We use the system:

$$\text{CashPosition} = \frac{f(\text{Price})}{\text{Volatility}(\text{Returns})}$$

This is very problematic:

- Prices may live on very different scales, hence trying to find a more universal function f is almost impossible. The sign-function was a good choice as the results don't depend on the scale of the argument.
- Price may come with all sorts of spikes/outliers/problems.

We need a simple price filter process

- We compute volatility-adjusted returns, filter them and compute prices from those returns.
- Don't call it Winsorizing in Switzerland. We apply Huber functions.

In [12]:

```
def filter(price, volatility=32,clip=4.2, min_periods=300):
    r = np.log(price).diff()
    vola = r.ewm(com=volatility, min_periods=min_periods).std()
    price_adj = (r/vola).clip(-clip, clip).cumsum()
    return price_adj
```

Oscillators

- All prices are now following a standard arithmetic Brownian motion with std 1.
- What we want is the difference of two moving means (exponentially weighted) to have a constant std regardless of the two lengths.
- An oscillator is the **scaled difference of two moving averages**.

In [13]:

```
import numpy as np
def osc(prices, fast=32, slow=96, scaling=True):
    f,g = 1 - 1/fast, 1-1/slow
    if scaling:
        s = np.sqrt(1.0 / (1 - f * f) - 2.0 / (1 - f * g) + 1.0 / (1 - g * g))
    else:
        s = 1
    return (prices.ewm(com=fast-1).mean() - prices.ewm(com=slow-1).mean())/s
```

In [14]:

```
from numpy.random import randn
import pandas as pd
price = pd.Series(data=randn(100000)).cumsum()

o = osc(price, 5, 200, scaling=False)
print("The std for the oscillator (Should be close to 1.0):")
print(np.std(o))
```

```
The std for the oscillator (Should be close to 1.0):
9.95027330922014
```

In [15]:

```python
#from pycta.signal import osc

# take two moving averages and apply tanh
def f(price, slow=96, fast=32, clip=3):
    # construct a fake-price, those fake-prices have homescedastic returns
    price_adj = filter(price, volatility=slow, clip=clip)
    # compute mu
    mu = np.tanh(osc(price_adj, fast=fast, slow=slow))
    return mu/price.pct_change().ewm(com=slow, min_periods=300).std()
```

In [16]:

```
portfolio = Portfolio(prices=prices, position=prices.apply(f))
analysis(portfolio.nav())
```

This system is still **univariate** and lacks **integrated risk management**:

$$\text{CashPosition} = \frac{f(\text{Filter}(\text{Price}))}{\text{Volatility}(\text{Returns})} = \frac{\mu}{\text{Volatility}}$$

Some hedge funds stop here. All energy goes into constructing μ.

- Suitable as it is possible to add/remove additional systems on the fly.
- A typical CTA would run with a set of 5 or 6 functions f acting on approx. 100 assets.
- Organisation by asset group.
- Scaled signals make it easy to apply functions such as $\tanh$ or combine various signals in a regression problem.

In [ ]: