CHAPTER 5
Fractionally Differentiated Features

5.1 MOTIVATION

It is known that, as a consequence of arbitrage forces, financial series exhibit low signal-to-noise ratios (López de Prado [2015]). To make matters worse, standard stationarity transformations, like integer differentiation, further reduce that signal by removing memory. Price series are often non-stationary, and often have memory. In contrast, integer differentiated series, like returns, have a memory cut-off, in the sense that history is disregarded entirely after a finite sample window. Once stationarity transformations have wiped out all memory from the data, statisticians resort to complex mathematical techniques to extract whatever residual signal remains. Not surprisingly, applying these complex techniques on memory-erased series likely leads to false discoveries. In this chapter we introduce a data transformation method that ensures the stationarity of the data while preserving as much memory as possible.

5.2 THE STATIONARITY VS. MEMORY DILEMMA

Intuitively, a time series is stationary when its statistical properties are invariant by change of the origin of time. A time series has memory when future values are related to past observations. In order to perform inferential analyses, researchers need to work with invariant processes, such as returns on prices (or changes in log-prices), changes in yield, or changes in volatility. Invariance is often achieved via data transformations that make the series stationary, at the expense of removing all memory from the original series (Alexander [2001], chapter 11). Although stationarity is a necessary property for inferential purposes, it is rarely the case in signal processing that we wish all memory to be erased, as that memory is the basis for the model’s predictive power. For example, equilibrium (stationary) models need some memory to assess how far the price process has drifted away from the long-term expected value in order to generate a forecast. The dilemma is that returns are stationary, however memory-less, and prices have memory, however they are non-stationary. The question arises: What is the minimum amount of differentiation that makes a price series stationary while preserving as much memory as possible? Accordingly, we would like to generalize the notion of returns to consider *stationary series where not all memory is erased.* Under this framework, returns are just one kind of (and in most cases suboptimal) price transformation among many other possibilities.

Part of the importance of cointegration methods is their ability to model series with memory. But why would the particular case of zero differentiation deliver best outcomes? Zero differentiation is as arbitrary as 1-step differentiation. There is a wide region between these two extremes (fully differentiated series on one hand, and zero differentiated series on the other) that can be explored through fractional differentiation for the purpose of developing a highly predictive ML model.

Supervised learning algorithms typically require stationary features. The reason is that we need to map a previously unseen (unlabeled) observation to a collection of labeled examples, and infer from them the label of that new observation. If the features are not stationary, we cannot map the new observation to a large number of mutually comparable examples. But stationarity does not ensure predictive power. Stationarity is a necessary, non-sufficient condition for the high performance of an ML algorithm. The problem is, differentiation imposes a trade-off between stationarity and memory. We can always make a series more stationary through differentiation, but it will be at the cost of erasing some memory, which will defeat the forecasting purpose of the ML algorithm. In this chapter, we will study one way to resolve this dilemma.

5.3 LITERATURE REVIEW

Virtually all the financial time series literature is based on the premise of making non-stationary series stationary through integer transformation (see Hamilton [1994] for an example). This raises two questions: (1) Why would integer 1 differentiation (like the one used for computing returns on log-prices) be optimal? (2) Is over-differentiation one reason why the literature has been so biased in favor of the efficient markets hypothesis?

The notion of fractional differentiation applied to the predictive time series analysis dates back at least to Hosking [1981]. In that paper, a family of ARIMA processes was generalized by permitting the degree of differencing to take fractional values. This was useful because fractionally differenced processes exhibit long-term persistence and antipersistence, hence enhancing the forecasting power compared to the standard ARIMA approach. In the same paper, Hosking states: “Apart from a passing reference by Granger (1978), fractional differencing does not appear to have been previously mentioned in connection with time series analysis.”

After Hosking’s paper, the literature on this subject has been surprisingly scarce, adding up to eight journal articles written by only nine authors: Hosking, Johansen, Nielsen, MacKinnon, Jensen, Jones, Popiel, Cavaliere, and Taylor. See the references for details. Most of those papers relate to technical matters, such as fast algorithms for the calculation of fractional differentiation in continuous stochastic processes (e.g., Jensen and Nielsen [2014]).

Differentiating the stochastic process is a computationally expensive operation. In this chapter we will take a practical, alternative, and novel approach to recover stationarity: We will generalize the difference operator to non-integer steps.

5.4 THE METHOD

Consider the backshift operator, B, applied to a matrix of real-valued features $\{X_t\}$, where $B^k X_t = X_{t-k}$ for any integer $k \geq 0$. For example, $(1 - B)^2 = 1 - 2B + B^2$, where $B^2 X_t = X_{t-2}$, so that $(1 - B)^2 X_t = X_t - 2X_{t-1} + X_{t-2}$. Note that $(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k} = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$, for n a positive integer. For a real number d, $(1+x)^d = \sum_{k=0}^{\infty} \binom{d}{k} x^k$, the binomial series.

In a fractional model, the exponent d is allowed to be a real number, with the following formal binomial series expansion:

$$
\begin{aligned}
(1-B)^d &= \sum_{k=0}^{\infty} \binom{d}{k} (-B)^k = \sum_{k=0}^{\infty} \frac{\prod_{i=0}^{k-1}(d-i)}{k!} (-B)^k \\
&= \sum_{k=0}^{\infty} (-B)^k \prod_{i=0}^{k-1} \frac{d-i}{k-i} \\
&= 1 - dB + \frac{d(d-1)}{2!} B^2 - \frac{d(d-1)(d-2)}{3!} B^3 + \cdots
\end{aligned}
$$

5.4.1 Long Memory

Let us see how a real (non-integer) positive *d* preserves memory. This arithmetic series consists of a dot product

$$\tilde{X}_t = \sum_{k=0}^{\infty} \omega_k X_{t-k}$$

with weights ω

$$\omega = \left\{ 1, -d, \frac{d(d-1)}{2!}, -\frac{d(d-1)(d-2)}{3!}, \dots, (-1)^k \prod_{i=0}^{k-1} \frac{d-i}{k!}, \dots \right\}$$

and values X

$$X = \{X_t, X_{t-1}, X_{t-2}, X_{t-3}, \dots, X_{t-k}, \dots\}$$

When *d* is a positive integer number, $\prod_{i=0}^{k-1} \frac{d-i}{k!} = 0, \forall k > d$, and memory beyond that point is cancelled. For example, *d* = 1 is used to compute returns, where $\prod_{i=0}^{k-1} \frac{d-i}{k!} = 0, \forall k > 1$, and $\omega = \{1, -1, 0, 0, \dots\}$.

