In [1]:

```
import talk.config as con

% matplotlib inline

con.config_mosek()
con.config_configManager()
con.config_matplotlib()
```

```
Set MOSEKLM_LICENSE_FILE environment variable
Update ConfigManager
```

Constructing estimators

https://en.wikipedia.org/wiki/Autoregressive_model
(https://en.wikipedia.org/wiki/Autoregressive_model)

Thomas Schmelzer

A very common estimator is based on AR models (autoregressive)

$$R_T = \sum_{i=1}^{n} w_i r_{T-i}$$

Predict the (unknown) return R_T using the last n previous returns. **Attention**: You may want to use volatility adjusted returns, apply filters etc.

How to pick the n free parameters in $\mathbf{w}$? (Partial) autocorrelations?

In [2]:

```
def convolution(ts, weights):
    from statsmodels.tsa.filters.filtertools import convolution_filter
    return convolution_filter(ts, weights, nsides=1)
```

In [3]:

```python
import pandas as pd

r = pd.Series([1.0, -2.0, 1.0, 1.0, 1.5, 0.0, 2.0])
weights = [2.0, 1.0]
# trendfollowing == positive weights
x=pd.DataFrame()
x["r"] = r
x["pred"] = convolution(r, weights)
x["before"] = x["pred"].shift(1)
print(x)
print(x.corr())
```

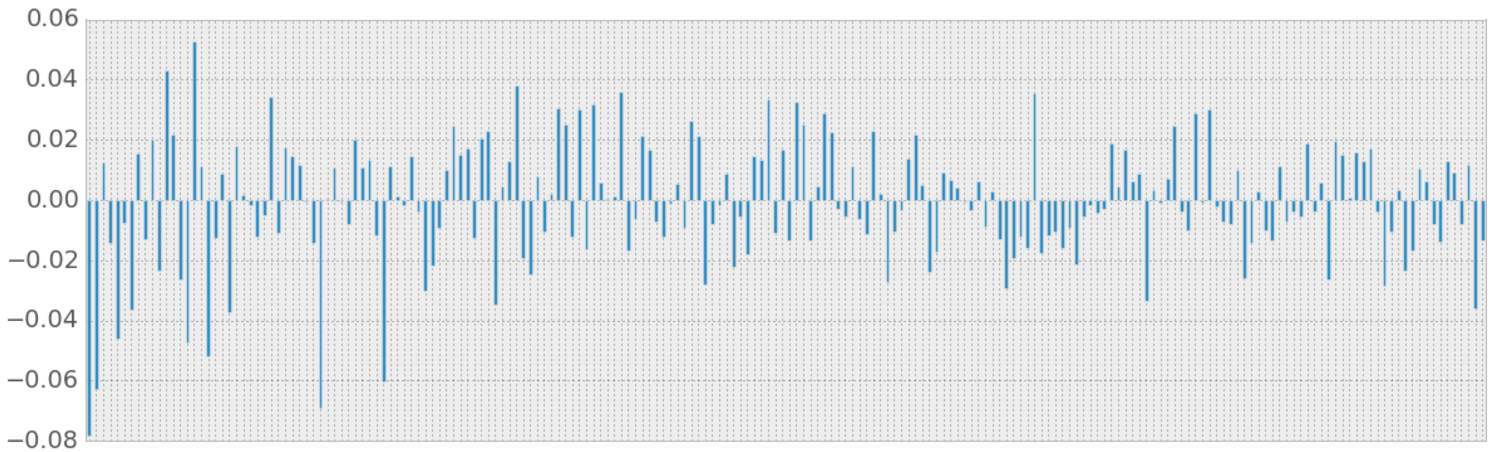
```
     r  pred  before
0  1.0   NaN     NaN
1 -2.0  -3.0     NaN
2  1.0   0.0    -3.0
3  1.0   3.0     0.0
4  1.5   4.0     3.0
5  0.0   1.5     4.0
6  2.0   4.0     1.5
               r      pred    before
r       1.000000  0.895788 -0.190159
pred    0.895788  1.000000  0.538431
before -0.190159  0.538431  1.000000
```

Looking only at the last two returns might be a bit ...

Is it a good idea to have $n = 200$ free parameters?

