Market-Based Valuation of Equity Options

A Python-based Journey

Dr Yves J Hilpisch

The Python Quants GmbH

CQF Lecture, 14. April 2016, London

About Me

I am a Python **entrepreneur** — Python and Open Source technologies, consulting, development and training for finance and data science.

In [2]:
```
from IPython.display import Image
```

In [3]:
```
Image('http://hilpisch.com/tpq_logo.png', width=500)
```

Out[3]:

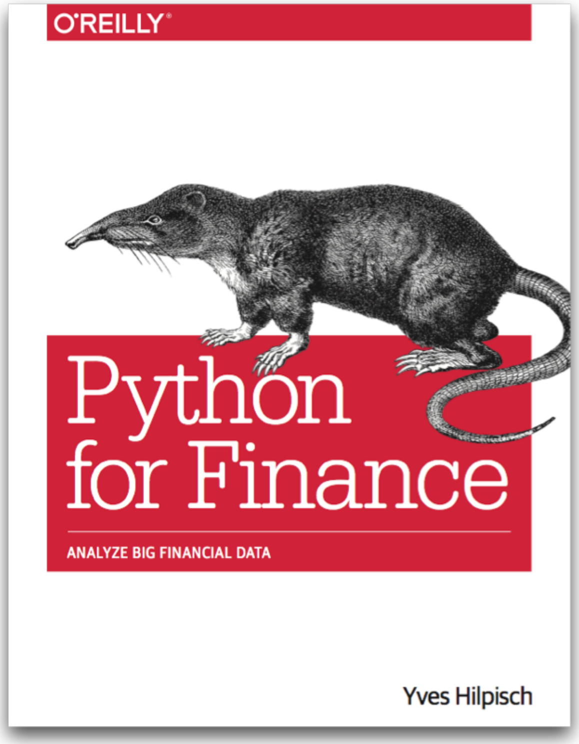
In [4]:
```
Image('http://datapark.io/img/logo.png', width=500)
```

Out[4]:

I am an **author** (I) — http://pff.tpq.io (http://pff.tpq.io).

In [5]:
```
Image('http://hilpisch.com/images/python_for_finance.png', width=300)
```

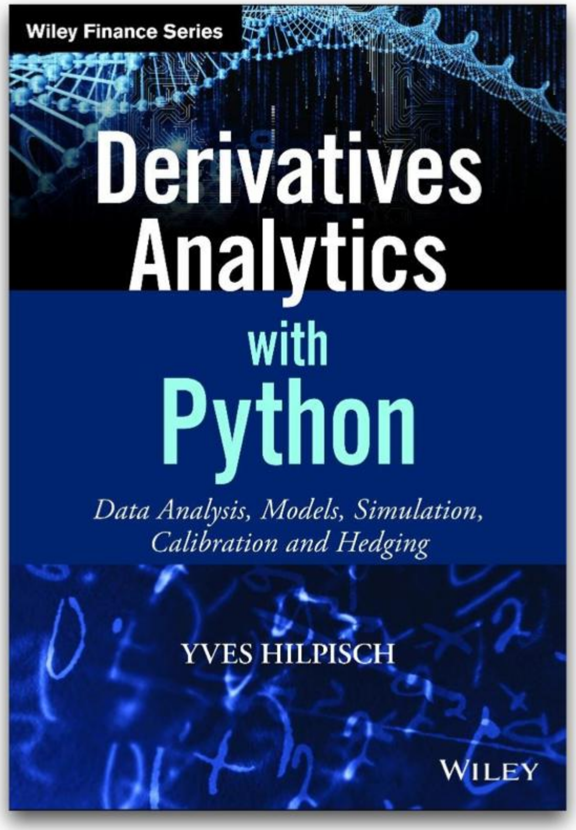
Out[5]:



I am an **author** (II) — [http://dawp.tpq.io (http://dawp.tpq.io)](http://dawp.tpq.io)

In [6]:
```
Image('http://hilpisch.com/images/derivatives_analytics_front.jpg', width=300)
```

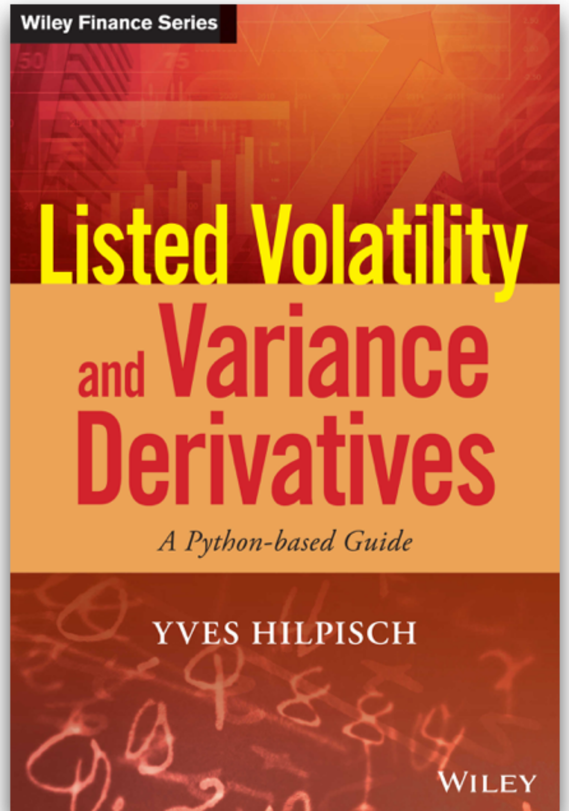
Out[6]:



I am an **author** (III).

In [7]:
```
Image('http://hilpisch.com/images/lvvd_cover.png', width=300)
```

Out[7]:



I organize a number of **events**.

- For Python Quants Conference and Workshops — http://fpq.io (http://fpq.io)
- Open Source in Quant Finance Conference — http://osqf.tpq.io (http://osqf.tpq.io)
- Python for Quant Finance Meetup Group London — http://pqf.tpq.io (http://pqf.tpq.io)
- Python for Finance Certification — http://pfc.tpq.io (http://pdc.tpq.io)

More information and further links under http://tpq.io (http://fpq.io) and http://hilpisch.com (http://hilpisch.com).

Agenda

- Benchmark Case of Normally Distributed Returns
- Market Stylized Facts about Index Prices and Equity Options
- Fourier-based Option Pricing
- Merton (1976) Jump-Diffusion Model
- Monte Carlo Simulation in the Merton (1976) Model
- Calibration of the Merton (1976) Model to Market Quotes

Go to http://derivatives-analytics-with-python.com (http://derivatives-analytics-with-python.com) to find links to all the resources and Python codes (eg Quant Platform, Github repository).

The Benchmark Case

Let us first set the stage with **standard normally distributed (pseudo-) random numbers** ...

In [8]:
```
import numpy as np
a = np.random.standard_normal(1000)
```

In [9]:
```
a.mean()
```

Out[9]:
```
0.018485352902666726
```

In [10]:
```
a.std()
```

Out[10]:
```
1.0084113501147707
```

... and a simulated **geometric Brownian motion** (GBM) path. We make the following assumptions.

In [11]:
```
import math
import pandas as pd
# model parameters
S0 = 100.0  # initial index level
T = 10.0  # time horizon
r = 0.05  # risk-less short rate
vol = 0.2  # instantaneous volatility

# simulation parameters
np.random.seed(250000)
gbm_dates = pd.DatetimeIndex(start='30-09-2004',
                             end='31-08-2015',
                             freq='B')
M = len(gbm_dates)  # time steps
dt = 1 / 252.  # fixed for simplicity
df = math.exp(-r * dt)  # discount factor
```

This **function** simulates GBM paths given the assumptions.

In [12]:

```
def simulate_gbm():
    # stock price paths
    rand = np.random.standard_normal((M, I))  # random numbers
    S = np.zeros_like(rand)  # stock matrix
    S[0] = S0  # initial values
    for t in range(1, M):  # stock price paths
        S[t] = S[t - 1] * np.exp((r - vol ** 2 / 2) * dt
                        + vol * rand[t] * math.sqrt(dt))

    gbm = pd.DataFrame(S[:, 0], index=gbm_dates, columns=['index'])
    gbm['returns'] = np.log(gbm['index'] / gbm['index'].shift(1))


    # Realized Volatility (eg. as defined for variance swaps)
    gbm['rea_var'] = 252 * np.cumsum(gbm['returns'] ** 2) / np.arange(len(gbm))
    gbm['rea_vol'] = np.sqrt(gbm['rea_var'])
    gbm = gbm.dropna()
    return gbm
```