5.4.2 Iterative Estimation

Looking at the sequence of weights, ω, we can appreciate that for $k = 0, \dots, \infty$, with $\omega_0 = 1$, the weights can be generated iteratively as:

$$\omega_k = -\omega_{k-1} \frac{d-k+1}{k}$$

Figure 5.1 plots the sequence of weights used to compute each value of the fractionally differentiated series. The legend reports the value of *d* used to generate each sequence, the x-axis indicates the value of *k*, and the y-axis shows the value of ω_k. For example, for $d = 0$, all weights are 0 except for $\omega_0 = 1$. That is the case where the differentiated series coincides with the original one. For $d = 1$, all weights are 0 except for $\omega_0 = 1$ and $\omega_1 = -1$. That is the standard first-order integer differentiation, which is used to derive log-price returns. Anywhere in between these two cases, all weights after $\omega_0 = 1$ are negative and greater than -1.

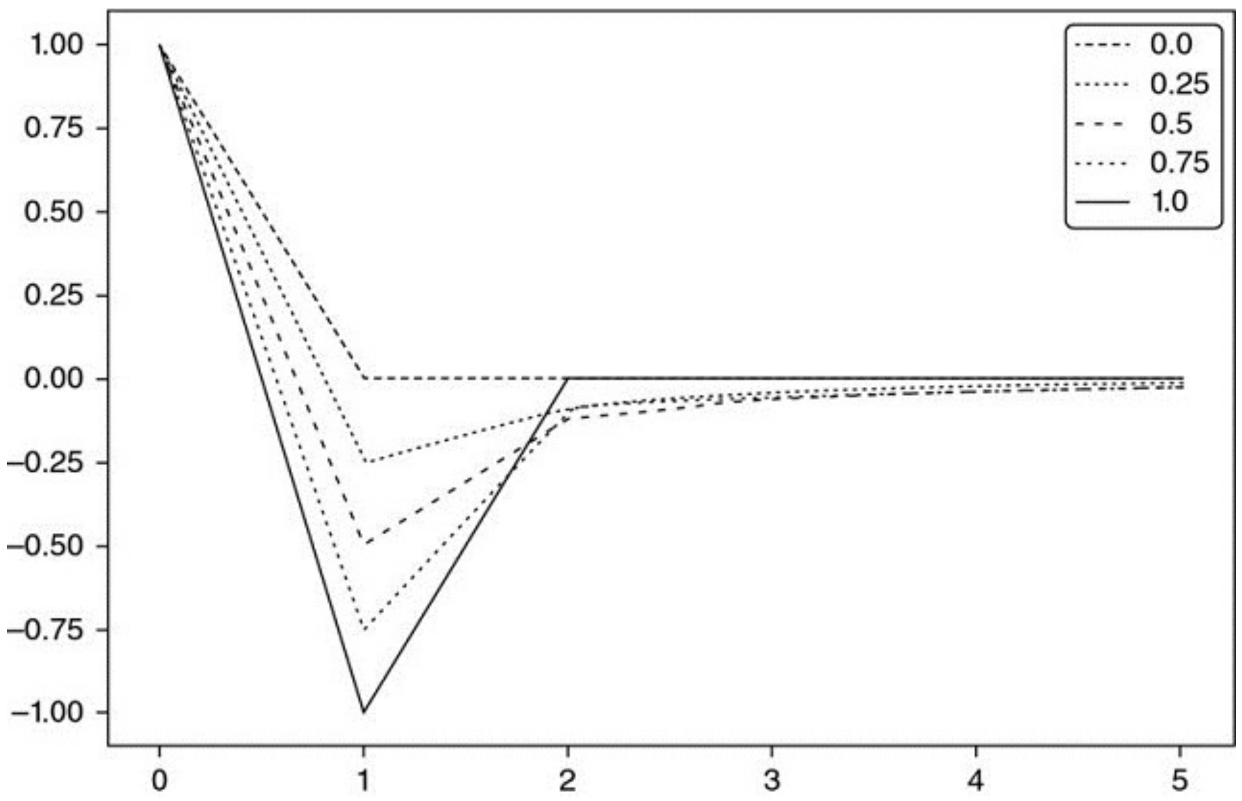


FIGURE 5.1 ω_k (y-axis) as k increases (x-axis). Each line is associated with a particular value of $d \in [0,1]$, in 0.1 increments.

Figure 5.2 plots the sequence of weights where $d \in [1, 2]$, at increments of 0.1. For $d > 1$, we observe $\omega_1 < -1$ and $\omega_k > 0, \forall k \geq 2$.

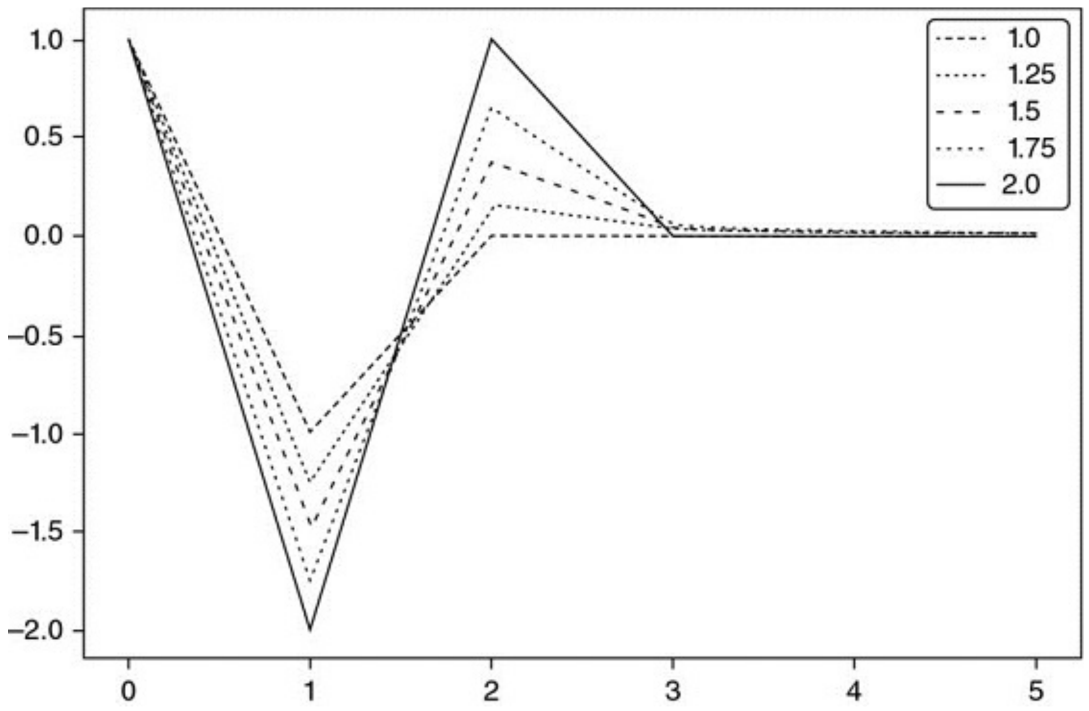


FIGURE 5.2 ω_k (y-axis) as k increases (x-axis). Each line is associated with a particular value of $d \in [1,2]$, in 0.1 increments.

Snippet 5.1 lists the code used to generate these plots.