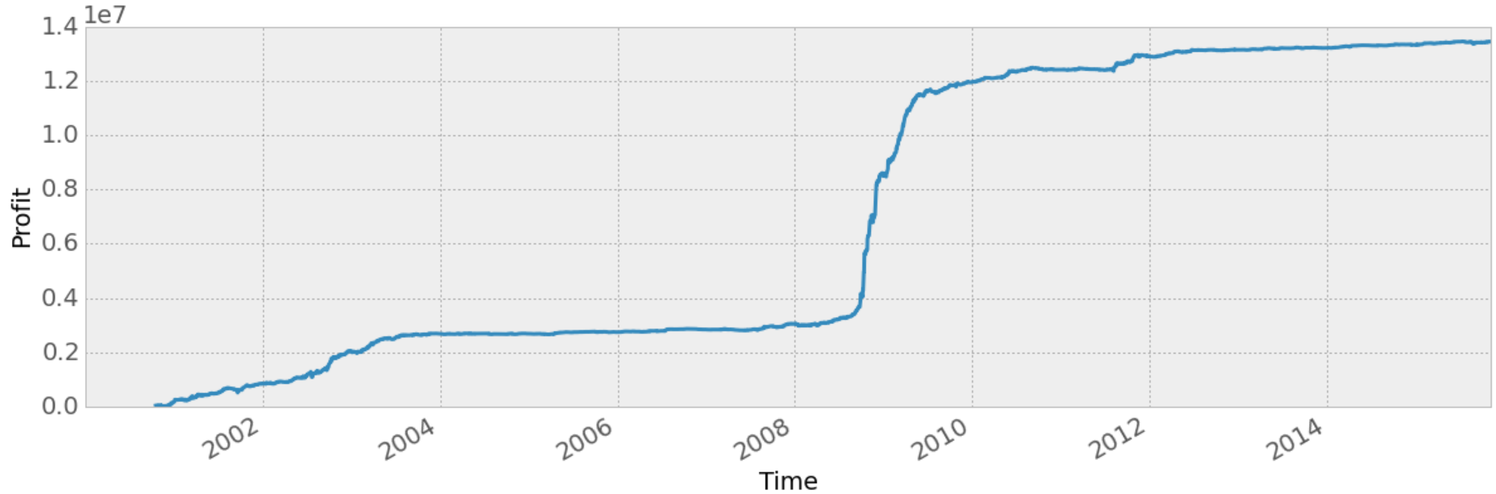
In [15]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import statsmodels.tsa.stattools as sts
# generate random returns
import pandas as pd
r = pd.read_csv("SPX_Index.csv", squeeze=True, index_col=0, parse_dates=True).pct_ch
ange().dropna()
# let's compute the optimal convolution!
weights = sts.pacf(r, nlags=200)
pd.Series(data=weights[1:]).plot(kind="bar")
plt.show()
```



In [6]:

```
# The trading system!
pos = convolution(r, weights[1:])
pos = 1e6*(pos/pos.std())
# profit = return[today] * position[yesterday]
(r*pos.shift(1)).cumsum().plot()
plt.xlabel('Time'), plt.ylabel('Profit')
plt.show()
```



Bias

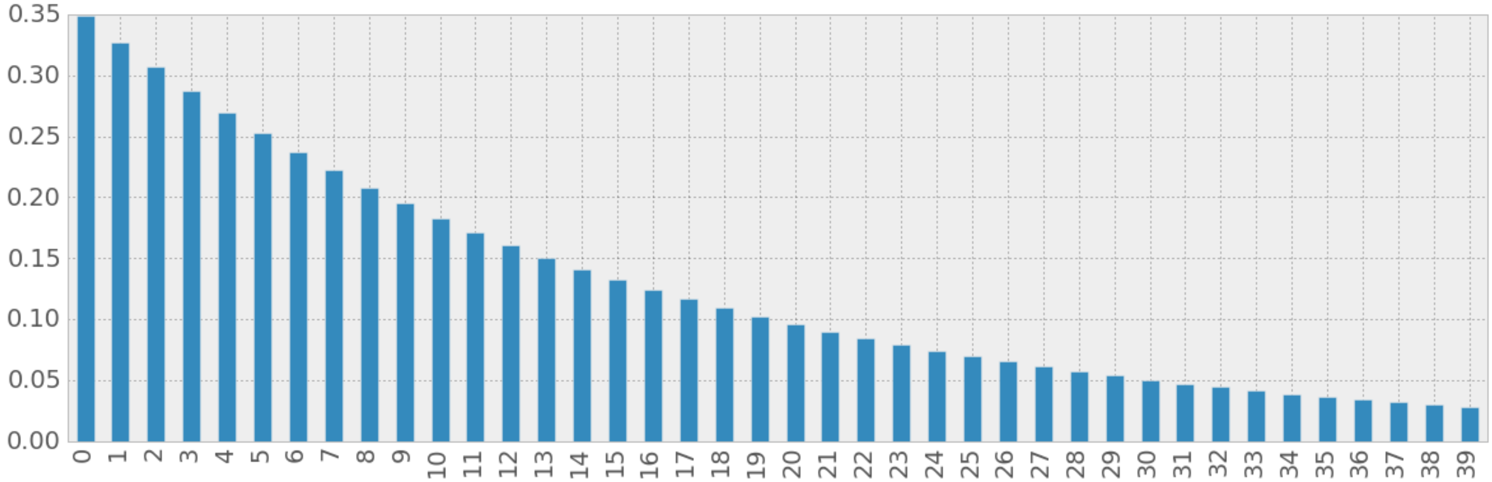
We assume the weights are exponentially decaying, e.g.

$$w_i = \frac{1}{S}\lambda^i$$

where S is a suitable scaling constant and $\lambda = 1 - 1/N$. Note that $N \neq n$.

Everything that is **not** an exponentially weighted moving average is **wrong**.

In [7]:
```
import numpy as np

