A Recursive Algorithm For Fractionally Differencing Long Data Series

Joseph McCarthy
Finance Department
Bryant College

Robert DiSario
Department of Mathematics
Bryant College

Hakan Saraoglu
Finance Department
Bryant College

We propose a recursive algorithm to fractionally difference time series data. The algorithm eliminates the need to evaluate the gamma function directly, and hence avoids the overflow problem that arises when fractionally differencing a long data series. The proposed algorithm can be implemented using any general matrix programming language. An implementation using SAS is presented. The algorithm and the code provide a practical approach to including fractional differencing as part of a time series data analysis.

Key words: Fractionally differencing, time series

Introduction

The process of differencing is widely used in time series data analysis. First differencing is often adequate to deal with nonstationary data for an ARIMA model. A useful generalization of integer differencing is fractional differencing. The resulting FARIMA models, or fractional ARIMA models, are often used for time series exhibiting long-range dependence (Beran (1994); Geweke and Porter-Hudak (1983); Granger and Joyeux (1980); Mandelbrot and Van Ness (1968)). Long-range dependent series have hyperbolically decaying autocorrelation functions, unlike the exponential decay found in autocorrelation functions for time series modeled by ARIMA.

Algorithms to do fractional differencing can be used in simulating FARIMA data, in fractionally differencing an empirical time series to obtain a series suitable for ARIMA modeling, and in testing for white noise of residuals after fitting a FARIMA model. Because long-range dependence is found in financial time series and in some geophysical time series, practical algorithms to accomplish fractional differencing are needed. Statistical packages are beginning to incorporate modules to do fractional differencing. However, some of these modules are limited to very small data sets. For example, the SAS function FDIF can only handle approximately 171 observations (SAS release 8.2 Proc IML; SAS Institute, Inc. 2001). This limit is apparently due to use of the gamma function. Our proposed algorithm uses a recursive approach to eliminate the need to compute gamma directly. Thus it provides a practical way to fractionally difference a time series of much more than 171 observations. As discussed in the results section, we have tested this procedure for a time series as large as 10,000 observations. The algorithm that we describe could be implemented in any general matrix programming language. We provide an implementation using the matrix language SAS IML (SAS Institute, Inc. 1990).

Method

Let \( y_t \) be obtained by taking the \( d \)th difference of a time series \( x_t; t = 0,1,\ldots,n - 1 \):
\[ y_t = (1 - B)^d x_t, \] (1)

where \( B \) is the backshift operator defined by

\[ Bx_t = x_{t-1}. \]

If \( d = 1 \), then \( y_t \) is the first difference:

\[ y_t = (1 - B)x_t = x_t - Bx_t = x_t - x_{t-1}. \] (2)

If \( d = 2 \), then \( y_t \) is the second difference:

\[ y_t = (1 - B)^2 x_t = (1 - 2B + B^2)x_t = x_t - 2x_{t-1} + x_{t-2}. \] (3)

We could also obtain this second difference by expanding \( (1 - B)^2 \) and applying the resulting second degree polynomial in \( B \) to \( x_t \).

\[ y_t = (1 - B)^2 x_t = (1 - 2B + B^2)x_t. \] (4)

In general, for any integer \( d \), the \( d \)th difference can be found by expanding \( (1 - B)^d \) and applying the resulting polynomial in \( B \) to \( x_t \). Fractional differencing \((-0.5 < d < 0.5)\) is defined in an analogous way. Expanding \( (1 - B)^d \) in a Taylor series (see Kaplan, 1984, p431):

\[
(1 - B)^d = \sum_{j=0}^{\infty} \frac{(-d)(1-d)(2-d)\cdots((j-1)-d)}{j!} B^j
\]

(6)

Next, multiplying by \( 1 = \frac{\Gamma(j - d)}{\Gamma(-d)} \) and reversing the order of the factors in the product we obtain:

\[
(1 - B)^d = \sum_{j=0}^{\infty} \frac{(j-1-d)(j-2-d)\cdots(j-d)\Gamma(j - d)}{j!\Gamma(-d)} B^j
\]

(7)

Finally, by repeatedly using the recurrence property of the gamma function:

\[ \Gamma(x) = (x - 1)\Gamma(x - 1) \]

we can re-express the numerator as \( \Gamma(j - d) \). Thus, we obtain

\[
(1 - B)^d = \sum_{j=0}^{\infty} \frac{\Gamma(j - d)}{\Gamma(j + 1)\Gamma(-d)} B^j,
\]

which is a commonly used representation for the fractional differencing operator (Jensen, 1999).