SNIPPET 5.1 WEIGHTING FUNCTION

```
def getWeights(d,size):
    # thres>0 drops insignificant weights
    w=[1.]
    for k in range(1,size):
        w_=-w[-1]/k*(d-k+1)
        w.append(w_)
    w=np.array(w[::-1]).reshape(-1,1)
    return w
#———————————————————————————————————————-
def plotWeights(dRange,nPlots,size):
    w=pd.DataFrame()
    for d in np.linspace(dRange[0],dRange[1],nPlots):
        w_=getWeights(d,size=size)
        w_=pd.DataFrame(w_,index=range(w_.shape[0])[::-1],columns=[d])
        w=w.join(w_,how='outer')
    ax=w.plot()
    ax.legend(loc='upper left');mpl.show()
    return
#———————————————————————————————————————-
if __name__=='__main__':
    plotWeights(dRange=[0,1],nPlots=11,size=6)
    plotWeights(dRange=[1,2],nPlots=11,size=6)
```

5.4.3 Convergence

Let us consider the convergence of the weights. From the above result, we can see that for $k > d$, if $\omega_{k-1} \neq 0$, then $\left|\frac{\omega_k}{\omega_{k-1}}\right| = \left|\frac{d-k+1}{k}\right| < 1$, and $\omega_k = 0$ otherwise. Consequently, the weights converge asymptotically to zero, as an infinite product of factors within the unit circle. Also, for a positive d and $k < d + 1$, we have $\frac{d-k+1}{k} \geq 0$, which makes the initial weights alternate in sign. For a non-integer d, once $k \geq d + 1$, ω_k will be negative if int[d] is even, and positive otherwise. Summarizing, $\lim_{k\to\infty} \omega_k = 0^-$ (converges to zero from the left) when int[d] is even, and $\lim_{k\to\infty} \omega_k = 0^+$ (converges to zero from the right) when Int[d] is odd. In the special case $d \in (0, 1)$, this means that $-1 < \omega_k < 0, \forall k > 0$. This alternation of weight signs is necessary to make $\{\tilde{X}_t\}_{t=1,\ldots,T}$ stationary, as memory wanes or is offset over the long run.

5.5 IMPLEMENTATION

In this section we will explore two alternative implementations of fractional differentiation: the standard “expanding window” method, and a new method that I call “fixed-width window fracdiff” (FFD).

5.5.1 Expanding Window

Let us discuss how to fractionally differentiate a (finite) time series in practice. Suppose a time series with T real observations, $\{X_t\}$, $t = 1,\dots, T$. Because of data limitations, the fractionally differentiated value $\tilde{X}_T$ cannot be computed on an infinite series of weights. For instance, the last point $\tilde{X}_T$ will use weights $\{\omega_k\}$, $k = 0, \dots, T - 1$, and $\tilde{X}_{T-l}$ will use weights $\{\omega_k\}$, $k = 0, \dots, T - l - 1$. This means that the initial points will have a different amount of memory compared to the final points. For each l, we can determine the relative weight-loss, $\lambda_l = \frac{\sum_{j=T-l}^{T} |\omega_j|}{\sum_{i=0}^{T-1} |\omega_i|}$. Given a tolerance level $\tau \in [0, 1]$, we can determine the value l^* such that $\lambda_{l^*} \leq \tau$ and $\lambda_{l^*+1} > \tau$. This value l^* corresponds to the first results $\{\tilde{X}_t\}_{t=1,\dots,l^*}$ where the weight-loss is beyond the acceptable threshold, $\lambda_t > \tau$ (e.g., $\tau = 0.01$).

From our earlier discussion, it is clear that λ_{l^*} depends on the convergence speed of $\{\omega_k\}$, which in turn depends on $d \in [0, 1]$. For $d = 1$, $\omega_k = 0$, $\forall k > 1$, and $\lambda_l = 0$, $\forall l > 1$, hence it suffices to drop $\tilde{X}_1$. As $d \to 0^+$, l^* increases, and a larger portion of the initial $\{\tilde{X}_t\}_{t=1,\dots,l^*}$ needs to be dropped in order to keep the weight-loss $\lambda_{l^*} \leq \tau$. Figure 5.3 plots the E-mini S&P 500 futures trade bars of size 1E4, rolled forward, fractionally differentiated, with parameters ($d = .4$, $\tau = 1$) on the top and parameters ($d = .4$, $\tau = 1E - 2$) on the bottom.

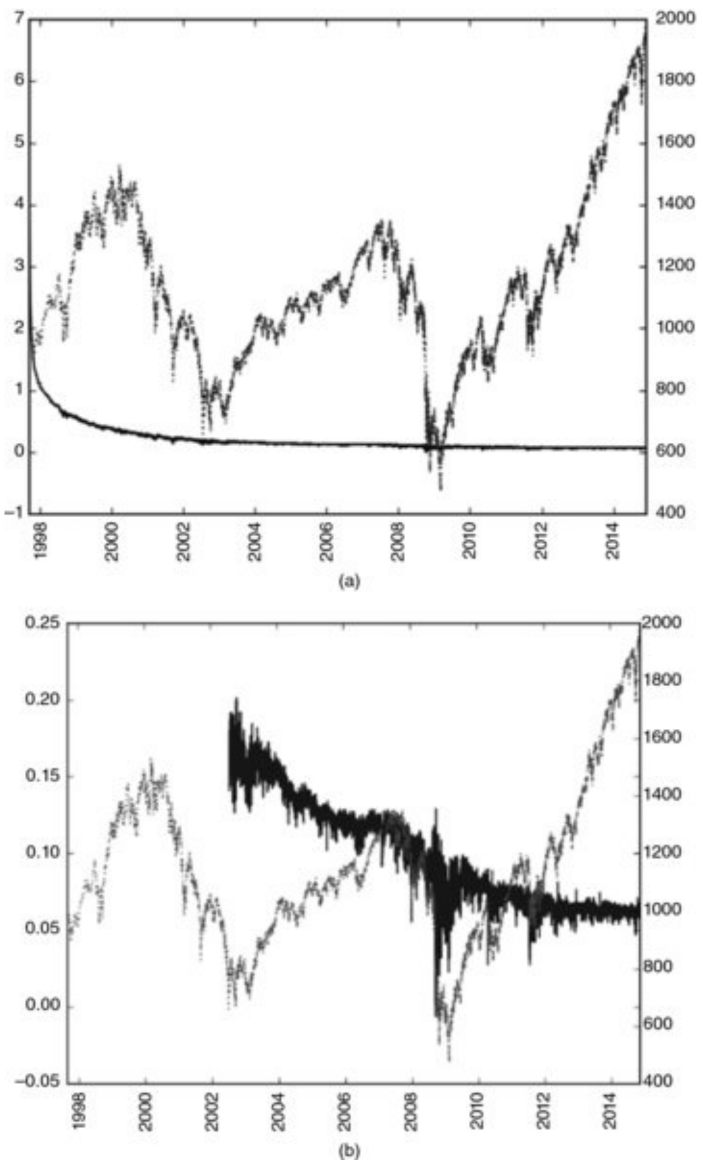


FIGURE 5.3 Fractional differentiation without controlling for weight loss (top plot) and after controlling for weight loss with an expanding window (bottom plot)

The negative drift in both plots is caused by the negative weights that are added to the initial observations as the window is expanded. When we do not control for weight loss, the negative drift is extreme, to the point that only that trend is visible. The negative drift is somewhat more moderate in the right plot, after controlling for the weight loss, however, it is still substantial, because values $\{\tilde{X}_t\}_{t=l^*+1,\dots,T}$ are computed on an expanding window. This problem can be corrected by a fixed-width window, implemented in Snippet 5.2.