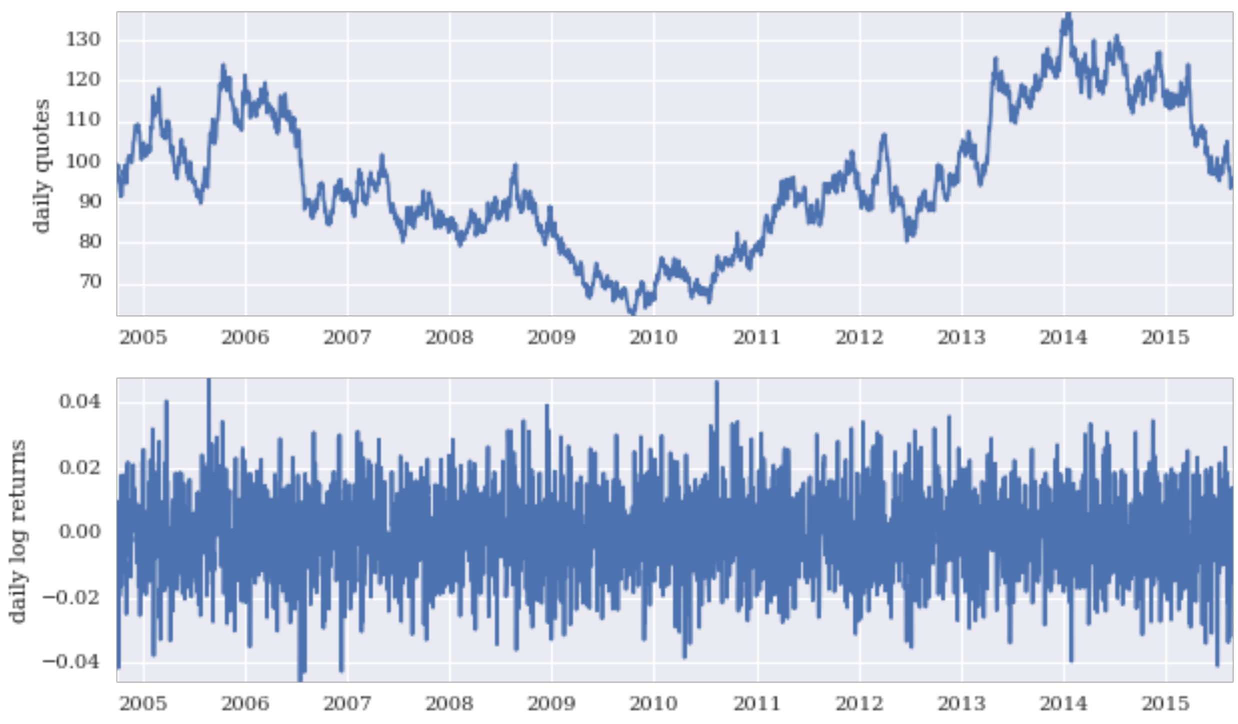
Let us simulate a single path and inspect **major statistics**.

In [13]:
```
from gbm_helper import *
I = 1  # index level paths
gbm = simulate_gbm()
print_statistics(gbm)
```

```
RETURN SAMPLE STATISTICS
---------------------------------------------
Mean of Daily  Log Returns -0.000017
Std  of Daily  Log Returns  0.012761
Mean of Annua. Log Returns -0.004308
Std  of Annua. Log Returns  0.202578
---------------------------------------------
Skew of Sample Log Returns -0.037438
Skew Normal Test p-value    0.413718
---------------------------------------------
Kurt of Sample Log Returns  0.106754
Kurt Normal Test p-value    0.239124
---------------------------------------------
Normal Test p-value         0.358108
---------------------------------------------
Realized Volatility         0.202578
Realized Variance           0.041038
```

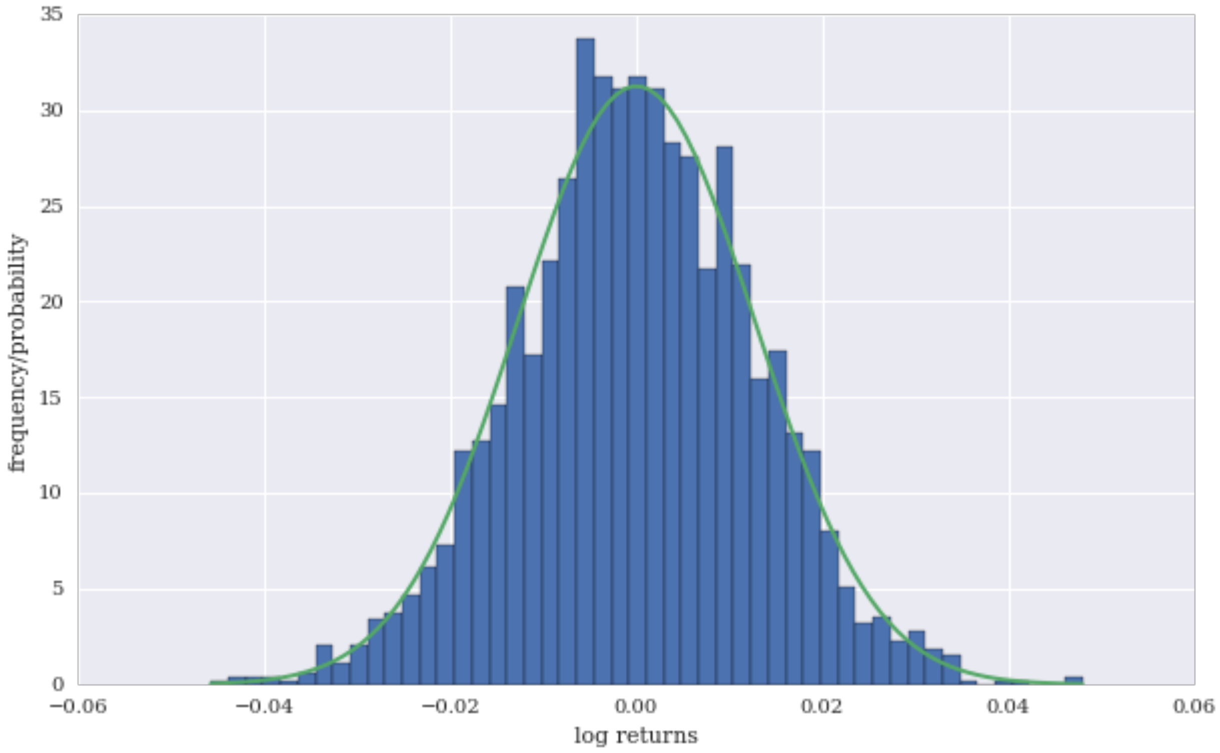
Simulated **prices and resulting log returns** visulized.

In [14]:
```
%matplotlib inline
quotes_returns(gbm)
```



A histogram of the **log returns compared to the normal distribution** (with same mean/std).

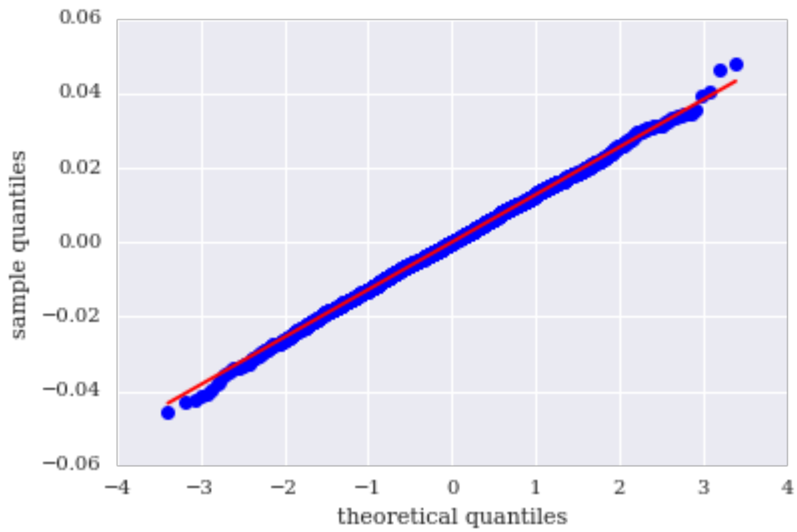
In [15]:
```
return_histogram(gbm)
```



And a Quantile-Quantile QQ-plot of the log returns.

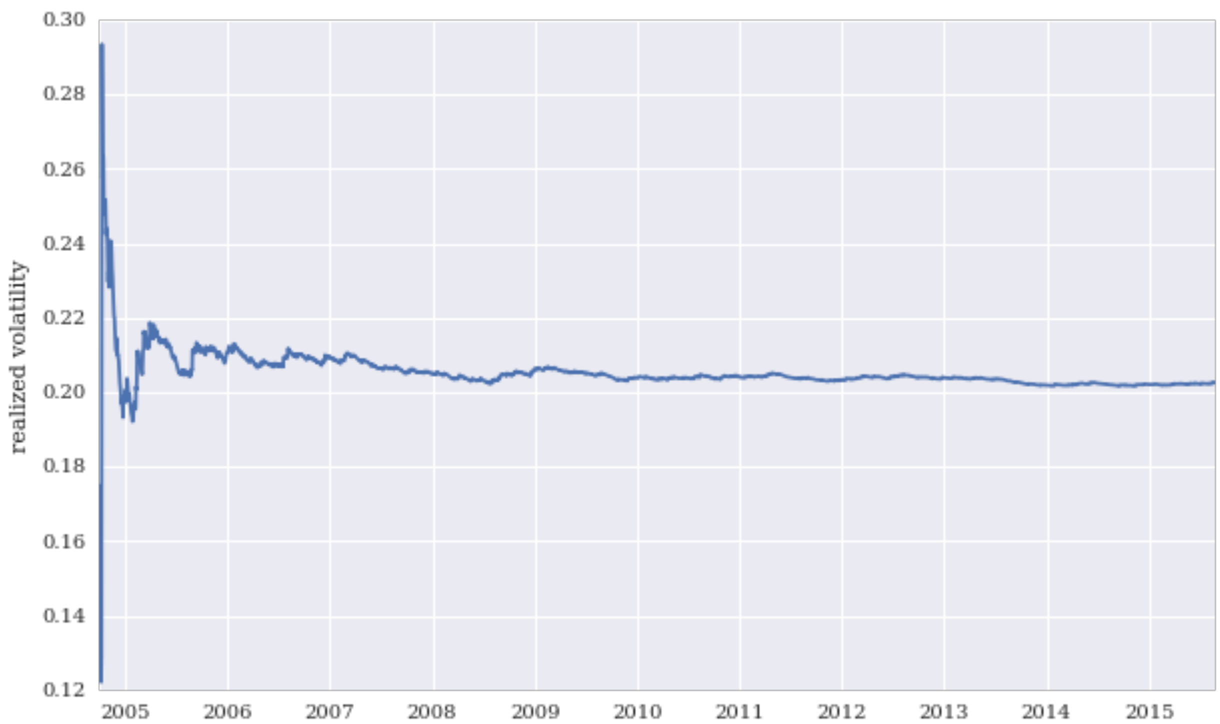
In [16]:
```
return_qqplot(gbm)
```

```
<matplotlib.figure.Figure at 0x7fc05f43b3d0>
```