To implement a fractional differencing algorithm it necessary to compute the coefficients in the above series:

\[
C_j = \frac{\Gamma(j - d)}{\Gamma(j + 1)\Gamma(-d)} \quad j = 0,1,2,\ldots
\]

(8)

Because these coefficients are used to multiply observations in the time series, this infinite sequence of coefficients can be truncated to the length of the data series.

A problem arises when calculating these coefficients because for large values of \( j \) the numerator and denominator become very large and exceed the computational capacity of the computer. For example, the gamma function evaluated at 171 is approximately 7.257E306. Our approach uses the recursive property of the gamma function, \( \Gamma(x) = (x - 1)\Gamma(x - 1) \), to obtain a recursive property for the \( C_j \) as follows:
\[ C_0 = \frac{\Gamma(0-d)}{\Gamma(1)\Gamma(-d)} = 1 \]

\[ C_j = \frac{\Gamma(j-d)}{\Gamma(j+1)\Gamma(-d)} = \frac{(j-d-1)\Gamma(j-d-1)}{j\Gamma(j)\Gamma(-d)} = \frac{(j-d-1)}{j} C_{j-1} \quad (9). \]

Because the above recursive formula does not involve use of the gamma function, it is possible to calculate \( C_j \) for large values of \( j \). It is only necessary to multiply \( C_{j-1} \) by \( \frac{(j-d-1)}{j} \) which is computationally trivial. Our SAS program which implements this appears in Appendix I. The key lines of code which recursively calculate \( C_j \) follow. Note that in SAS the array \( C_j \) must be indexed from 1 to \( n \), rather than from 0 to \( n-1 \).

The fractionally differenced time series, \( y_t \), is obtained by convolving the input time series, \( x_t \), with the vector of coefficients \( C_j \). That is

\[ y_t = (1-B)^d x_t = \sum_{j=0}^{\infty} C_j x_{t-j} \quad (10). \]

The lines of SAS code that implement the convolution appear below.

```sas
jj=0;
do i=1 to n;
    if i=1 then Cj[i] = 1;
    else Cj[i] = Cj[i-1] * ((jj-d-1)/jj) ;
    jj=jj+1;
end;
```

The fractionally differenced time series, \( y_t \), is obtained by convolving the input time series, \( x_t \), with the vector of coefficients \( C_j \). That is

\[ y_t = (1-B)^d x_t = \sum_{j=0}^{\infty} C_j x_{t-j} \quad (10). \]

The lines of SAS code that implement the convolution appear below.

```sas
do i=1 to n;
    yt[i] = Cj[1:i]` * xt[i:1];
end;
```

Results

In the first example, we fractionally difference a small integer data series using \( d=0.5 \), then fractionally difference the result again using \( d=0.5 \). For this example, fractional differencing was done in two ways: first using the SAS function FDIF (SAS release 8.2 Proc IML); then using the code described above.

One reason for performing this test was to confirm that both approaches to fractional differencing produce the same result for a small time series. A second reason was to check that the \( d \) values are additive: fractional differencing twice with \( d=0.5 \) is the same as first differencing.

The data series and the two fractionally differenced series are presented in Table 1. The column labeled \( XT \) is the integer data series. \( YJ \) is the fractional difference of \( XT \) using ‘Call FDIF’ with \( d=0.5 \). \( ZJ \) is the fractional difference of \( YJ \) using Call FDIF with \( d=0.5 \). Next, \( YT \) is the fractional difference of \( XT \) using our recursive procedure with \( d=0.5 \). Finally, \( ZT \) is the fractional difference of \( YT \) using the procedure with \( d=0.5 \). Clearly, \( YT = YJ \) and \( ZT = ZJ \), showing that the two procedures produce the same results for this small data series. Also, the reader can check that \( ZT \) and \( ZJ \) are the same as would be obtained by doing first differencing. The program that produced all four series appears in Appendix II.

In the second example we use our recursive method to fractionally difference a random series of 10,000 observations. Note that the method using the SAS FDIF function will not run on a time series that is longer than approximately 171 observations (using a Pentium IV, running at 1.7 GHz) and therefore was not included in this example. The SAS LOG in Appendix III shows that the program using our method successfully ran. Thus this method provides a practical way to fractionally difference long time series. Implementing this algorithm in SAS provides a convenient way to include fractional differencing as part of a complete analysis of a long memory time series.
Conclusion

FARIMA models are commonly used to model long range dependent time series. In such cases, fractional differencing is often a useful part of the analysis. The practical way to fractionally difference a long time series is to use an algorithm that avoids calculating gamma(n) directly. (Although not discussed in the results section, we also ran our program on a series of 100,000 observations using 5 minutes of CPU time). Our implementation in SAS is a convenient way to incorporate fractional differencing into time series data analysis.