def exp_weights(m, n=100):
    x = np.power(1.0 - 1.0/m, range(1,n+1))
    S = np.linalg.norm(x)
    return x/S

pd.Series(exp_weights(m=16,n=40)).plot(kind="bar")
plt.show()
```

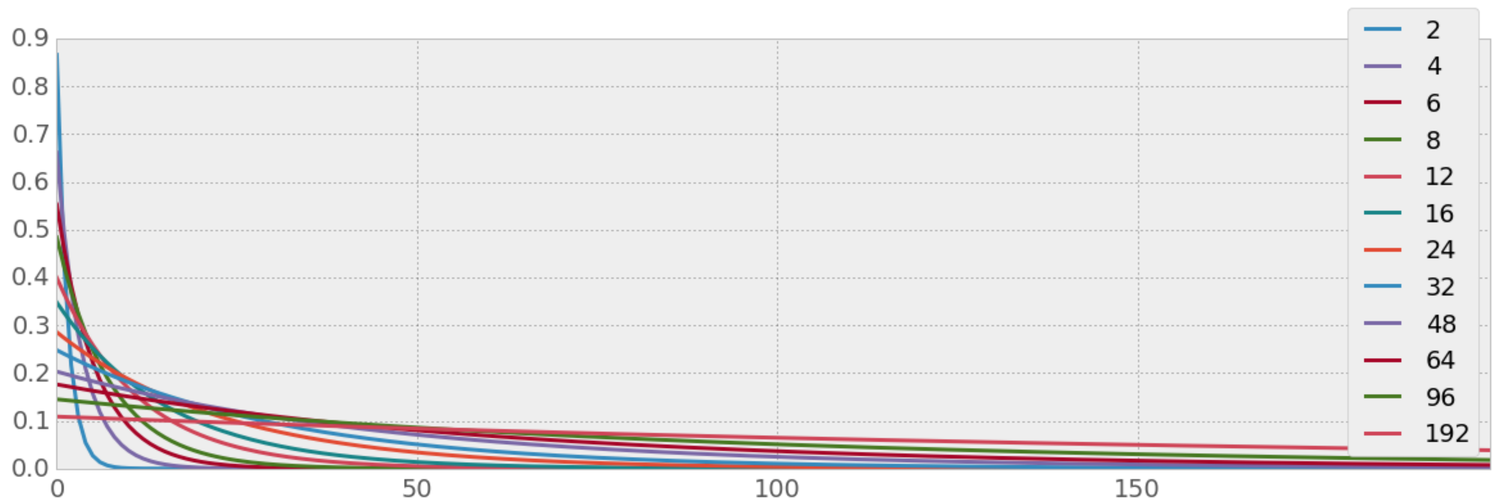


In [8]:

```python
import numpy as np
import pandas as pd

periods = [2,4,6,8,12,16,24,32,48,64,96,192]
# matrix of weights
W = pd.DataFrame({period : exp_weights(m=period, n=200) for period in periods})
W.plot()
```

Out[8]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fc5e8218350>
```



In [9]:
```
# each column of A is a convoluted return time series
A = pd.DataFrame({period : convolution(r, W[period]).shift(1) for period in period
s})

A = A.dropna(axis=0)
r = r[A.index].dropna()
```

(Naive) regression

$$\mathbf{w}^* = \arg\min_{\mathbf{w}\in\mathbb{R}^m} \|\mathbf{A}\mathbf{w} - \mathbf{r}\|_2$$

Mean variation

We provide a few indicators. Avoid fast indicators. Prefer slower indicators as they induce less trading costs. Use the mean variation of the signal (convoluted returns here)

$$f(\mathbf{x}) = \frac{1}{n} \sum |x_i - x_{i-1}| = \frac{1}{n} \|\Delta \mathbf{x}\|_1$$