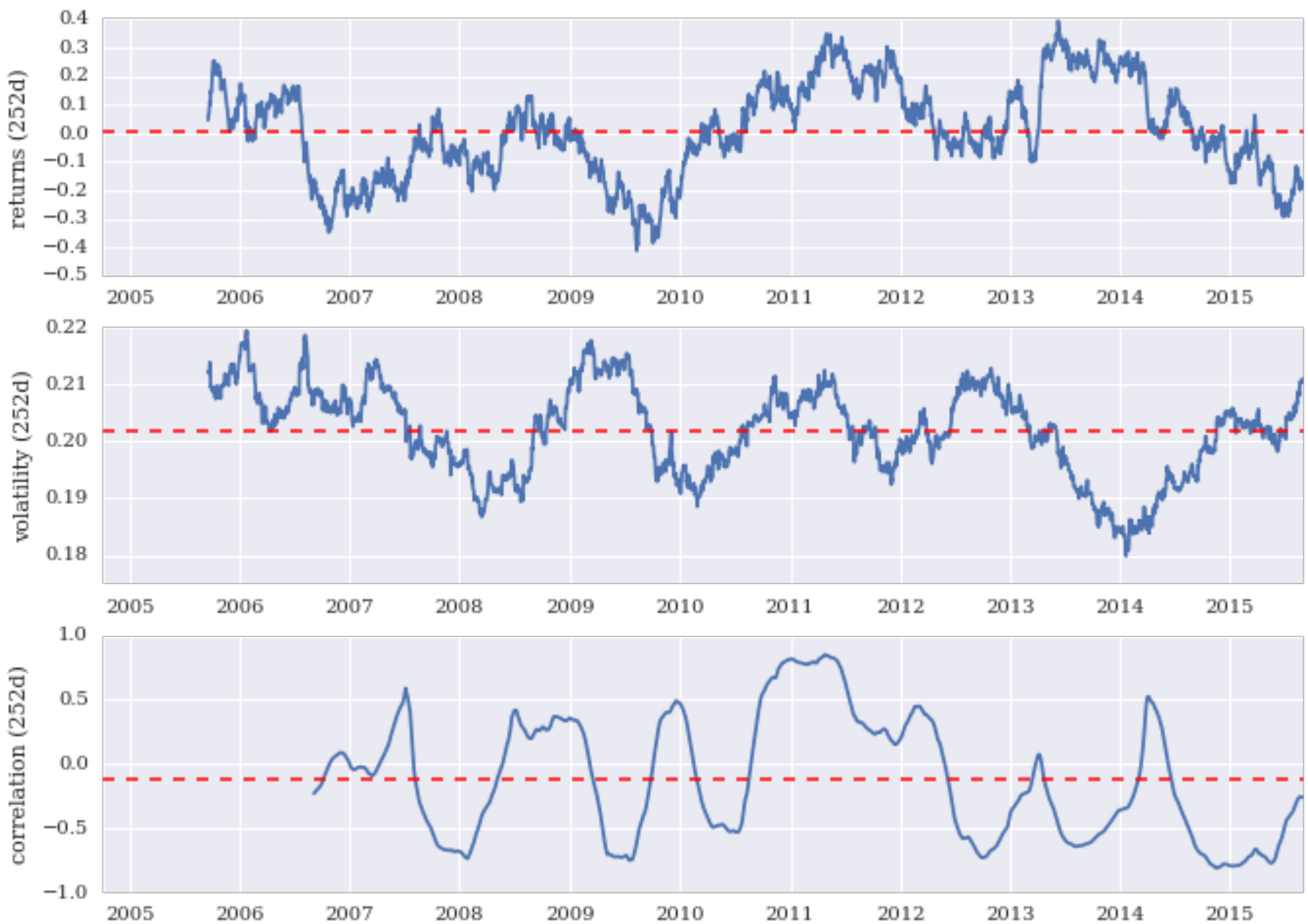
The **realized volatility** over time.

In [17]:
```
realized_volatility(gbm)
```



Some **rolling annualized statistics**.

In [18]:
```
rolling_statistics(gbm)
```



Market Stylized Facts

We work with **historical DAX data**. The following function equips us with the necessary time series data.

In [19]:
```
import pandas.io.data as web

def read_dax_data():
    ''' Reads historical DAX data from Yahoo! Finance, calculates log returns,
    realized variance and volatility.'''
    DAX = web.DataReader('^GDAXI', data_source='yahoo',
                    start='30-09-2004', end='31-08-2015')
    DAX.rename(columns={'Adj Close' : 'index'}, inplace=True)
    DAX['returns'] = np.log(DAX['index'] / DAX['index'].shift(1))
    DAX['rea_var'] = 252 * np.cumsum(DAX['returns'] ** 2) / np.arange(len(DAX))
    DAX['rea_vol'] = np.sqrt(DAX['rea_var'])
    DAX = DAX.dropna()
    return DAX
```

Lets **retrieve and inspect** the data.

In [20]:
```
%time DAX = read_dax_data()
```

```
CPU times: user 123 ms, sys: 7.97 ms, total: 131 ms
Wall time: 427 ms
```

In [21]:
```
print_statistics(DAX)
```

```
RETURN SAMPLE STATISTICS
---------------------------------------------
Mean of Daily  Log Returns  0.000348
Std  of Daily  Log Returns  0.013820
Mean of Annua. Log Returns  0.087652
Std  of Annua. Log Returns  0.219385
---------------------------------------------
Skew of Sample Log Returns  0.013407
Skew Normal Test p-value    0.772079
---------------------------------------------
Kurt of Sample Log Returns  6.559547
Kurt Normal Test p-value    0.000000
---------------------------------------------
Normal Test p-value         0.000000
---------------------------------------------
Realized Volatility         0.219454
Realized Variance           0.048160
```

The (in-memory) **data structure**.

In [22]:
```
DAX.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2786 entries, 2004-10-01 to 2015-08-31
Data columns (total 9 columns):
Open       2786 non-null float64
High       2786 non-null float64
Low        2786 non-null float64
Close      2786 non-null float64
Volume     2786 non-null int64
index      2786 non-null float64
returns    2786 non-null float64
rea_var    2786 non-null float64
rea_vol    2786 non-null float64
dtypes: float64(8), int64(1)
memory usage: 217.7 KB
```

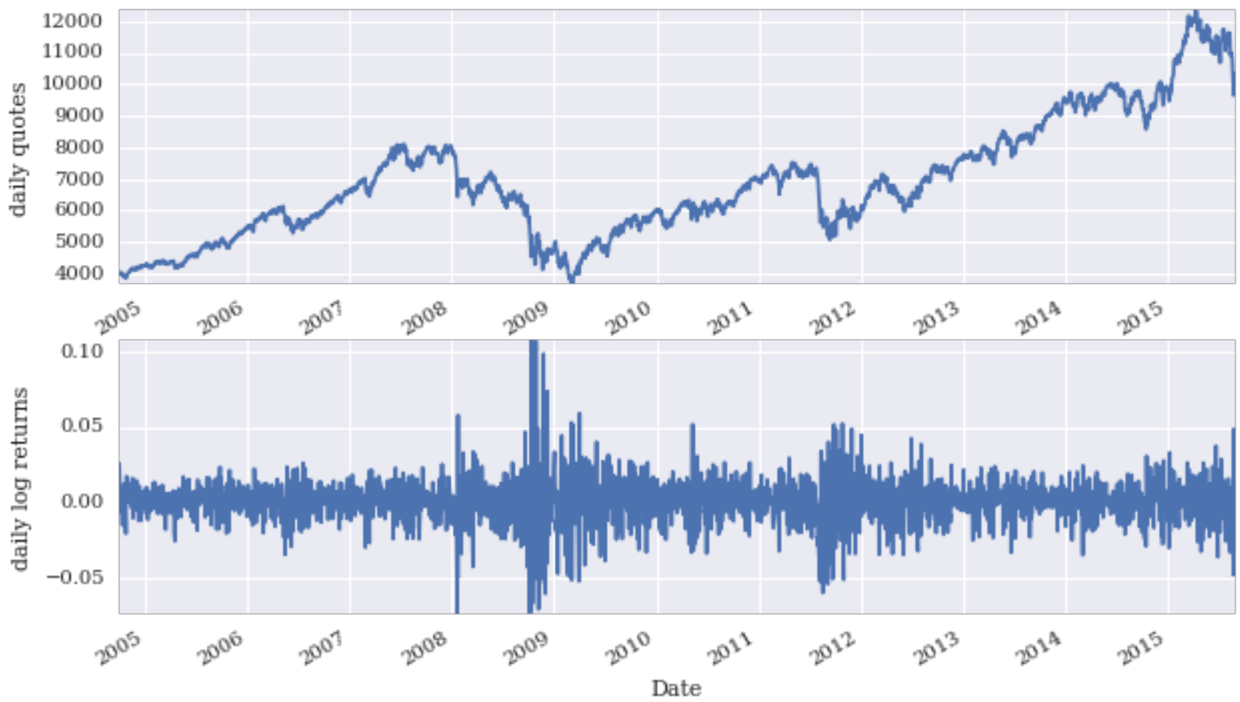
In [23]:
```
DAX[['index', 'returns', 'rea_var', 'rea_vol']].tail()
```

Out[23]:

	index	returns	rea_var	rea_vol
Date				
2015-08-25	10128.120117	0.048521	0.048124	0.219371
2015-08-26	9997.429688	-0.012988	0.048122	0.219366
2015-08-27	10315.620117	0.031331	0.048193	0.219529
2015-08-28	10298.530273	-0.001658	0.048176	0.219491
2015-08-31	10259.459961	-0.003801	0.048160	0.219454

