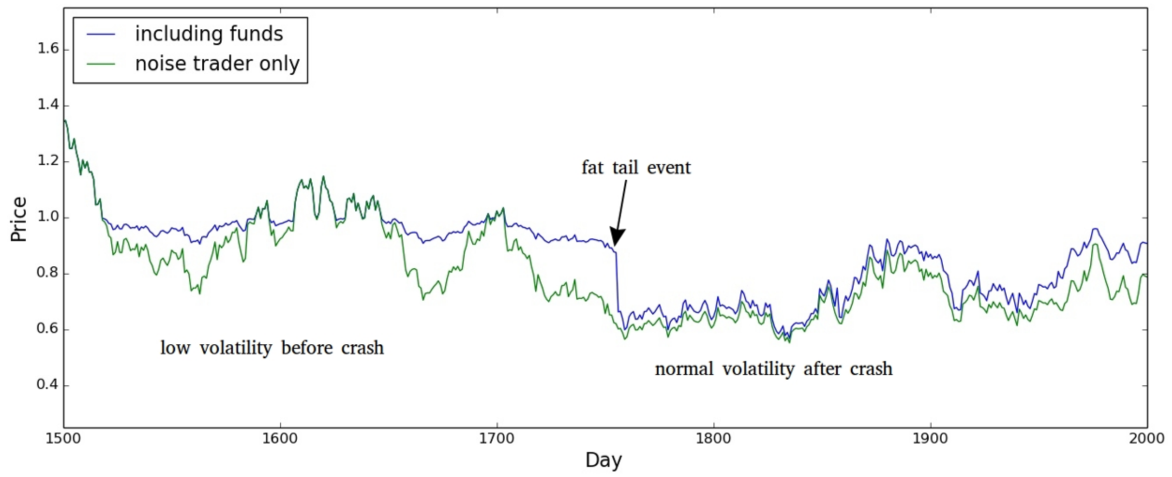
Luzius Meisser, Marlon Azonivic, November 2016
Mathematical Finance Course, Prof. Marc Chesney

Leverage Causes Fat Tails and Clustered Volatility

1. Introduction

This is an implementation of the leveraged trading model described in the 2013 paper *Leverage Causes Fat Tails and Clustered Volatility* (http://www.uvm.edu/~cmplxsys/legacy/newsevents/pdfs/2013/thurner-volatility-finance.pdf) by Thurner, Farmer and Geanakoplos.

The model consists of three agent types. First, there is a noise trader that buys and sells stocks every day without much of a plan. On its own, it generates a mean-reverting random walk like the green one shown in the chart below.



Second, there are a number of leveraged funds following a value strategy. They buy stocks when they are below the fundamental value and sell them again when they rise back above it. Their buying prevents prices from falling significantly below the fundamental value, thereby dampening volatility. However, being leveraged, these funds might be forced to sell if prices fall too much, thereby quickly driving prices even lower. This causes the fat tail event shown in the chart above. The funds go bankrupt and the price falls to where it would be without them. After such a crash, it takes some time until the funds recover. These are periods (clusters) with higher volatility than before.

Third, there are background investors that rebalance their investments between the funds. Their main purpose is to withdraw money from the funds in order to prevent them from getting unboundedly rich.

2. The Noise Trader

The behavior of the noise trader is modeled around the *cash value* $\xi_{nt}(t)$ the trader has invested in the stock market at time t. It is specified such that it follows the autoregressive random process described by equation 1. In absence of other traders, it makes the price $p(t)$ follow a mean-reverting random walk.

$$log(\xi_{nt}(t)) = \rho \, log(\xi_{nt}(t-1)) + \sigma\chi(t) + (1-\rho)log(NV) \tag{1}$$