SNIPPET 5.2 STANDARD FRACDIFF (EXPANDING WINDOW)

```
def fracDiff(series,d,thres=.01):
    '''
    Increasing width window, with treatment of NaNs
    Note 1: For thres=1, nothing is skipped.
    Note 2: d can be any positive fractional, not necessarily bounded [0,1].
    '''
    #1) Compute weights for the longest series
    w=getWeights(d,series.shape[0])
    #2) Determine initial calcs to be skipped based on weight-loss threshold
    w_=np.cumsum(abs(w))
    w_/=w_[-1]
    skip=w_[w_>thres].shape[0]
    #3) Apply weights to values
    df={}
    for name in series.columns:
        seriesF,df_=series[[name]].fillna(method='ffill').dropna(),pd.Series()
        for iloc in range(skip,seriesF.shape[0]):
            loc=seriesF.index[iloc]
            if not np.isfinite(series.loc[loc,name]):continue # exclude NAs
            df_[loc]=np.dot(w[-(iloc+1):,:].T,seriesF.loc[:loc])[0,0]
        df[name]=df_.copy(deep=True)
    df=pd.concat(df,axis=1)
    return df
```

5.5.2 Fixed-Width Window Fracdiff

Alternatively, fractional differentiation can be computed using a fixed-width window, that is, dropping the weights after their modulus ($|\omega_k|$) falls below a given threshold value (τ). This is equivalent to finding the first l^* such that $|\omega_{l^*}| \geq \tau$ and $|\omega_{l^*+1}| \leq \tau$, setting a new variable $\tilde{\omega}_k$

$$\tilde{\omega}_k = \begin{cases} \omega_k & \text{if } k \leq l^* \\ 0 & \text{if } k > l^* \end{cases}$$

and $\tilde{X}_t = \sum_{k=0}^{l^*} \tilde{\omega}_k X_{t-k}$, for $t = T - l^* + 1, \ldots, T$. Figure 5.4 plots E-mini S&P 500 futures trade bars of size 1E4, rolled forward, fractionally differentiated ($d = .4$, $\tau = 1E - 5$).

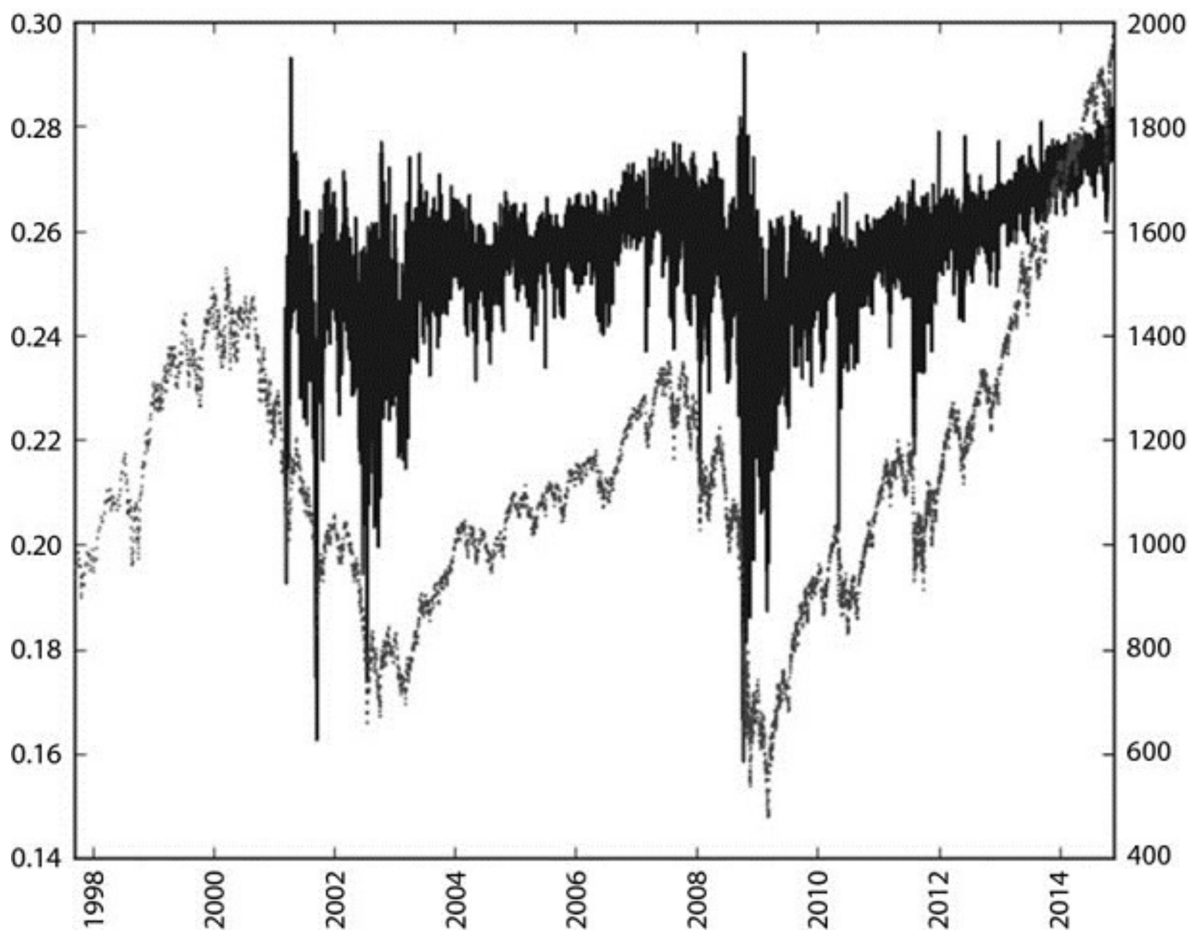


FIGURE 5.4 Fractional differentiation after controlling for weight loss with a fixed-width window

This procedure has the advantage that the same vector of weights is used across all estimates of $\{\tilde{X}_t\}_{t=l^*,\dots,T}$, hence avoiding the negative drift caused by an expanding window's added weights. The result is a driftless blend of level plus noise, as expected. The distribution is no longer Gaussian, and observations exhibit positive serial correlation, however now the sample passes the ADF test (we reject the null hypothesis that a unit root is present in the time series sample). Snippet 5.3 presents an implementation of this idea.