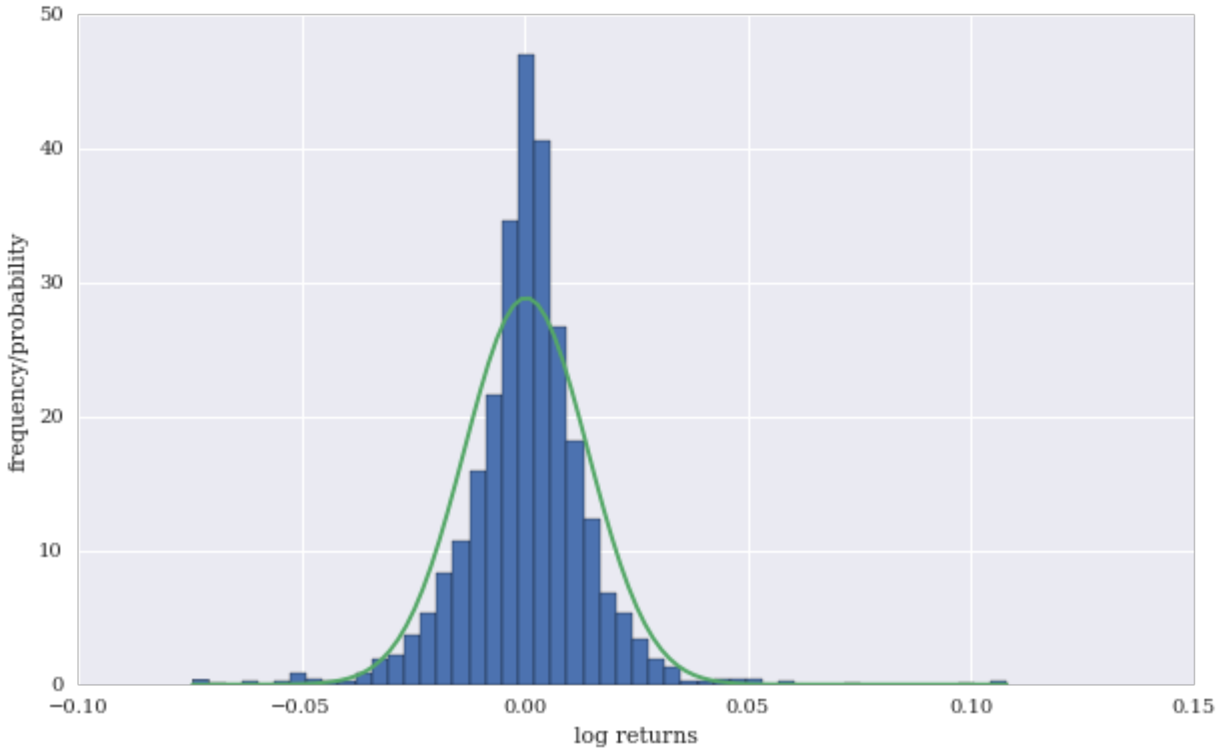
The **index levels and log returns**.

In [24]:
```
quotes_returns(DAX)
```



A histogram of the **log returns compared to the normal distribution** (with same mean/std).

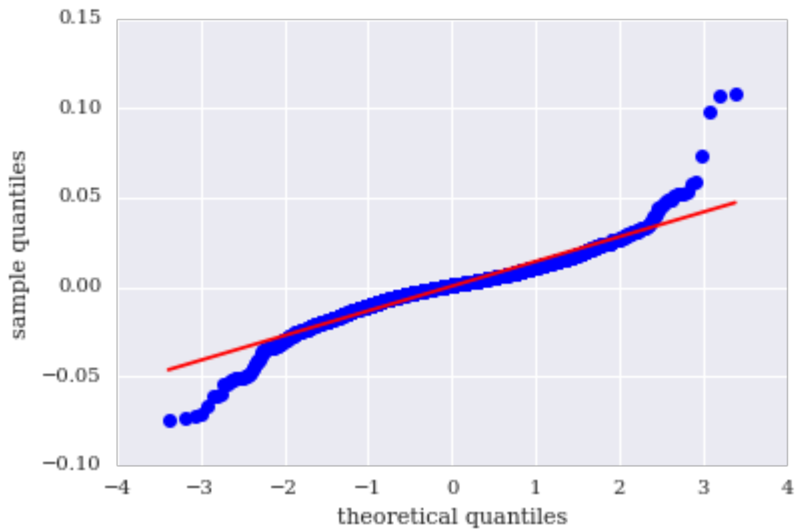
In [25]:
```
return_histogram(DAX)
```



The **QQ-plot**.

In [26]:
```
return_qqplot(DAX)
```

<matplotlib.figure.Figure at 0x7fc05f476190>



The **realized volatility**.

In [27]:
```
realized_volatility(DAX)
```

And finally the **rolling annualized statistics**.

In [28]:
```
rolling_statistics(DAX)
```

Finally, we want to look for **jumps** (heuristically). We use this simple function.

In [29]:
```
def count_jumps(data, value):
    ''' Counts the number of return jumps as defined in size by value. '''
    jumps = np.sum(np.abs(data['returns']) > value)
    return jumps
```

We define a **jump** as a log return higher in absolute value than 0.05.

In [30]:
```
count_jumps(DAX, 0.05)  # "jumps" in the DAX index
```
Out[30]:
```
31
```

In [31]:
```
count_jumps(gbm, 0.05)  # "jumps" in the GBM path
```
Out[31]:
```
0
```

In a Gaussian setting we have

- negative jumps:
$$P(r_n < -0.05) = 0.0002911$$
- positive jumps:
$$P(r_n > +0.05) = 0.0003402$$

for the DAX index given a return observation r_n. In such a setting the number of return observations lower than -0.05 and higher than $+0.05$ would be expected to be:

In [32]:
```
0.0002911 * len(DAX)  # 'lower than -0.05'
```

Out[32]:
```
0.8110046
```

In [33]:
```
0.0003402 * len(DAX)  # 'higher than +0.05'
```

Out[33]:
```
0.9477971999999999
```

In summary, we "discover" the following stylized facts:

- **stochastic volatility**: volatility is neither constant nor deterministic; there is no mechanism to forecast volatility at a high confidence level
- **volatility clustering**: empirical data suggests that high volatility events seem to cluster in time; there is often a positive autocorrelation of volatility measures
- **volatility mean reversion**: volatility is a mean-reverting quantity — it never reaches zero nor does it go to infinity; however, the mean can change over time
- **leverage effect**: our data suggests that volatility is negatively correlated (on average) with asset returns; if return measures increase, volatility measures often decrease and vice versa
- **fat tails**: compared to a normal distribution large positive and negative index returns are more frequent
- **jumps**: index levels may move by magnitudes that cannot be explained within a Gaussian, i.e. normal, diffusion setting; some jump component may be necessary to explain certain large moves

Important add-on topic: **volatility smiles and term structure** — here implied volatilities from European call options on the EURO STOXX 50 on 30. September 2014.

In [34]:
```
%run es50_imp_vol.py
```

```
<matplotlib.figure.Figure at 0x7fc05ec7b750>
```

In [35]:
```
data.tail()
```

Out[35]:

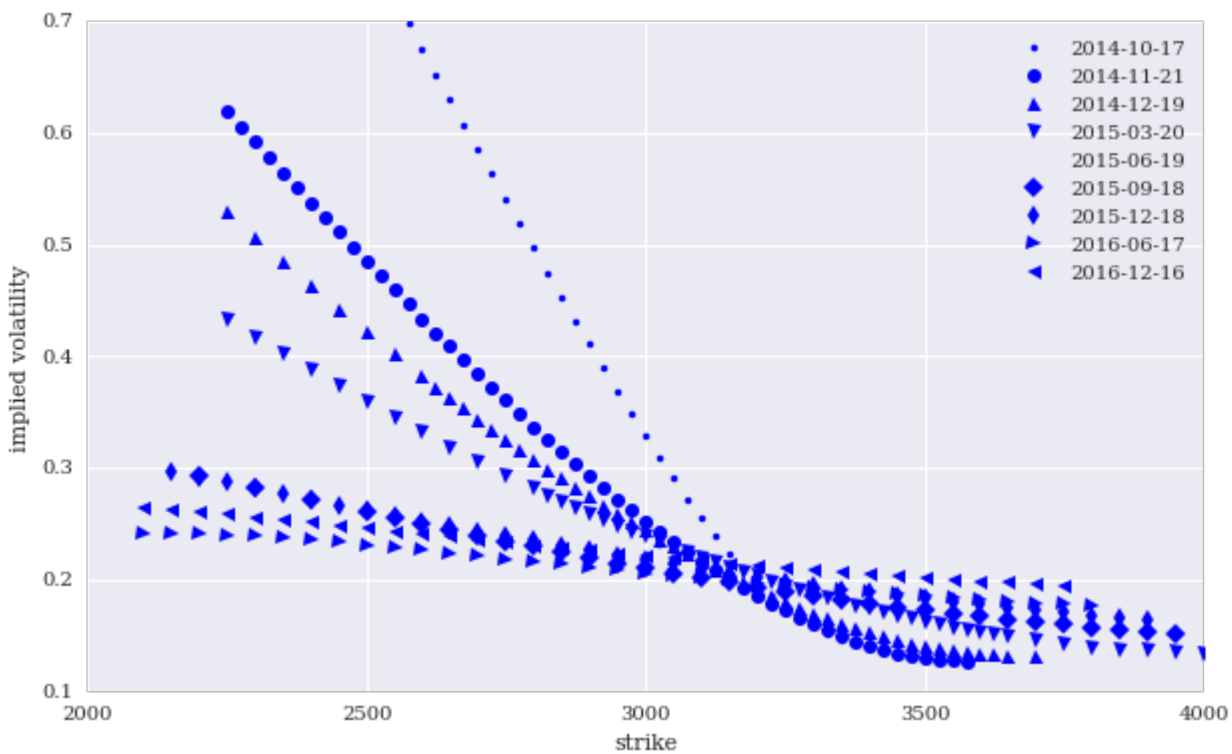
	Date	Strike	Call	Maturity	Put
498	2014-09-30	3750.0	27.4	2015-09-18	635.9
499	2014-09-30	3800.0	21.8	2015-09-18	680.3
500	2014-09-30	3850.0	17.2	2015-09-18	725.7
501	2014-09-30	3900.0	13.4	2015-09-18	772.0
502	2014-09-30	3950.0	10.4	2015-09-18	818.9

The calculation of the **implied volatilities** and the visualization.