Parameters $\rho = 0.99$ and $\sigma = 0.035$ are constant, with $\chi(t) \sim \mathcal{N}(0, 1)$ being a standard-normal random variable. The *fundamental value* V of each of the $N = 1000$ shares is constant with $V = 1$. Thus, NV represents the total fundamental value of the stock market and the anchor for the mean-reversion. In absence of other traders, the expected logarithmic amount the noise trader has invested in the stock market matches the logarithmic fundamental value:

$$E[log(\xi_{nt})] = log(NV)$$

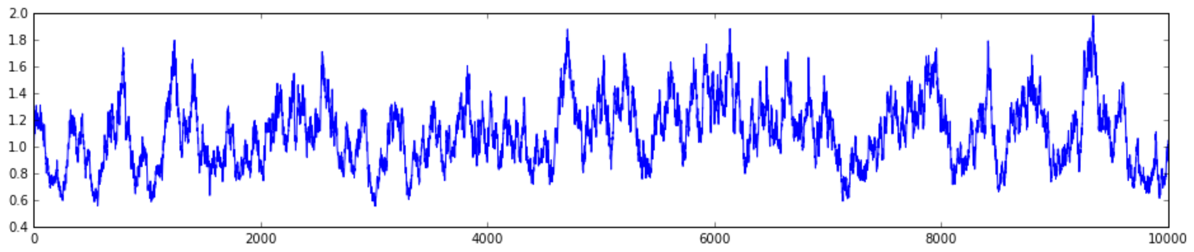
Market clearing further dictates $\xi_{nt} = Np$ and therefore $E[log(p)] = log(V)$ for the price of a single share.

As a first step, we will simulate the noise trader in isolation to get a more intuitive understanding of its behavior and the resulting prices.

In [1]:
```
# Load and configure required modules
%load_ext autoreload
%autoreload 2

import math
import numpy # numerics
import scipy
from scipy import optimize # we use their root finding algorithm
import matplotlib.pyplot as plot
%matplotlib inline
```

In [2]:
```
#Constants
N = 1000 # number of shares
V = 1    # fundamental value per share
RANDOM_SEED = 13 # change to get a different outcome

# Function for noise trader's investment given the previous value (equation 1)
default_rho = 0.99
sigma = 0.035

def calculateNoiseTraderInvestment(previousInvestment):
    return calculateNoiseTraderInvestment2(previousInvestment, default_rho)

def calculateNoiseTraderInvestment2(previousInvestment, rho):
    return math.exp(rho * math.log(previousInvestment) + sigma *
numpy.random.normal() + (1-rho)*math.log(N*V))
```

In [3]:
```
# Create and plot price time series
def createNoiseTraderTimeSeries(length, rho):
    numpy.random.seed(RANDOM_SEED)
    investment = N*V # start value
    prices = [investment / N]
    while len(prices) < length:
        investment = calculateNoiseTraderInvestment2(investment, rho)
        prices.append(investment / N)
    return prices

plot.figure(figsize=(16,3))
noiseResult = createNoiseTraderTimeSeries(10000, default_rho)
plot.plot(noiseResult)
plot.show()
```



3. Investment Funds

3.1 Specification

As a next step, investment funds following a value strategy are added to the model. Each fund h has wealth $W_h(t) = \xi_h(t) + C_h(t)$, consisting of a stock market investment ξ_h and cash C_h. Cash can be negative when the fund is leveraged. All funds base their decisions on the same mispricing signal

$$m(t) = V - p(t) \tag{2}$$

whereas a positive $m(t)$ signals that the stock market is undervalued. As long as the market is overvalued (i.e. $m(t) \leq 0$), the funds do not hold any stocks. However, as soon as an undervaluation is detected, funds demand $\xi_h(t)$ worth of stocks, which depends on the magnitude of the signal as well as the fund's wealth:

$$\xi_h(t) = min\big(\ \beta_h m(t),\ \lambda_{MAX}\big)\ W_h(t) \tag{3}$$

The maximum leverage is set to $\lambda_{MAX} = 20$. This sounds high, but can be justified when considering that European bank leverage currently stands at about 30 (source: Marathon Asset Management (http://www.marathonfund.com/), Global Investment Review, Volume 30 No 1). A deeper investigation on how different values of λ_{MAX} affect the results can be found in the original paper, but is skipped here. The parameter β_h varies between funds and specifies how aggressively a fund responds to the buy signal. There are 10 funds with $5 \leq \beta_h \leq 50$.