SNIPPET 5.3 THE NEW FIXED-WIDTH WINDOW FRACDIFF METHOD

```
def getWeights_FFD(d,thres):
    # thres>0 drops insignificant weights
    w,k=[1.],1
    while True:
        w_=-w[-1]/k*(d-k+1)
        if abs(w_)<thres:break
        w.append(w_)
        k+=1
    w=np.array(w[::-1]).reshape(-1,1)
    return w
#----------------------------------------------------------------
------------
def fracDiff_FFD(series,d,thres=1e-5):
    '''
    Constant width window (new solution)
    Note 1: thres determines the cut-off weight for the window
    Note 2: d can be any positive fractional, not necessarily
bounded [0,1].
    '''
    #1) Compute weights for the longest series
    w=getWeights_FFD(d,thres)
    width=len(w)-1
    #2) Apply weights to values
    df={}
    for name in series.columns:
        seriesF =series[[name]].fillna(method='ffill').dropna()
        df_=pd.Series()
        for iloc1 in range(width,seriesF.shape[0]):
            loc0,loc1=seriesF.index[iloc1-
width],seriesF.index[iloc1]
            if not np.isfinite(series.loc[loc1,name]):continue #
exclude NAs
            df_[loc1]=np.dot(w.T,seriesF.loc[loc0:loc1])[0,0]
        df[name]=df_.copy(deep=True)
    df=pd.concat(df,axis=1)
    return df
```

5.6 STATIONARITY WITH MAXIMUM MEMORY PRESERVATION

Consider a series $\{X_t\}_{t=1,\dots,T}$. Applying the fixed-width window fracdiff (FFD) method on this series, we can compute the minimum coefficient *d** such that the resulting fractionally differentiated series $\{\tilde{X}_t\}_{t=l^*,\dots,T}$ is stationary. This coefficient *d** quantifies the amount of memory that needs to be removed to achieve stationarity. If $\{X_t\}_{t=l^*,\dots,T}$ is already stationary, then $d^* = 0$. If $\{X_t\}_{t=l^*,\dots,T}$ contains a unit root, then $d^* < 1$. If $\{X_t\}_{t=l^*,\dots,T}$ exhibits explosive behavior (like in a bubble), then $d^* > 1$. A case of particular interest is $0 < d^* \ll 1$, when the original series is “mildly non-stationary.” In this case, although differentiation is needed, a full integer differentiation removes excessive memory (and predictive power).

Figure 5.5 illustrates this concept. On the right y-axis, it plots the ADF statistic computed on E-mini S&P 500 futures log-prices, rolled forward using the ETF trick (see Chapter 2), downsampled to daily frequency, going back to the contract’s inception. On the x-axis, it displays the d value used to generate the series on which the ADF statistic was computed. The original series has an ADF statistic of –0.3387, while the returns series has an ADF statistic of –46.9114. At a 95% confidence level, the test’s critical value is –2.8623. The ADF statistic crosses that threshold in the vicinity of $d = 0.35$. The left y-axis plots the correlation between the original series ($d = 0$) and the differentiated series at various d values. At $d = 0.35$ the correlation is still very high, at 0.995. This confirms that the procedure introduced in this chapter has been successful in achieving stationarity without giving up too much memory. In contrast, the correlation between the original series and the returns series is only 0.03, hence showing that the standard integer differentiation wipes out the series’ memory almost entirely.

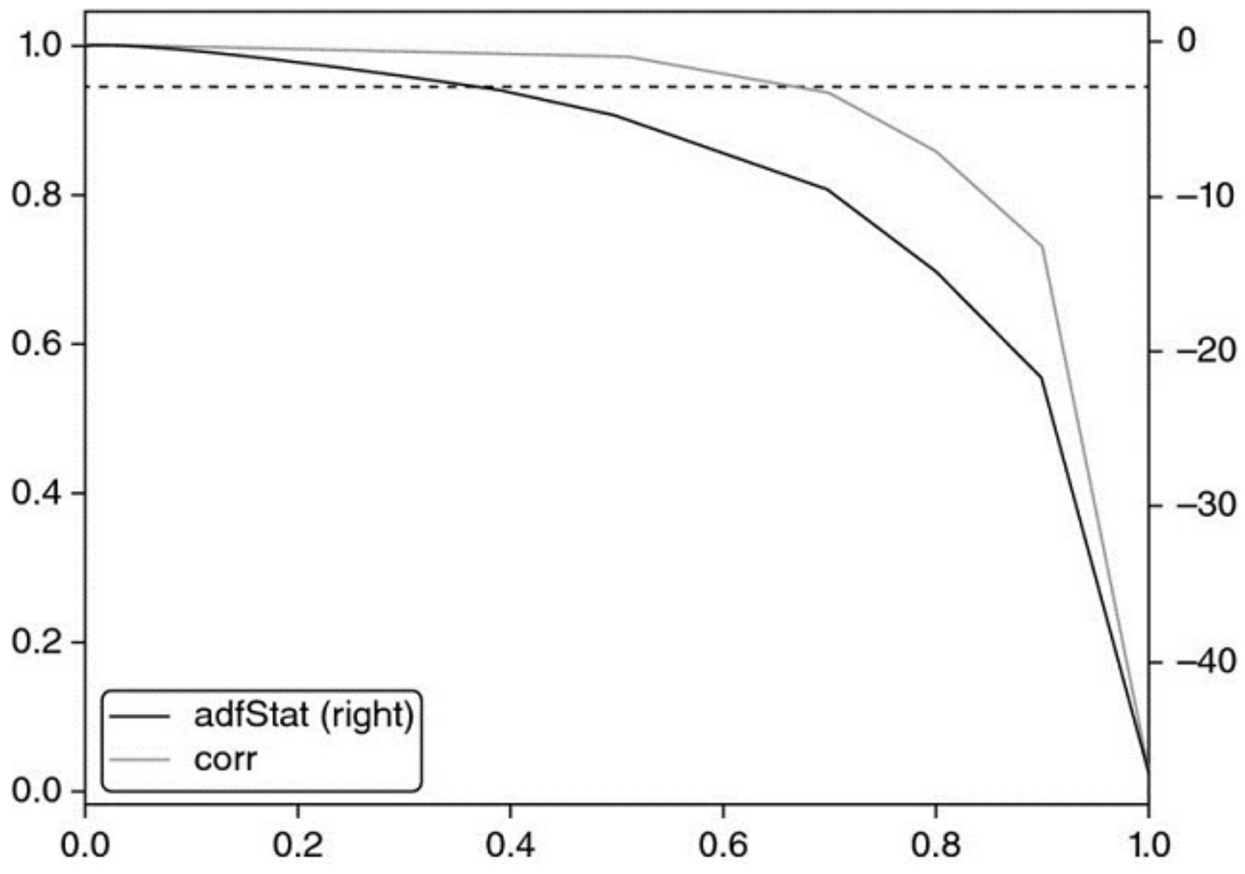


FIGURE 5.5 ADF statistic as a function of *d*, on E-mini S&P 500 futures log-prices