In [36]:
```
%time imp_vols = calculate_imp_vols(data)
```

```
CPU times: user 18 s, sys: 7.75 ms, total: 18.1 s
Wall time: 18.1 s
```

In [37]:
```
plot_imp_vols(data)
```



Fourier-based Option Pricing

The Fourier-based option pricing approach has three main advantages:

- **generality**: the approach is applicable whenever the characteristic function of the process driving uncertainty is known; and this is the case for the majority of processes/models applied in practice
- **accuracy**: the semi-analytic formulas can be evaluated numerically in such a way that a high degree of accuracy is reached at little computational cost (e.g. compared to simulation techniques)
- **speed**: the formulas can in general be evaluated very fast such that 10s, 100s or even 1,000s of options can be valued per second

Let us start with a **market model** of the form:

$$\mathcal{M} = \{(\Omega, \mathcal{F}, \mathbb{F}, P), T, (S, B)\}$$

- $(\Omega, \mathcal{F}, \mathbb{F}, P)$ is a filtered probability space
- $T > 0$ is a fixed time horizon
- (S, B) are two traded assets, S a risky one and B a risk-less one

We then know that the **arbitrage value of an attainable European call option** is

$$C_t = e^{-r(T-t)} \mathbf{E}_t^Q (C_T)$$

where $C_T \equiv \max[S_T - K, 0]$ for a strike $K > 0$. In integral from, setting $t = 0$, call option pricing reads

$$\begin{aligned} C_0 &= e^{-rT} \int_0^\infty C_T(s) Q(ds) \\ &= e^{-rT} \int_0^\infty C_T(s) q(s) ds \end{aligned}$$

where $q(s)$ is the risk-neutral probability density function (pdf) of S_T. Unfortunately, the pdf is quite often not known in closed form — whereas the characteristic function (CF) of S_T is.

The fundamental insight of Fourier-based option pricing is to replace both the pdf by the CF and the call option payoff C_T by its Fourier transform.

Let a random variable X be distributed with pdf $q(x)$. The **characteristic function** $\hat{q}$ of X is the Fourier transform of its pdf

$$\hat{q}\left(u\right) \equiv \int_{-\infty}^{\infty} e^{iux} q(x) dx = \mathbf{E}^Q \left(e^{iuX}\right)$$

For $u = u_r + iu_i$ with $u_i > 1$, the **Fourier transform of the European call option payoff** $C_T = \max[S_T - K, 0]$ is given by:

$$\widehat{C}_T(u) = -\frac{K^{iu+1}}{u^2 - iu}$$

Lewis (2001): With φ as the CF of the rv S_T and assuming $u_i \in (0, 1)$, the call option present value is

$$C_0 = S_0 - \frac{Ke^{-rT}}{2\pi} \int_{-\infty+iu_i}^{\infty+iu_i} e^{-iuk} \varphi(-u) \frac{du}{u^2 - ui}$$

Furthermore, setting $u_i = 0.5$ gives

$$C_0 = S_0 - \frac{\sqrt{S_0 K} e^{-rT/2}}{\pi} \int_0^\infty \mathbf{Re}\left[e^{izk} \varphi(z - i/2)\right] \frac{dz}{z^2 + 1/4}$$

where $\mathbf{Re}[x]$ denotes the real part of x.

The Merton (1976) Jump-Diffusion Model

In the Merton (1976) jump-diffusion model, the **risk-neutral index level dynamics** are given by the SDE

$$dS_t = (r - r_J)S_t dt + \sigma S_t dZ_t + J_t S_t dN_t$$

The variables and parameters have the following meaning:

- S_t index level at date t
- r constant risk-less short rate
- $r_J \equiv \lambda \cdot \left(e^{\mu_J + \delta^2/2} - 1\right)$ drift correction for jump
- σ constant volatility of S
- Z_t standard Brownian motion
- J_t jump at date t with distribution $\log(1 + J_t) \approx \mathbf{N}\left(\log(1 + \mu_J) - \frac{\delta^2}{2}, \delta^2\right)$
- $\mathbf{N}$ as the cumulative distribution function of a standard normal random variable
- N_t Poisson process with intensity λ

The **characteristic function for the Merton (1976) model** is given as:

$$\varphi_0^{M76}(u,T) = \exp\left(\left(iu\omega - \frac{u^2\sigma^2}{2} + \lambda\left(e^{iu\mu_J - u^2\delta^2/2} - 1\right)\right)T\right)$$

where the **risk-neutral drift term** ω takes on the form

$$\omega = r - \frac{\sigma^2}{2} - \lambda\left(e^{\mu_J + \delta^2/2} - 1\right)$$

Combining this with the option pricing result from Lewis (2001) we get for the **price of a European call option**

$$C_0 = S_0 - \frac{\sqrt{S_0 K} e^{-rT/2}}{\pi} \int_0^\infty \mathbf{Re} \left[e^{izk} \varphi_0^{M76}(z - i/2, T) \right] \frac{dz}{z^2 + 1/4}$$

Let us implement European call option in Python. First, the **characteristic function**.

In [38]:
```
import math
import numpy as np
from scipy.integrate import quad

def M76_characteristic_function(u, T, r, sigma, lamb, mu, delta):
    omega = r - 0.5 * sigma ** 2 - lamb * (np.exp(mu + 0.5 * delta ** 2) - 1)
    value = np.exp((1j * u * omega - 0.5 * u ** 2 * sigma ** 2 +
            lamb * (np.exp(1j * u * mu - u ** 2 * delta ** 2 * 0.5) - 1))  * T)
    return value
```

Second, the **integration function**.

In [39]:

```
def M76_integration_function(u, S0, K, T, r, sigma, lamb, mu, delta):
    JDCF = M76_characteristic_function(u - 0.5 * 1j, T, r,
                                       sigma, lamb, mu, delta)
    value = 1 / (u ** 2 + 0.25) * (np.exp(1j * u * math.log(S0 / K))
                                    * JDCF).real
    return value
```

Third, the **evaluation of the integral** via numerical quadrature.

In [40]:
```
def M76_value_call_INT(S0, K, T, r, sigma, lamb, mu, delta):
    int_value = quad(lambda u: M76_integration_function(u, S0, K, T, r,
                    sigma, lamb, mu, delta), 0, 50, limit=250)[0]
    call_value = S0 - np.exp(-r * T) * math.sqrt(S0 * K) / math.pi * int_value
    return call_value
```

Fourth, a **numerical example**.

In [41]:
```
S0 = 100.0  # initial index level
K = 100.0  # strike level
T = 1.0  # call option maturity
r = 0.05  # constant short rate
sigma = 0.4  # constant volatility of diffusion
lamb = 1.0  # jump frequency p.a.
mu = -0.2  # expected jump size
delta = 0.1  # jump size volatility
```

In [42]:
```
print "Value of Call Option %8.3f" \
            % M76_value_call_INT(S0, K, T, r, sigma, lamb, mu, delta)
```

```
Value of Call Option   19.948
```

Monte Carlo Simulation

To value a European call option with strike price K by MCS consider the following **discretization of the Merton (1976) SDE**

$$S_t = S_{t-\Delta t}\left(e^{(r-r_J-\sigma^2/2)\Delta t+\sigma\sqrt{\Delta t}z_t^1} + \left(e^{\mu_J+\delta z_t^2}-1\right)y_t\right)$$

with the z_t^n being standard normally distributed and the y_t being Poisson distributed with intensity λ.

The Python code implementing the MCS:

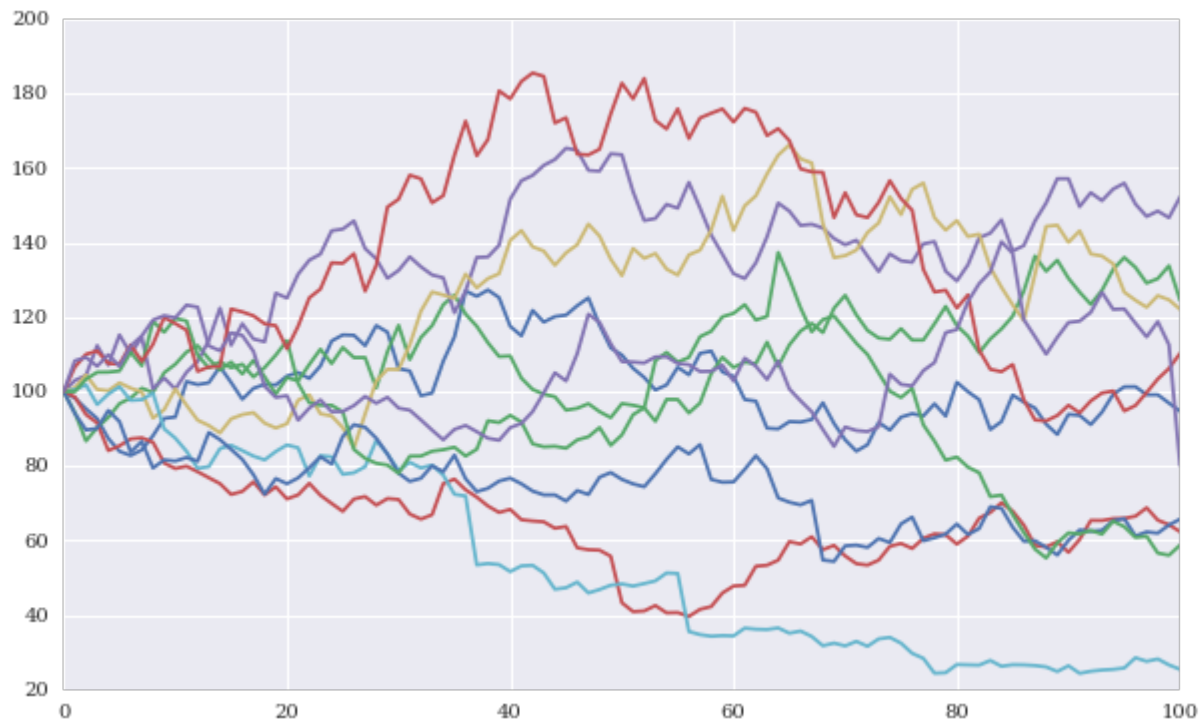
In [43]:
```
def M76_generate_paths(S0, T, r, sigma, lamb, mu, delta, M, I):
    dt = T / M
    rj = lamb * (math.exp(mu + 0.5 * delta ** 2) - 1)
    shape = (M + 1, I)
    S = np.zeros((M + 1, I), dtype=np.float)
    S[0] = S0

    np.random.seed(10000)
    rand1 = np.random.standard_normal(shape)
    rand2 = np.random.standard_normal(shape)
    rand3 = np.random.poisson(lamb * dt, shape)

    for t in xrange(1, M + 1, 1):
        S[t] = S[t - 1] * (np.exp((r - rj - 0.5 * sigma ** 2) * dt
                           + sigma * math.sqrt(dt) * rand1[t])
                           + (np.exp(mu + delta * rand2[t]) - 1)
                           * rand3[t])
    return S
```