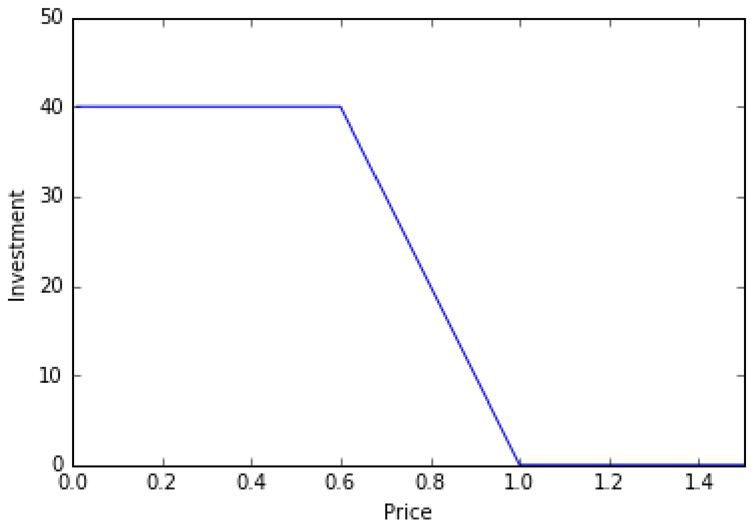
It is key to understand that the investment $\xi_h(t)$ of a fund depends not only of the signal, but indirectly through wealth also on current prices. Once a fund is fully leveraged, a further decline of $p(t)$ can lead to a decreased investment $\xi_h(t)$ despite the signal $m(t)$ getting stronger, simply because the fund does not have the necessary wealth any more.

In [4]:
```
class Fund:

    initialWealth = 2
    maxLeverage = 20  # named lambda_max in the paper

    def __init__(self, aggressiveness):
        self.aggressiveness = aggressiveness
        self.cash = self.initialWealth  # W_h(0) in the equation-based model
        self.shares = 0
        self.activationDelay = 0

    def checkBankrupt(self, price):
        if self.isActive():
            # check for bankrupty
            if self.getWealth(price) <= self.initialWealth * 0.1:
                self.shares = 0
                self.cash = self.initialWealth
                self.activationDelay = 100 # the fund is bankrupt and will res
urrect in 100 days
                return True
            else:
                return False
        else:
            self.activationDelay = self.activationDelay - 1
            return False

    def processInflows(self, oldprice, newprice): # used later
        return

    # returns true if the fund is not bankrupt
    def isActive(self):
        return self.activationDelay == 0;

    def getWealth(self, price):
        return max(0, self.cash + self.shares * price);

    # the number of shares the fund would like to hold at the given price
    def getDemand(self, price):
        if self.isActive():
            m = V - price # equation 2
            if (m < 0):
                return 0

            # equation 3 divided by price
            leverage = min(self.aggressiveness * m, self.maxLeverage)
            return leverage * self.getWealth(price) / price
        else:
            return 0

    # updates the fund's holdings by trading at the given price
    def trade(self, price):
        if self.isActive():
            wealth = self.getWealth(price)
            self.shares = self.getDemand(price)
            self.cash = wealth - self.shares * price
```

3.2 Test

Having defined a fund class, let us test it by reproducing figure 1 from the paper, but with price p on the x-axis instead of signal m and default parameters. The y-axis shows investment ξ_h. The figure illustrates how much a fund with wealth $W_h = 2$, $\beta_h = 50$ and maximum leverage $\lambda_{max} = 20$ would like to invest. As expected $\xi_h \leq W_h \lambda_{max} = 40$ for all prices.

In [5]:
```
testFund = Fund(50)

prices = []
demand = []
for i in range(-99, 100):
    price = V + i / 100
    investment = testFund.getDemand(price) * price
    prices.append(price)
    demand.append(investment)

plot.xlabel('Price')
plot.ylabel('Investment')
plot.axis([0.0, 1.5, 0, 50])
plot.plot(prices, demand)
plot.show()
```



4. Market Clearing

Market clearing dictates that supply must match demand:

$$\xi_{nt}(t) + \sum_h \xi_h(t, p(t)) = Np(t) \tag{4}$$

Total investments must match market capitalization.