The ith column of $\mathbf{A}$ has a mean variation d_i. We introduce the diagonal penalty matrix $\mathbf{D}$ with $D_{i,i} = d_i$.

$$\mathbf{w}^* = \arg \min_{\mathbf{w} \in \mathbb{R}^m} \|\mathbf{A}\mathbf{w} - \mathbf{r}\|_2 + \lambda \|\mathbf{D}\mathbf{w}\|_1$$

In [11]:

```
def mean_variation(ts):
    return ts.diff().abs().mean()

d = A.apply(mean_variation)
D = np.diag(d)

from mosek.fusion import *
# but you could use Mosek:
def __two_norm(model, v):
    # add variable to model for the 2-norm of the residual
    x = model.variable(1, Domain.greaterThan(0.0))
    # add the quadratic cone
    model.constraint(Expr.vstack(x,v), Domain.inQCone())
    return x


def __one_norm(model, v):
    t = model.variable(int(v.size()), Domain.greaterThan(0.0))
    model.constraint(Expr.hstack(t, v), Domain.inQCone(int(v.size()), 2))
    return Expr.sum(t)


def ar(A, D, r, lamb=0.0):

    with Model('lasso') as model:
        n = int(A.shape[1])
        # introduce the variable for the var
        x = model.variable("x", n, Domain.unbounded())
        # minimization of the conditional value at risk
        a1 = __two_norm(model, Expr.sub(Expr.mul(DenseMatrix(A),x), Expr.constTer
m(r)))
        a2 = __one_norm(model, Expr.mul(DenseMatrix(D),x))

        model.objective(ObjectiveSense.Minimize, Expr.add(a1, Expr.mul(a2, float(lam
b))))
```

```
        model.solve()

        return x.level()
```

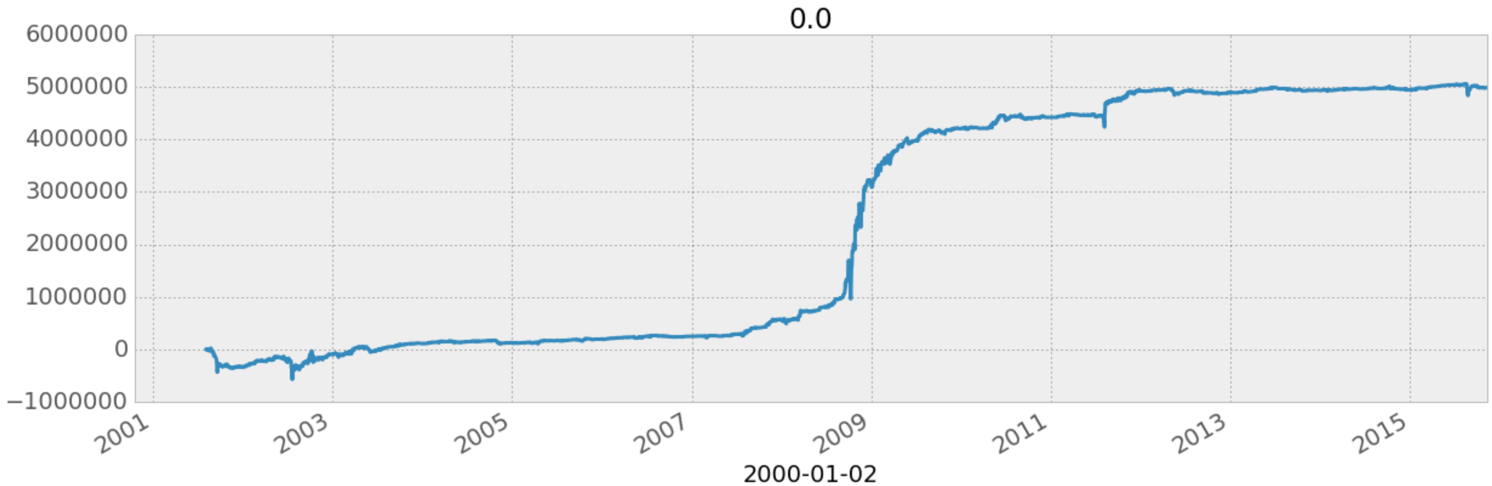
In [16]:
```
t_weight = dict()
for lamb in [0.0, 0.2, 0.4, 0.8, 1.4, 2.0, 2.5, 3.0, 3.5, 5.0, 7.0, 9.0, 12.0, 15.0, 
20.0, 30.0]:
    weights = pd.Series(index=periods, data=ar(A.values, D, r.values, lamb=lamb))
    print(weights.transpose())
    t_weight[lamb] = (W*weights).sum(axis=1)
    t_weight[lamb].plot(kind="bar")
    plt.title(lamb)
    plt.show()
```