The function in action.

In [44]:
```
M = 100  # time steps
I = 10  # paths
S = M76_generate_paths(S0, T, r, sigma, lamb, mu, delta, M, I)
```

The paths visualized.

In [45]:
```
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 6))
plt.plot(S);
```



As simple function to **value a European call option by MCS**.

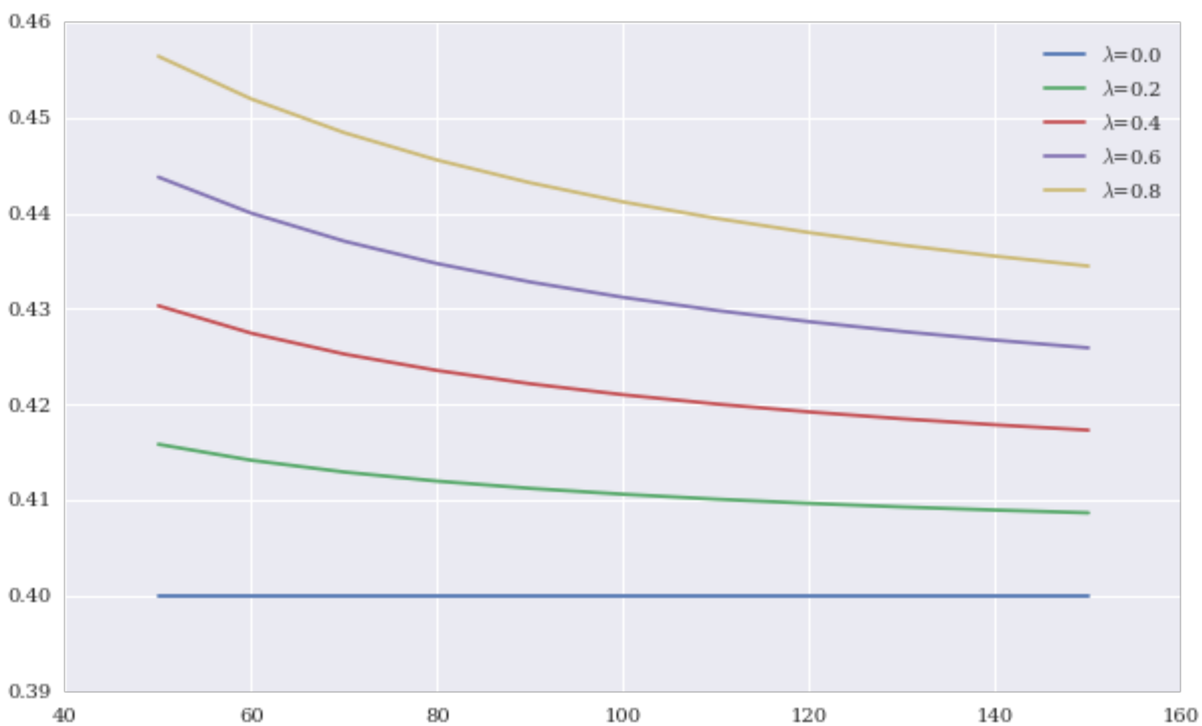
In [46]:
```
def M76_value_call_MCS(K):
    return math.exp(-r * T) * np.sum(np.maximum(S[-1] - K, 0)) / I
```

In [47]:
```
%%time
I = 200000
S = M76_generate_paths(S0, T, r, sigma, lamb, mu, delta, M, I)
print "Value of Call Option %8.3f" % M76_value_call_MCS(K)
```

```
Value of Call Option   19.941
CPU times: user 5.97 s, sys: 840 ms, total: 6.81 s
Wall time: 6.82 s
```

The model of Merton (1976) is capable of generating a **volatility smile**.

In [48]:
```
start = pd.Timestamp('2015-1-1')
end =  pd.Timestamp('2016-1-1')
strikes = range(50, 151, 10)
plt.figure(figsize=(10, 6))
for l in np.arange(0, 1.0, 0.2):
    imp_vols = []
    for k in strikes:
        call = call_option(S0, k, start, end, r, 0.2)
        M76_value = M76_value_call_INT(S0, k, T, r, sigma, l, mu, delta)
        imp_vols.append(call.imp_vol(M76_value))
    plt.plot(strikes, imp_vols, label='$\lambda$=%2.1f' % l)
plt.legend(loc=0); plt.savefig('vol_smile.png')
```



Calibration of the Model

In simple terms, the problem of **calibration** is to find parameters for the Merton (1976) model such that observed market quotes of liquidly traded plain vanilla options are replicated as good as possible. To this end, one defines an error function that is to be minimized. Such a function could be the Root Mean Squared Error (RMSE). The task is then to solve the problem

$$\min_{\sigma,\lambda,\mu_J,\delta} \sqrt{\frac{1}{N}\sum_{n=1}^{N}\left(C_n^* - C_n^{M76}(\sigma,\lambda,\mu_J,\delta)\right)^2}$$

with the C_n^* being the market or input prices and the C_n^{M76} being the model or output prices for the options $n = 1, \ldots, N$.

EXCURSION: The minimization problem is ill-posed (I). Let's analyze properties of the error function for a single European call option.

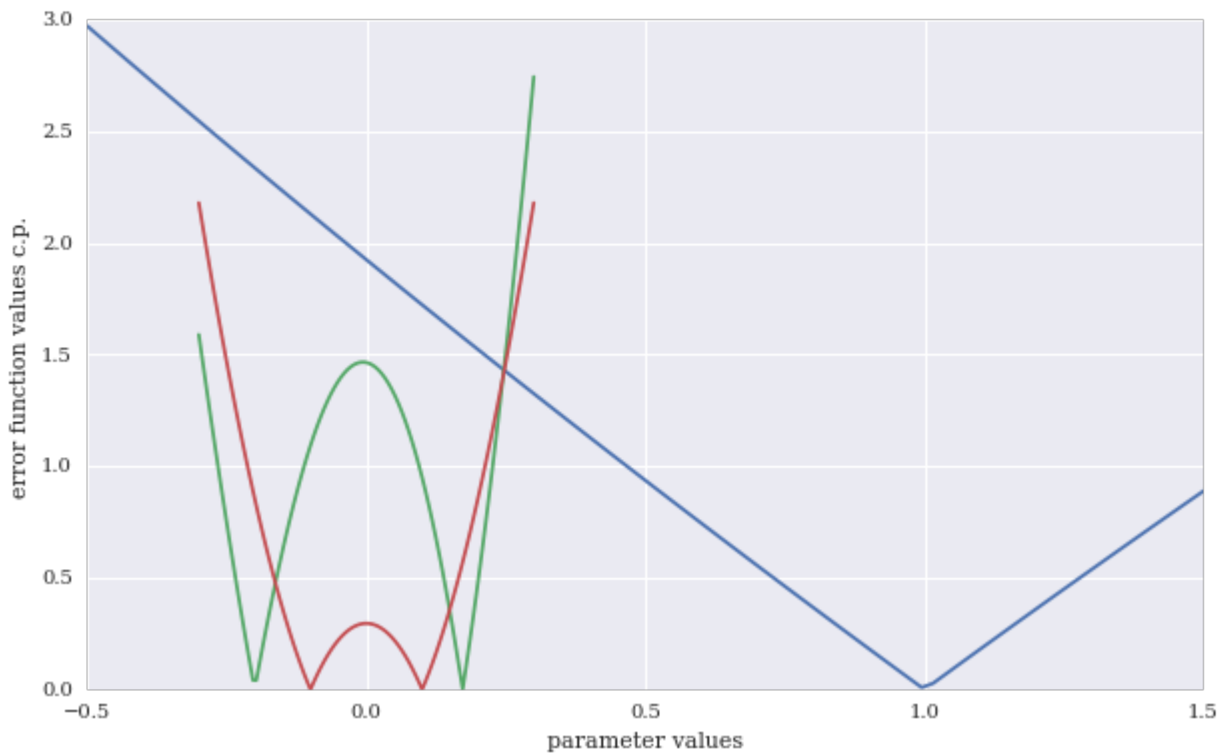
In [49]:
```
C0 = M76_value_call_INT(S0, K, T, r, sigma, lamb, mu, delta)
def error_function(p0):
    sigma, lamb, mu, delta = p0
    return abs(C0 - M76_value_call_INT(S0, K, T, r, sigma, lamb, mu, delta))
```

EXCURSION: The minimization problem is ill-posed (II).