In [6]:
```
minPrice = 0.01
maxPrice = 5

# Rearranged equation 4
def calculateExcessDemand(noiseTraderInvestment, funds, price):
    demand = noiseTraderInvestment / price
    for f in funds:
        demand = demand + f.getDemand(price)
    return demand - N

def findEquilibrium(noiseTraderInvestment, funds):
    # The scipy solver wants an univariate function, so we create a temporary
 demand function
    # that only depends on p, with the other two parameters staying constant
    currentExcessDemandFunction = lambda p : calculateExcessDemand(noiseTrader
Investment, funds, p)
    return scipy.optimize.brentq(currentExcessDemandFunction, minPrice, maxPri
ce)
```

5. Running the Simulation

5.1 Noise Trader and Funds

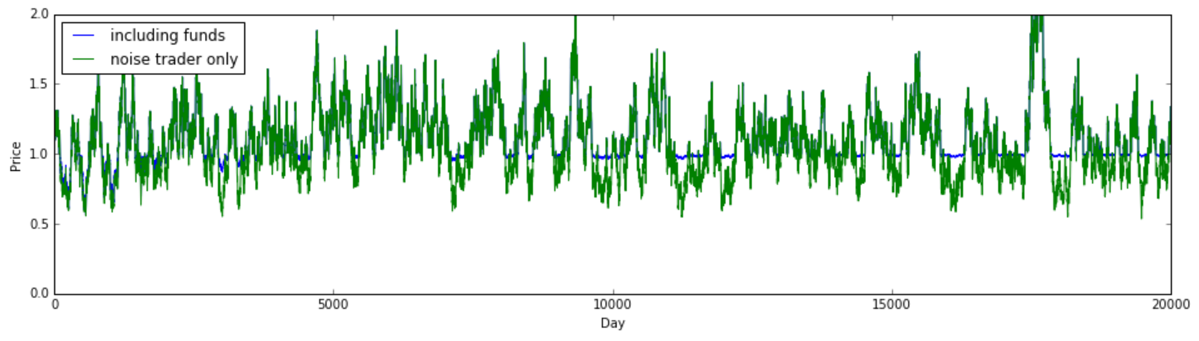
Before moving on to the background investor, let us test what the stock market looks like in a world consisting of only noise traders and funds.

In [7]:
```
def initializeFunds(fundType):
    funds = []
    Fund.maxLeverage = 50
    for beta in range(1, 11):
        funds.append(fundType(beta * 5))
    return funds
```

In [8]:
```
def simulate(time, fundType):
    numpy.random.seed(RANDOM_SEED)
    funds = initializeFunds(fundType)
    days = range(1, time)
    oldprice = 1
    prices = [oldprice]
    noiseTraderInvestment = 1000
    noiseTraderPrices = [noiseTraderInvestment/N]
    fundWealth = []
    for day in days:
        noiseTraderInvestment = calculateNoiseTraderInvestment(noiseTraderInve
stment)
        newprice = findEquilibrium(noiseTraderInvestment, funds)
        currentWealth = []
        for f in funds:
            f.trade(newprice)
            f.checkBankrupt(newprice)
            f.processInflows(oldprice, newprice)
            currentWealth.append(f.getWealth(newprice))

        prices.append(newprice)
        oldprice = newprice
        noiseTraderPrice = min(noiseTraderInvestment/N, maxPrice)

        noiseTraderPrices.append(noiseTraderPrice)
        fundWealth.append(currentWealth)
    return {'days': [0] + list(days), 'prices':prices, 'noiseP':noiseTraderPri
ces, 'wealth':fundWealth}
```

In [9]:
```
def plotResult(result, start=0, end=-1):
    if (end == -1):
        end = len(result['days'])
    plot.figure(figsize=(16,4))
    plot.plot(result['days'], result['prices'], 'b', label='including funds')
    plot.plot(result['days'], result['noiseP'], 'g', label='noise trader
only')
    plot.xlabel('Day')
    plot.ylabel('Price')
    plot.legend(loc='upper left')
    plot.axis([start, end, 0, 2])
    plot.show()
```

In [10]:
```
fundResult = simulate(20000, Fund)
plotResult(fundResult)
```



With the funds investing at undervalued prices, the prices do not fall much below the fundamental value.

As a side effect of following a value stategy, the funds accumulate wealth over time, as the following plot shows.