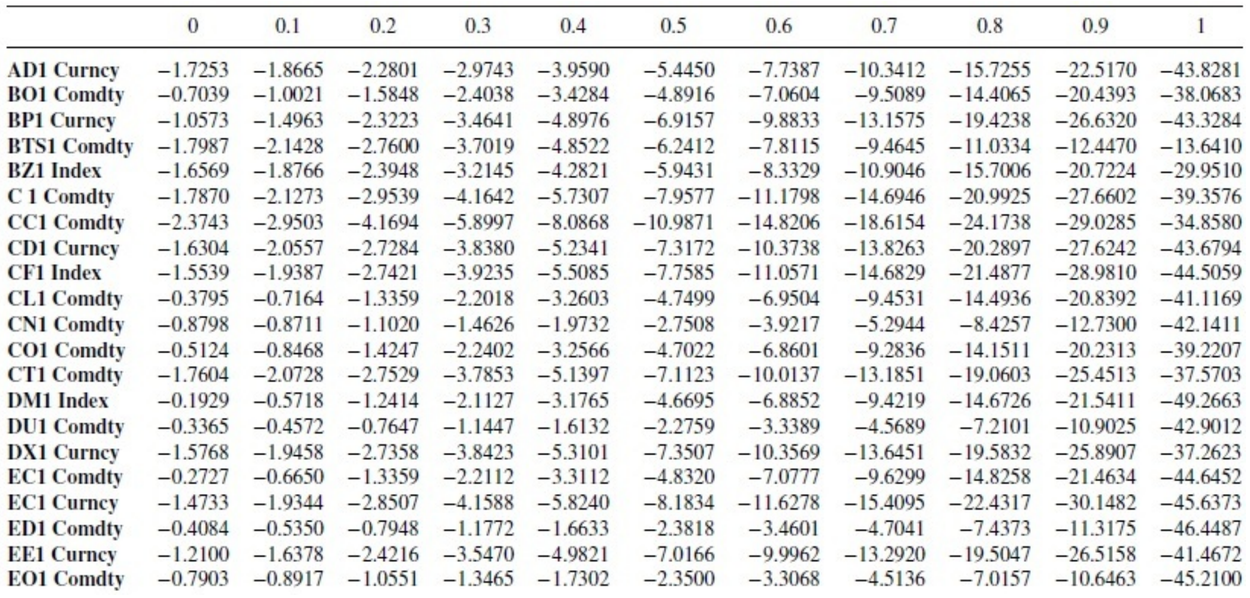
Virtually all finance papers attempt to recover stationarity by applying an integer differentiation $d = 1 \gg 0.35$, which means that most studies have over-differentiated the series, that is, they have removed much more memory than was necessary to satisfy standard econometric assumptions. Snippet 5.4 lists the code used to produce these results.

SNIPPET 5.4 FINDING THE MINIMUM *D* VALUE THAT PASSES THE ADF TEST

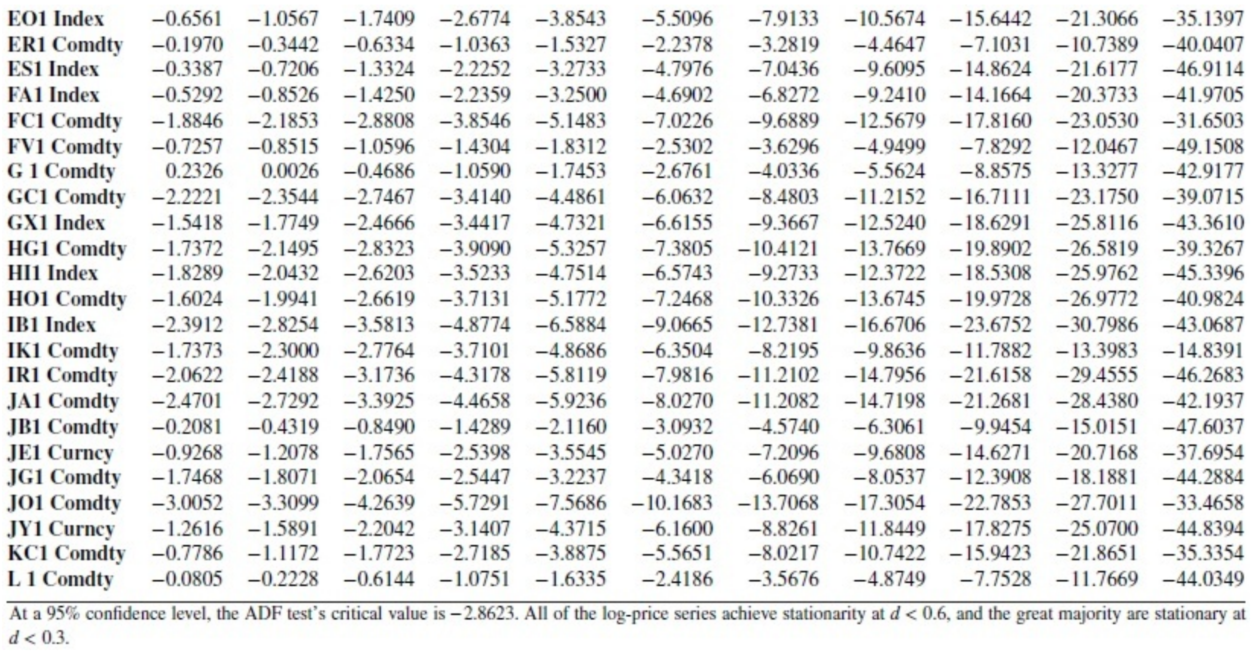
```
def plotMinFFD():
    from statsmodels.tsa.stattools import adfuller
    path,instName='./','ES1_Index_Method12'
    out=pd.DataFrame(columns=['adfStat','pVal','lags','nObs','95% conf','corr'])
    df0=pd.read_csv(path+instName+'.csv',index_col=0,parse_dates=True)
    for d in np.linspace(0,1,11):
        df1=np.log(df0[['Close']]).resample('1D').last() # downcast to daily obs
        df2=fracDiff_FFD(df1,d,thres=.01)
        corr=np.corrcoef(df1.loc[df2.index,'Close'],df2['Close'])[0,1]
        df2=adfuller(df2['Close'],maxlag=1,regression='c',autolag=None)
        out.loc[d]=list(df2[:4])+[df2[4]['5%']]+[corr] # with critical value
    out.to_csv(path+instName+'_testMinFFD.csv')
    out[['adfStat','corr']].plot(secondary_y='adfStat')
    mpl.axhline(out['95% conf'].mean(),linewidth=1,color='r',linestyle='dotted')
    mpl.savefig(path+instName+'_testMinFFD.png')
    return
```

The example on E-mini futures is by no means an exception. Table 5.1 shows the ADF statistics after applying FFD(d) on various values of d, for 87 of the most liquid futures worldwide. In all cases, the standard $d = 1$ used for computing returns implies over-differentiation. In fact, in all cases stationarity is achieved with $d < 0.6$. In some cases, like orange juice (JO1 Comdty) or live cattle (LC1 Comdty) no differentiation at all was needed.