In [50]:
```
def plot_error_function():
    plt.figure(figsize=(10, 6))
    # Plotting (lamb)
    l = np.linspace(-0.5, 1.5, 100); EFv = []
    for i in l:
        EFv.append(error_function([sigma, i, mu, delta]))
    plt.plot(l, EFv)
    plt.xlabel('parameter values')
    plt.ylabel('error function values c.p.')
    # Plotting (mu)
    l = np.linspace(-0.3, 0.3, 100); EFv = []
    for i in l:
        EFv.append(error_function([sigma, lamb, i, delta]))
    plt.plot(l, EFv)
    # Plotting (delta)
    l = np.linspace(-0.3, 0.3, 100); EFv = []
    for i in l:
        EFv.append(error_function([sigma, lamb, mu, i]))
    plt.plot(l, EFv);
```

EXCURSION: The minimization problem is ill-posed (III).

In [51]:
```
plot_error_function()
```



EXCURSION: The simple example illustrates that the calibration of the Merton (1976) jump diffusion model leads to a number of problems:

- **convexity**: the error function is only locally convex
- **determinacy**: the error function exhibits multiple minima
- **degeneracy**: different parameter combinations yield the same result
- **consistency**: the approach mathematically allows parameter values that are economically implausible
- **stability**: slight changes in the input values can change the solution significantly ('sudden' change from one local minimum to another is possible)

Let us import some **real option quotes for European call options on the EURO STOXX 50 index**.

In [52]:
```
import pandas as pd
h5 = pd.HDFStore('es50_option_data.h5', 'r')
data = h5['data']  # European call & put option data (3 maturities)
h5.close()
S0 = 3225.93  # EURO STOXX 50 level
r = 0.005  # assumption

# Option Selection
tol = 0.05
options = data[(np.abs(data['Strike'] - S0) / S0) < tol]
mats = sorted(set(options['Maturity']))
options = options[options['Maturity'] == mats[0]]
```

These are the **option quotes** we are dealing with (I).

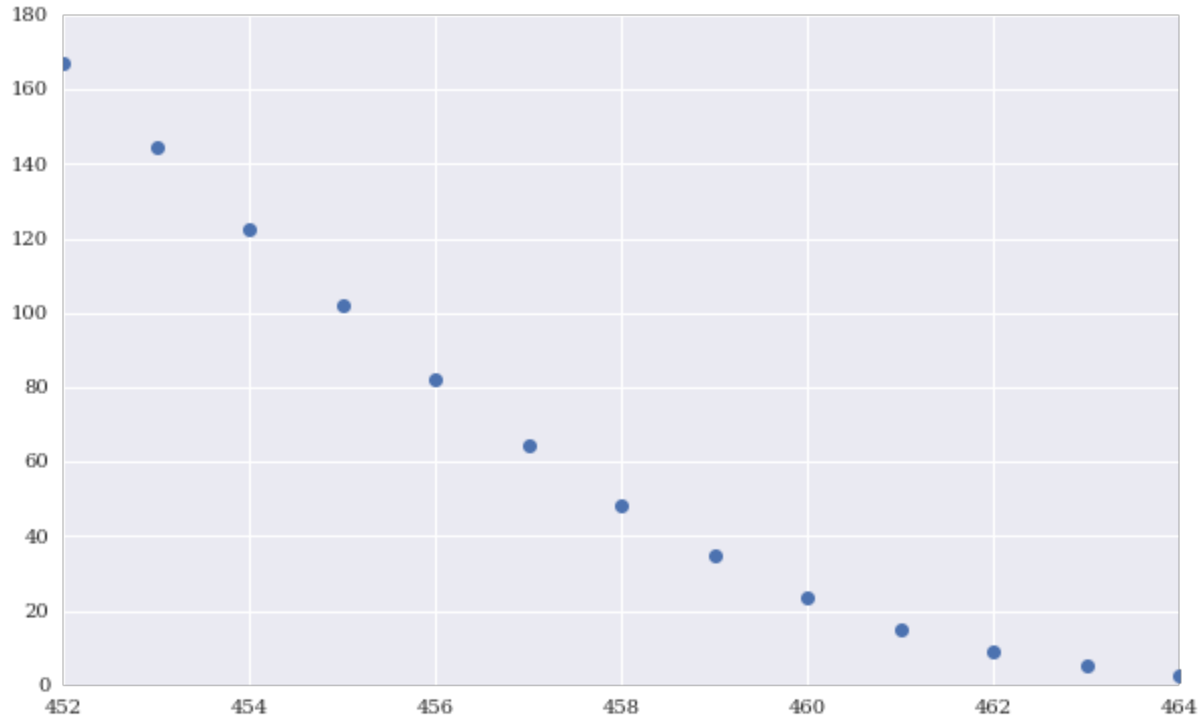
In [53]:
```
options
```

Out[53]:

	Date	Strike	Call	Maturity	Put
452	2014-09-30	3075.0	167.0	2014-10-17	9.3
453	2014-09-30	3100.0	144.5	2014-10-17	11.7
454	2014-09-30	3125.0	122.7	2014-10-17	14.9
455	2014-09-30	3150.0	101.8	2014-10-17	19.1
456	2014-09-30	3175.0	82.3	2014-10-17	24.5
457	2014-09-30	3200.0	64.3	2014-10-17	31.5
458	2014-09-30	3225.0	48.3	2014-10-17	40.5
459	2014-09-30	3250.0	34.6	2014-10-17	51.8
460	2014-09-30	3275.0	23.5	2014-10-17	65.8
461	2014-09-30	3300.0	15.1	2014-10-17	82.3
462	2014-09-30	3325.0	9.1	2014-10-17	101.3
463	2014-09-30	3350.0	5.1	2014-10-17	122.4
464	2014-09-30	3375.0	2.8	2014-10-17	145.0

These are the **option quotes** we are dealing with (II).

In [54]:
```
options['Call'].plot(style='o', figsize=(10, 6));
```



Next, we define an **error function** in Python for the calibration.

In [55]:

```
i = 0; min_RMSE = 100.
def M76_error_function(p0):
    global i, min_RMSE
    sigma, lamb, mu, delta = p0
    if sigma < 0.0 or delta < 0.0 or lamb < 0.0:
        return 500.0
    se = []
    for row, option in options.iterrows():
        T = (option['Maturity'] - option['Date']).days / 365.
        model_value = M76_value_call_INT(S0, option['Strike'], T,
                                         r, sigma, lamb, mu, delta)
        se.append((model_value - option['Call']) ** 2)
    RMSE = math.sqrt(sum(se) / len(se))
    min_RMSE = min(min_RMSE, RMSE)
    if i % 100 == 0:
        print '%4d |' % i, np.array(p0), '| %7.3f | %7.3f' % (RMSE, min_RMSE)
    i += 1
    return RMSE
```

The calibration is done in two steps. First, a **global optimization**.

In [56]:
```
%%time
import scipy.optimize as sop
np.set_printoptions(suppress=True,
                    formatter={'all': lambda x: '%6.3f' % x})
p0 = sop.brute(M76_error_function, ((0.10, 0.201, 0.025),
                   (0.10, 0.80, 0.10), (-0.40, 0.01, 0.10),
                   (0.00, 0.121, 0.02)), finish=None)
```

```
   0 | [ 0.100  0.100 -0.400  0.000] |  12.676 |  12.676
 100 | [ 0.100  0.300  0.000  0.020] |  15.240 |   7.372
 200 | [ 0.100  0.600 -0.100  0.060] |  11.777 |   2.879
 300 | [ 0.125  0.200 -0.200  0.100] |   9.004 |   2.443
 400 | [ 0.125  0.500 -0.200  0.000] |   5.056 |   1.125
 500 | [ 0.150  0.100 -0.300  0.040] |   6.109 |   0.970
 600 | [ 0.150  0.400 -0.400  0.080] |   4.135 |   0.970
 700 | [ 0.150  0.600  0.000  0.120] |   6.333 |   0.970
 800 | [ 0.175  0.200  0.000  0.020] |   5.955 |   0.970
 900 | [ 0.175  0.500 -0.100  0.060] |   5.536 |   0.970
1000 | [ 0.200  0.100 -0.200  0.100] |   8.596 |   0.970
1100 | [ 0.200  0.400 -0.200  0.000] |  10.692 |   0.970
1200 | [ 0.200  0.700 -0.300  0.040] |  17.578 |   0.970
CPU times: user 1min 22s, sys: 234 ms, total: 1min 22s
Wall time: 1min 22s
```

Second, the **local (convex) optimization**.