5.2 Adding the Background Investor

Investors invest or disinvest from each fund based on its recent performance $r_h^{perf}(t)$, which is based on a moving average of recent returns $r(t)$, with $a = 0.1$:

$$r_h^{perf}(t) = (1-a)\, r_h^{perf}(t-1) + a\, r_h(t) \tag{5}$$

$$r(t) = \frac{(\frac{p(t)}{p(t-1)} - 1)\, \xi_h(t-1))}{W_h(t-1)}$$

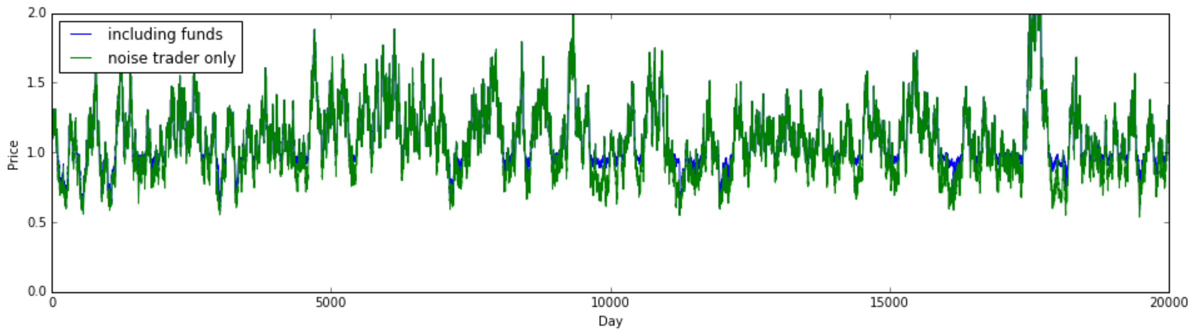
The flow of capital in or out of the fund $F_h(t)$:

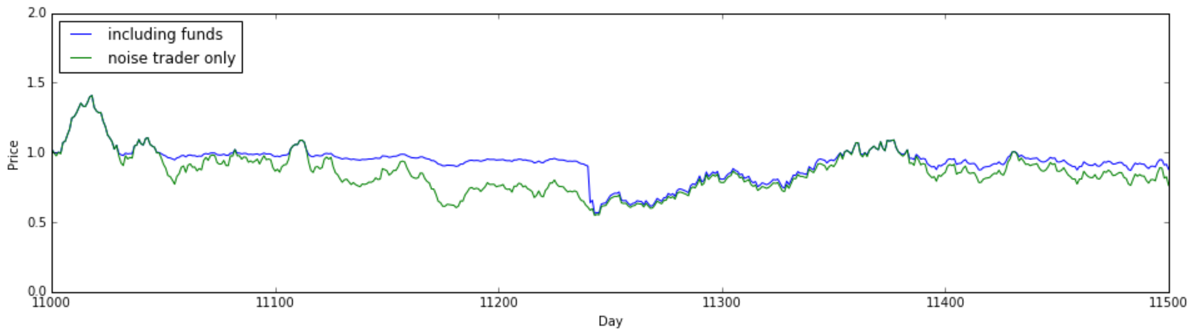
$$F_h(t) = max\big(\text{-}1,\ b\, (r_h^{perf}(t) - r^b)\big)\ W_h(t) \tag{7}$$

with $r^b = 0.005$ being a benchmark return and $b = 0.15$ being a sensitivity parameter. The benchmark return can be used to tune how wealthy the funds can grow. With a higher benchmark return, profits are withdrawn sooner and the asymptotic fund size smaller.

In [11]:
```
# DynamicFund extends Fund by adding inflow/outflow dynamics
class DynamicFund(Fund):

    benchmarkPerformance = 0.005 # r^b
    sensitivity = 0.10 # b, original paper uses 0.15, but 0.10 looks more inte
resting to me

    def __init__(self, aggressiveness):
        super(DynamicFund, self).__init__(aggressiveness)
        self.performance = 0.0
        self.previousWealth = self.initialWealth
        self.previousInvestment = 0.0

    def updatePerformance(self, oldprice, newprice, wealth):
        ret = (newprice/oldprice - 1)*self.previousInvestment/self.previousWea
lth
        self.performance = 0.9 * self.performance + 0.1 * ret # equation 5
        # remember values for next round
        self.previousInvestment = self.shares * newprice
        self.previousWealth = wealth
        return self.performance

    def processInflows(self, oldprice, newprice):
        if self.isActive():
            wealth = self.getWealth(newprice)
            perf = self.updatePerformance(oldprice, newprice, wealth)
            inflow = self.sensitivity*(perf - self.benchmarkPerformance)*wealt
h
            self.cash += max(inflow, -wealth)
```