TABLE 5.1 ADF Statistic on FFD(*d*) for Some of the Most Liquid Futures Contracts



	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
AD1 Curncy	−1.7253	−1.8665	−2.2801	−2.9743	−3.9590	−5.4450	−7.7387	−10.3412	−15.7255	−22.5170	−43.8281
BO1 Comdty	−0.7039	−1.0021	−1.5848	−2.4038	−3.4284	−4.8916	−7.0604	−9.5089	−14.4065	−20.4393	−38.0683
BP1 Curncy	−1.0573	−1.4963	−2.3223	−3.4641	−4.8976	−6.9157	−9.8833	−13.1575	−19.4238	−26.6320	−43.3284
BTS1 Comdty	−1.7987	−2.1428	−2.7600	−3.7019	−4.8522	−6.2412	−7.8115	−9.4645	−11.0334	−12.4470	−13.6410
BZ1 Index	−1.6569	−1.8766	−2.3948	−3.2145	−4.2821	−5.9431	−8.3329	−10.9046	−15.7006	−20.7224	−29.9510
C 1 Comdty	−1.7870	−2.1273	−2.9539	−4.1642	−5.7307	−7.9577	−11.1798	−14.6946	−20.9925	−27.6602	−39.3576
CC1 Comdty	−2.3743	−2.9503	−4.1694	−5.8997	−8.0868	−10.9871	−14.8206	−18.6154	−24.1738	−29.0285	−34.8580
CD1 Curncy	−1.6304	−2.0557	−2.7284	−3.8380	−5.2341	−7.3172	−10.3738	−13.8263	−20.2897	−27.6242	−43.6794
CF1 Index	−1.5539	−1.9387	−2.7421	−3.9235	−5.5085	−7.7585	−11.0571	−14.6829	−21.4877	−28.9810	−44.5059
CL1 Comdty	−0.3795	−0.7164	−1.3359	−2.2018	−3.2603	−4.7499	−6.9504	−9.4531	−14.4936	−20.8392	−41.1169
CN1 Comdty	−0.8798	−0.8711	−1.1020	−1.4626	−1.9732	−2.7508	−3.9217	−5.2944	−8.4257	−12.7300	−42.1411
CO1 Comdty	−0.5124	−0.8468	−1.4247	−2.2402	−3.2566	−4.7022	−6.8601	−9.2836	−14.1511	−20.2313	−39.2207
CT1 Comdty	−1.7604	−2.0728	−2.7529	−3.7853	−5.1397	−7.1123	−10.0137	−13.1851	−19.0603	−25.4513	−37.5703
DM1 Index	−0.1929	−0.5718	−1.2414	−2.1127	−3.1765	−4.6695	−6.8852	−9.4219	−14.6726	−21.5411	−49.2663
DU1 Comdty	−0.3365	−0.4572	−0.7647	−1.1447	−1.6132	−2.2759	−3.3389	−4.5689	−7.2101	−10.9025	−42.9012
DX1 Curncy	−1.5768	−1.9458	−2.7358	−3.8423	−5.3101	−7.3507	−10.3569	−13.6451	−19.5832	−25.8907	−37.2623
EC1 Comdty	−0.2727	−0.6650	−1.3359	−2.2112	−3.3112	−4.8320	−7.0777	−9.6299	−14.8258	−21.4634	−44.6452
EC1 Curncy	−1.4733	−1.9344	−2.8507	−4.1588	−5.8240	−8.1834	−11.6278	−15.4095	−22.4317	−30.1482	−45.6373
ED1 Comdty	−0.4084	−0.5350	−0.7948	−1.1772	−1.6633	−2.3818	−3.4601	−4.7041	−7.4373	−11.3175	−46.4487
EE1 Curncy	−1.2100	−1.6378	−2.4216	−3.5470	−4.9821	−7.0166	−9.9962	−13.2920	−19.5047	−26.5158	−41.4672
EO1 Comdty	−0.7903	−0.8917	−1.0551	−1.3465	−1.7302	−2.3500	−3.3068	−4.5136	−7.0157	−10.6463	−45.2100



EO1 Index	−0.6561	−1.0567	−1.7409	−2.6774	−3.8543	−5.5096	−7.9133	−10.5674	−15.6442	−21.3066	−35.1397
ER1 Comdty	−0.1970	−0.3442	−0.6334	−1.0363	−1.5327	−2.2378	−3.2819	−4.4647	−7.1031	−10.7389	−40.0407
ES1 Index	−0.3387	−0.7206	−1.3324	−2.2252	−3.2733	−4.7976	−7.0436	−9.6095	−14.8624	−21.6177	−46.9114
FA1 Index	−0.5292	−0.8526	−1.4250	−2.2359	−3.2500	−4.6902	−6.8272	−9.2410	−14.1664	−20.3733	−41.9705
FC1 Comdty	−1.8846	−2.1853	−2.8808	−3.8546	−5.1483	−7.0226	−9.6889	−12.5679	−17.8160	−23.0530	−31.6503
FV1 Comdty	−0.7257	−0.8515	−1.0596	−1.4304	−1.8312	−2.5302	−3.6296	−4.9499	−7.8292	−12.0467	−49.1508
G 1 Comdty	0.2326	0.0026	−0.4686	−1.0590	−1.7453	−2.6761	−4.0336	−5.5624	−8.8575	−13.3277	−42.9177
GC1 Comdty	−2.2221	−2.3544	−2.7467	−3.4140	−4.4861	−6.0632	−8.4803	−11.2152	−16.7111	−23.1750	−39.0715
GX1 Index	−1.5418	−1.7749	−2.4666	−3.4417	−4.7321	−6.6155	−9.3667	−12.5240	−18.6291	−25.8116	−43.3610
HG1 Comdty	−1.7372	−2.1495	−2.8323	−3.9090	−5.3257	−7.3805	−10.4121	−13.7669	−19.8902	−26.5819	−39.3267
HI1 Index	−1.8289	−2.0432	−2.6203	−3.5233	−4.7514	−6.5743	−9.2733	−12.3722	−18.5308	−25.9762	−45.3396
HO1 Comdty	−1.6024	−1.9941	−2.6619	−3.7131	−5.1772	−7.2468	−10.3326	−13.6745	−19.9728	−26.9772	−40.9824
IB1 Index	−2.3912	−2.8254	−3.5813	−4.8774	−6.5884	−9.0665	−12.7381	−16.6706	−23.6752	−30.7986	−43.0687
IK1 Comdty	−1.7373	−2.3000	−2.7764	−3.7101	−4.8686	−6.3504	−8.2195	−9.8636	−11.7882	−13.3983	−14.8391
IR1 Comdty	−2.0622	−2.4188	−3.1736	−4.3178	−5.8119	−7.9816	−11.2102	−14.7956	−21.6158	−29.4555	−46.2683
JA1 Comdty	−2.4701	−2.7292	−3.3925	−4.4658	−5.9236	−8.0270	−11.2082	−14.7198	−21.2681	−28.4380	−42.1937
JB1 Comdty	−0.2081	−0.4319	−0.8490	−1.4289	−2.1160	−3.0932	−4.5740	−6.3061	−9.9454	−15.0151	−47.6037
JE1 Curncy	−0.9268	−1.2078	−1.7565	−2.5398	−3.5545	−5.0270	−7.2096	−9.6808	−14.6271	−20.7168	−37.6954
JG1 Comdty	−1.7468	−1.8071	−2.0654	−2.5447	−3.2237	−4.3418	−6.0690	−8.0537	−12.3908	−18.1881	−44.2884
JO1 Comdty	−3.0052	−3.3099	−4.2639	−5.7291	−7.5686	−10.1683	−13.7068	−17.3054	−22.7853	−27.7011	−33.4658
JY1 Curncy	−1.2616	−1.5891	−2.2042	−3.1407	−4.3715	−6.1600	−8.8261	−11.8449	−17.8275	−25.0700	−44.8394
KC1 Comdty	−0.7786	−1.1172	−1.7723	−2.7185	−3.8875	−5.5651	−8.0217	−10.7422	−15.9423	−21.8651	−35.3354
L 1 Comdty	−0.0805	−0.2228	−0.6144	−1.0751	−1.6335	−2.4186	−3.5676	−4.8749	−7.7528	−11.7669	−44.0349