In [57]:
```
%%time
opt = sop.fmin(M76_error_function, p0, xtol=0.00001,
                    ftol=0.00001, maxiter=750, maxfun=1500)
```

```
1300 | [ 0.122  0.723 -0.247  0.110] |   0.798 |   0.798
1400 | [ 0.119  1.001 -0.187  0.013] |   0.769 |   0.768
1500 | [ 0.119  1.031 -0.183  0.000] |   0.767 |   0.767
1600 | [ 0.119  1.033 -0.183  0.000] |   0.767 |   0.767
Optimization terminated successfully.
         Current function value: 0.766508
         Iterations: 278
         Function evaluations: 477
CPU times: user 26.1 s, sys: 104 ms, total: 26.2 s
Wall time: 26.1 s
```

The **optimal parameter values** are:

In [58]:
```
i = 0
M76_error_function(p0)
```

```
   0 | [ 0.125  0.600 -0.300  0.120] |   0.970 |   0.767
```

Out[58]:
```
0.9695219560421178
```

In [59]:
```
i = 0
M76_error_function(opt)
```

```
   0 | [ 0.119  1.033 -0.183  0.000] |   0.767 |   0.767
```

Out[59]:
```
0.7665083562548708
```

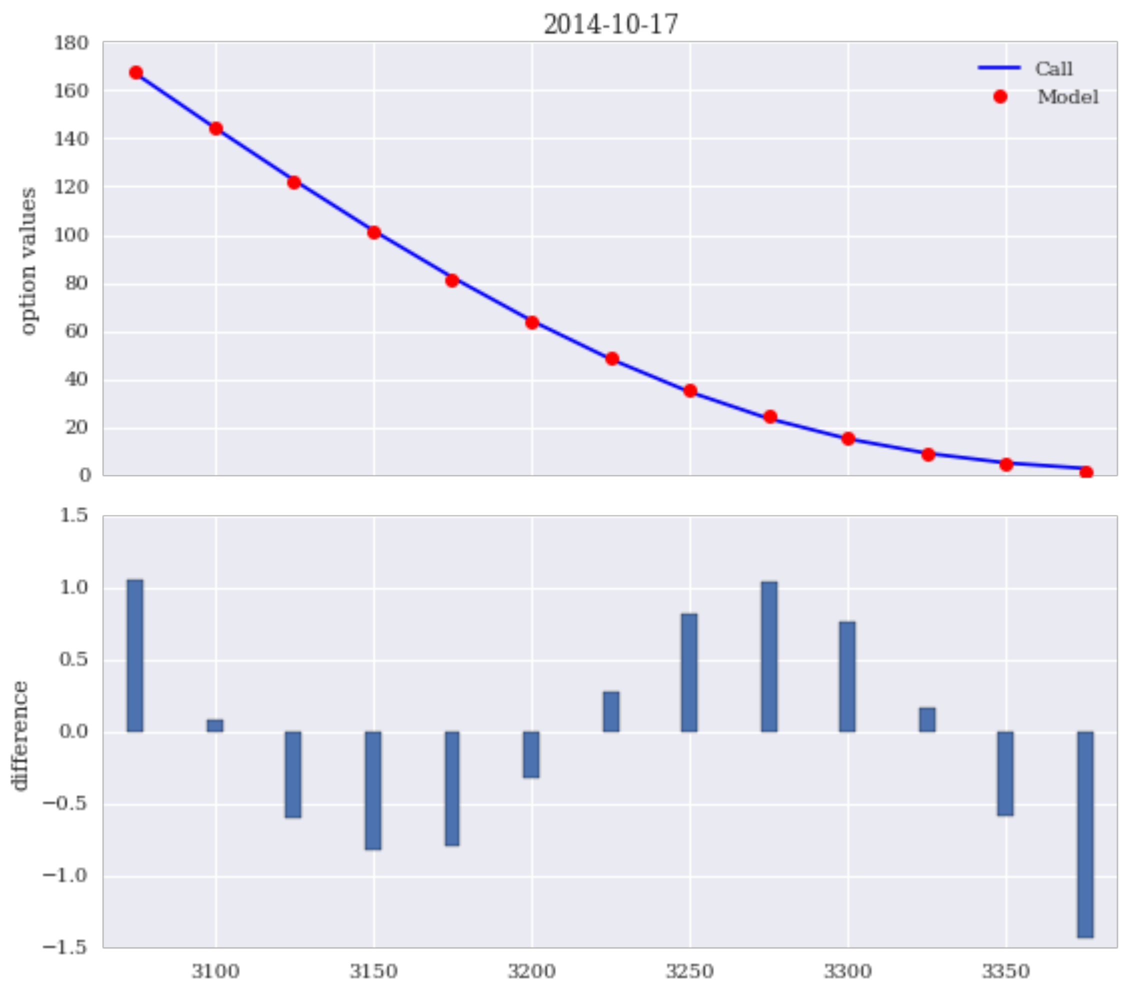
Comparison of market and model prices (I).

In [60]:

```
def generate_plot(opt, options):
    sigma, lamb, mu, delta = opt
    options['Model'] = 0.0
    for row, option in options.iterrows():
        T = (option['Maturity'] - option['Date']).days / 365.
        options.loc[row, 'Model'] = M76_value_call_INT(S0, option['Strike'],
                                    T, r, sigma, lamb, mu, delta)
    options = options.set_index('Strike')
    fig, ax = plt.subplots(2, sharex=True, figsize=(8, 7))
    options[['Call', 'Model']].plot(style=['b-', 'ro'],
                    title='%s' % str(option['Maturity'])[:10], ax=ax[0])
    ax[0].set_ylabel('option values')
    xv = options.index.values
    ax[1] = plt.bar(xv - 5 / 2., options['Model'] - options['Call'],
                    width=5)
    plt.ylabel('difference')
    plt.xlim(min(xv) - 10, max(xv) + 10)
    plt.tight_layout()
```

Comparison of market and model prices (II).

In [61]:
```
generate_plot(opt, options)
```



Finally, a look at the implied volatilities.

In [62]:
```
S0 = 3225.93; r = 0.005
def calc_imp_vols(data):
    data['Imp_Vol_Mod'] = 0.0
    data['Imp_Vol_Mar'] = 0.0
    tol = 0.30  # tolerance for moneyness
    for row in data.index:
        t = data['Date'][row]
        T = data['Maturity'][row]
        ttm = (T - t).days / 365.
        forward = np.exp(r * ttm) * S0
        if (abs(data['Strike'][row] - forward) / forward) < tol:
            call = call_option(S0, data['Strike'][row], t, T, r, 0.2)
            data['Imp_Vol_Mod'][row] = call.imp_vol(data['Model'][row])
            data['Imp_Vol_Mar'][row] = call.imp_vol(data['Call'][row])
    return data
```

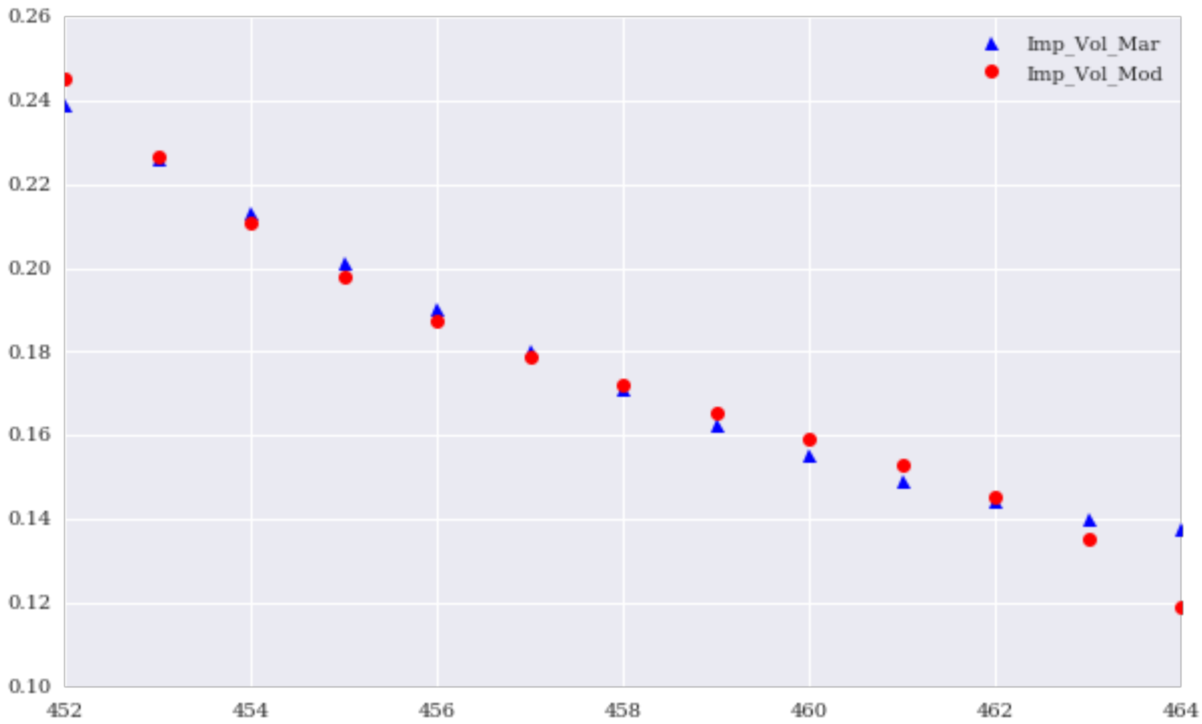
The calculation of the **model implied volatilities** and a comparison.

In [63]:

```
options = calc_imp_vols(options)
options[['Imp_Vol_Mar', 'Imp_Vol_Mod']].plot(figsize=(10, 6), style=['b^', 'ro'])
```

Out[63]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fc05edb2390>
```



Conclusions

In conclusion, we can state the following:

- **markets**: time series in financial markets strongly deviate from the Gaussian benchmark(s); there are generally eg stochastic volatility, jumps, implied volatility smiles observed in historical data
- **Merton (1976) model**: the model of Merton is capable of accounting for some observed stylized facts like jumps and volatility smiles
- **calibration issues**: numerical finance and optimization (i.e. calibration) faces a number of issues, eg with regard the determinacy of solutions and convexity of error functions
- **Python**: Python is really close to mathematical and financial syntax; the implementation of financial algorithms generally is efficient and performant (when using the right libraries and idioms like NumPy with vectorization)

All details, codes, proofs, etc. in the book "Derivatives Analytics with Python" — cf. http://derivatives-analytics-with-python.com (http://derivatives-analytics-with-python.com).

http://tpq.io (http://tpq.io) | @dyjh (http://twitter.com/dyjh) | team@tpq.io (mailto:team@tpq.io)

Quant Platform | http://quant-platform.com (http://quant-platform.com)

Python for Finance | Python for Finance @ O'Reilly (http://python-for-finance.com)

Derivatives Analytics with Python | Derivatives Analytics @ Wiley Finance (http://derivatives-analytics-with-python.com)