In [12]:
```
rebalanceResult = simulate(20000, DynamicFund)
plotResult(rebalanceResult)
plotResult(rebalanceResult, 11000, 11500)
```





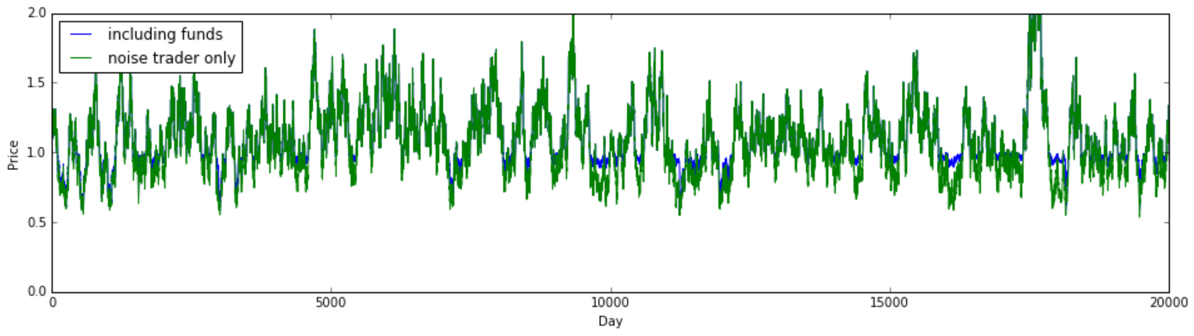
5.3 Replacing the Background Investor with a Wealth Tax

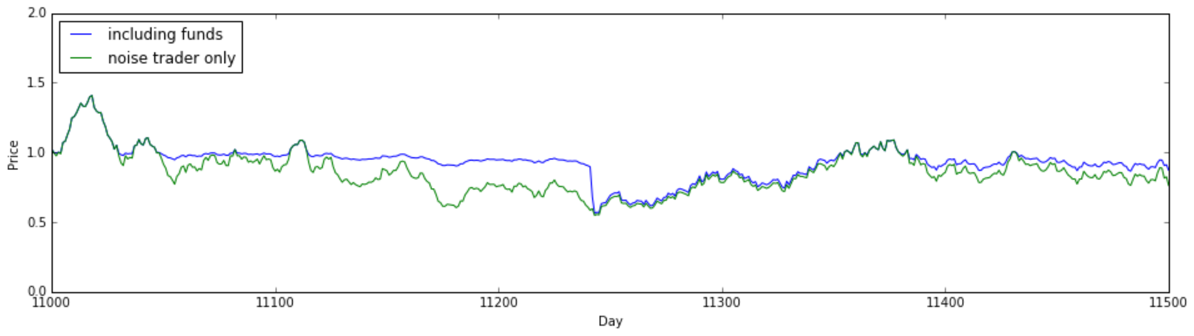
The authors of the original paper say about the background investor: "We introduced this into our model because it guarantees a steady-state behavior, with welldefined long-term statistical averages. Without this the wealth of the funds grows without bound, since the funds consistently profit at the expense of the noise traders. [...] Since the wealth dynamics we have chosen is a form of trend following, it unfortunately introduces some confusion about the source of the heavy tails that we observe here. As we explain later, based on various experiments we are confident that the wealth dynamics of the investors is not the source of the heavy tails."

To verify this claim, let us create a simpler version of the DynamicFund, namely a TaxedFund that pays a daily tax of 0.004% of its wealth.

In [13]:
```
# TaxedFund pays a wealth tax
class TaxedFund(Fund):

    def processInflows(self, oldprice, newprice):
        if self.isActive():
            wealth = self.getWealth(newprice)
            self.cash -= 0.0004*wealth
```

In [14]:
```
taxedResult = simulate(20000, TaxedFund)
plotResult(taxedResult) # full plot
plotResult(taxedResult, 11000, 11500)
```


including funds
noise trader only
Price
Day


including funds
noise trader only
Price
Day