At a 95% confidence level, the ADF test's critical value is −2.8623. All of the log-price series achieve stationarity at $d < 0.6$, and the great majority are stationary at $d < 0.3$.

5.7 CONCLUSION

To summarize, most econometric analyses follow one of two paradigms:

1. Box-Jenkins: Returns are stationary, however memory-less.
2. Engle-Granger: Log-prices have memory, however they are non-stationary. Cointegration is the trick that makes regression work on non-stationary series, so that memory is preserved. However the number of cointegrated variables is limited, and the cointegrating vectors are notoriously unstable.

In contrast, the FFD approach introduced in this chapter shows that there is no need to give up all of the memory in order to gain stationarity. And there is no need for the cointegration trick as it relates to ML forecasting. Once you become familiar with FFD, it will allow you to achieve stationarity without renouncing to memory (or predictive power).

In practice, I suggest you experiment with the following transformation of your features: First, compute a cumulative sum of the time series. This guarantees that some order of differentiation is needed. Second, compute the FFD(d) series for various $d \in [0, 1]$. Third, determine the minimum d such that the p-value of the ADF statistic on FFD(d) falls below 5%. Fourth, use the FFD(d) series as your predictive feature.

EXERCISES

5.1 Generate a time series from an IID Gaussian random process. This is a memory-less, stationary series:

a. Compute the ADF statistic on this series. What is the p-value?

b. Compute the cumulative sum of the observations. This is a non-stationary series without memory.

   i. What is the order of integration of this cumulative series?

   ii. Compute the ADF statistic on this series. What is the p-value?

c. Differentiate the series twice. What is the p-value of this over-differentiated series?

5.2 Generate a time series that follows a sinusoidal function. This is a stationary series with memory.

a. Compute the ADF statistic on this series. What is the p-value?

b. Shift every observation by the same positive value. Compute the cumulative sum of the observations. This is a non-stationary series with memory.

   i. Compute the ADF statistic on this series. What is the p-value?

   ii. Apply an expanding window fracdiff, with $\tau = 1E - 2$. For what minimum d value do you get a p-value below 5%?

   iii. Apply FFD, with $\tau = 1E - 5$. For what minimum d value do you get a p-value below 5%?

5.3 Take the series from exercise 2.b:

a. Fit the series to a sine function. What is the R-squared?

b. Apply FFD($d = 1$). Fit the series to a sine function. What is the R-squared?

c. What value of d maximizes the R-squared of a sinusoidal fit on FFD(d). Why?

5.4 Take the dollar bar series on E-mini S&P 500 futures. Using the code in Snippet 5.3, for some $d \in [0, 2]$, compute `fracDiff_FFD(fracDiff_FFD(series,d),-d)`. What do you get? Why?

5.5 Take the dollar bar series on E-mini S&P 500 futures.

a. Form a new series as a cumulative sum of log-prices.

b. Apply FFD, with $\tau = 1E - 5$. Determine for what minimum $d \in [0, 2]$ the new series is stationary.

c. Compute the correlation of the fracdiff series to the original (untransformed) series.

d. Apply an Engel-Granger cointegration test on the original and fracdiff series. Are they cointegrated? Why?

e. Apply a Jarque-Bera normality test on the fracdiff series.

5.6 Take the fracdiff series from exercise 5.

a. Apply a CUSUM filter (Chapter 2), where h is twice the standard deviation of the series.

b. Use the filtered timestamps to sample a features' matrix. Use as one of the features the fracdiff value.

c. Form labels using the triple-barrier method, with symmetric horizontal barriers of twice the daily standard deviation, and a vertical barrier of 5 days.

d. Fit a bagging classifier of decision trees where:

   i. The observed features are bootstrapped using the sequential method from Chapter 4.

   ii. On each bootstrapped sample, sample weights are determined using the techniques from Chapter 4.

REFERENCES

Alexander, C. (2001): *Market Models*, 1st edition. John Wiley & Sons.

Hamilton, J. (1994): *Time Series Analysis*, 1st ed. Princeton University Press.

Hosking, J. (1981): "Fractional differencing." *Biometrika*, Vol. 68, No. 1, pp. 165–176.

Jensen, A. and M. Nielsen (2014): "A fast fractional difference algorithm." *Journal of Time Series Analysis*, Vol. 35, No. 5, pp. 428–436.

López de Prado, M. (2015): “The Future of Empirical Finance.” *Journal of Portfolio Management*, Vol. 41, No. 4, pp. 140–144. Available at https://ssrn.com/abstract=2609734.

BIBLIOGRAPHY

Cavaliere, G., M. Nielsen, and A. Taylor (2017): “Quasi-maximum likelihood estimation and bootstrap inference in fractional time series models with heteroskedasticity of unknown form.” *Journal of Econometrics*, Vol. 198, No. 1, pp. 165–188.

Johansen, S. and M. Nielsen (2012): “A necessary moment condition for the fractional functional central limit theorem.” *Econometric Theory*, Vol. 28, No. 3, pp. 671–679.

Johansen, S. and M. Nielsen (2012): “Likelihood inference for a fractionally cointegrated vector autoregressive model.” *Econometrica*, Vol. 80, No. 6, pp. 2267–2732.

Johansen, S. and M. Nielsen (2016): “The role of initial values in conditional sum-of-squares estimation of nonstationary fractional time series models.” *Econometric Theory*, Vol. 32, No. 5, pp. 1095–1139.

Jones, M., M. Nielsen and M. Popiel (2015): “A fractionally cointegrated VAR analysis of economic voting and political support.” *Canadian Journal of Economics*, Vol. 47, No. 4, pp. 1078–1130.

Mackinnon, J. and M. Nielsen, M. (2014): “Numerical distribution functions of fractional unit root and cointegration tests.” *Journal of Applied Econometrics*, Vol. 29, No. 1, pp. 161–171.