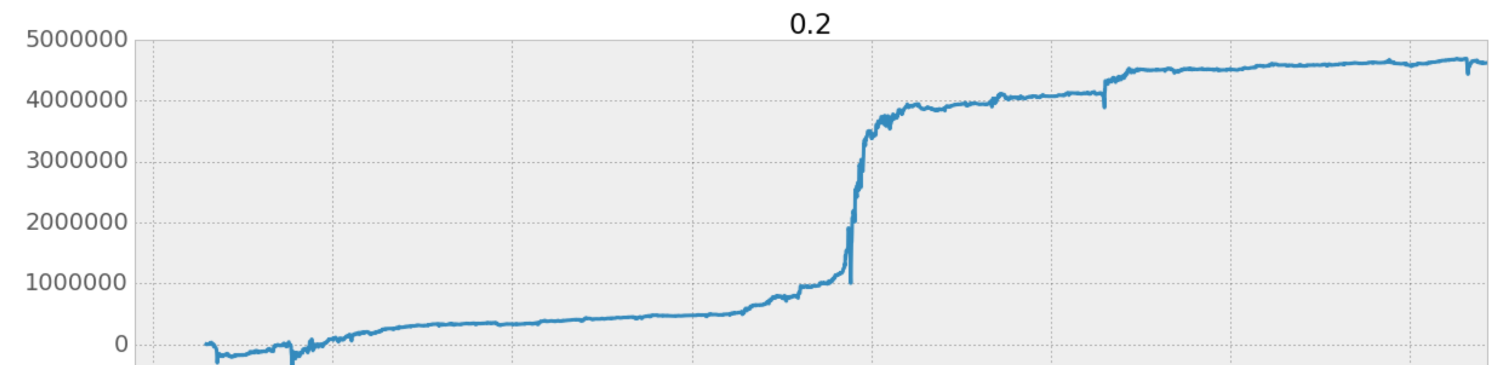
```
---------------------------------------------------------------------------
LengthError                               Traceback (most recent call last)
<ipython-input-16-c5c40f916006> in <module>()
      1 t_weight = dict()
      2 for lamb in [0.0, 0.2, 0.4, 0.8, 1.4, 2.0, 2.5, 3.0, 3.5, 5.0, 7.0, 9.0, 
12.0, 15.0, 20.0, 30.0]:
----> 3     weights = pd.Series(index=periods, data=ar(A.values, D, r.values, lam
b=lamb))
      4     print(weights.transpose())
      5     t_weight[lamb] = (W*weights).sum(axis=1)

<ipython-input-11-2d584a9efb43> in ar(A, D, r, lamb)
     28         x = model.variable("x", n, Domain.unbounded())
     29         # minimization of the conditional value at risk
---> 30         a1 = __two_norm(model, Expr.sub(Expr.mul(DenseMatrix(A),x), Exp
r.constTerm(r)))
     31         a2 = __one_norm(model, Expr.mul(DenseMatrix(D),x))
     32 

/home/thomas/thalesians/env/lib/python2.7/site-packages/mosek/fusion/__init__.pyc
in sub(*args)
  24938       return Expr._sub__3F_3F(*args)
  24939     elif Expr._matchargs_sub_Omosek_fusion_Expression_2Omosek_fusion_Expr
ession_2(args):
> 24940       return Expr._sub_Omosek_fusion_Expression_2Omosek_fusion_Expressio
```

```
n_2(*args)
  24941     else:
  24942       argtypestr = "sub(%s)" % ",".join([_argtypestr(a) for a in args])

/home/thomas/thalesians/env/lib/python2.7/site-packages/mosek/fusion/__init__.pyc
in _sub_0mosek_fusion_Expression_2Omosek_fusion_Expression_2(lhs, rhs)
```

In [13]:

```
for lamb in sorted(t_weight.keys()):
    #www = t_weight[3.0]
    pos = convolution(r, t_weight[lamb])
    pos = 1e6*(pos/pos.std())
    (r*pos.shift(1)).cumsum().plot()
    plt.title(lamb)
    plt.show()
```





Summary

- The problem of constructing an estimator is corresponds to tracking an index. The index is here a given historic return time series. The assets are standard estimators you can pick. Go wild here. Don't restrict yourself to moving averages!
- Using the (mean) total variation of your signals you can prefer slower signals as they trade cheaper.
- Using LARS you can establish a ranking amongst the indicators and construct them robustly.
- You can (vertical) stack the resulting systems to find optimal weights across a group of assets.

Back to Overview (http://localhost:8888)