References


Table 1. Fractional differencing using SAS Call Fdif and using the recursive procedure.

<table>
<thead>
<tr>
<th>XT</th>
<th>YT</th>
<th>ZT</th>
<th>YJ</th>
<th>ZJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>582</td>
<td>582</td>
<td>582</td>
<td>582</td>
<td>582</td>
</tr>
<tr>
<td>227</td>
<td>-64</td>
<td>-355</td>
<td>-64</td>
<td>-355</td>
</tr>
<tr>
<td>410</td>
<td>223.75</td>
<td>183</td>
<td>223.75</td>
<td>183</td>
</tr>
<tr>
<td>109</td>
<td>-160.75</td>
<td>-301</td>
<td>-160.75</td>
<td>-301</td>
</tr>
<tr>
<td>686</td>
<td>543.3281</td>
<td>577</td>
<td>543.3281</td>
<td>577</td>
</tr>
<tr>
<td>753</td>
<td>345.9688</td>
<td>67</td>
<td>345.9688</td>
<td>67</td>
</tr>
<tr>
<td>903</td>
<td>399.7793</td>
<td>150</td>
<td>399.7793</td>
<td>150</td>
</tr>
<tr>
<td>996</td>
<td>377.9981</td>
<td>93</td>
<td>377.9981</td>
<td>93</td>
</tr>
<tr>
<td>60</td>
<td>-647.4</td>
<td>-936</td>
<td>-647.4</td>
<td>-936</td>
</tr>
<tr>
<td>76</td>
<td>-201.273</td>
<td>16</td>
<td>-201.273</td>
<td>16</td>
</tr>
<tr>
<td>716</td>
<td>523.3205</td>
<td>640</td>
<td>523.3205</td>
<td>640</td>
</tr>
<tr>
<td>202</td>
<td>-272.01</td>
<td>-514</td>
<td>-272.01</td>
<td>-514</td>
</tr>
<tr>
<td>637</td>
<td>361.6509</td>
<td>435</td>
<td>361.6509</td>
<td>435</td>
</tr>
<tr>
<td>60</td>
<td>-394.921</td>
<td>-577</td>
<td>-394.921</td>
<td>-577</td>
</tr>
<tr>
<td>314</td>
<td>109.65</td>
<td>254</td>
<td>109.65</td>
<td>254</td>
</tr>
<tr>
<td>969</td>
<td>691.8636</td>
<td>655</td>
<td>691.8636</td>
<td>655</td>
</tr>
<tr>
<td>87</td>
<td>-524.382</td>
<td>-882</td>
<td>-524.382</td>
<td>-882</td>
</tr>
<tr>
<td>660</td>
<td>406.5947</td>
<td>573</td>
<td>406.5947</td>
<td>573</td>
</tr>
<tr>
<td>719</td>
<td>248.2841</td>
<td>59</td>
<td>248.2841</td>
<td>59</td>
</tr>
<tr>
<td>784</td>
<td>241.7671</td>
<td>65</td>
<td>241.7671</td>
<td>65</td>
</tr>
</tbody>
</table>

Appendix I - SAS Program FRACDIFF.SAS

```
***************************************************************************;
* fracdiff.sas              *;
*                           *;
***************************************************************************;
* create random data for fractional differencing algorithm *;
* data one ;
* for n=171 both methods of fractional differencing work *;
* for n=172 call to fdif fails, but convolution works *;
* do i=1 to 10000;
```
MCCARTHY, DISARIO, & SARAOGLU

```plaintext
x=rand('NORMAL');
output;
end;
* fractional differencing algorithm implemented below *;
proc iml;
    use one;
    read all into xx;
    index=xx[,1];
    xt=xx[,2];
    n=nrow(xt);
* d = fractional differencing parameter *;
    d=.5;
* initialization;
    yt=j(n,1,0);
    Cj=j(n,1,0);
* do loop calculates coefficients using recursive method *;
    jj=0;
    do i=1 to n;
        if i=1 then Cj[i] = 1;
        else Cj[i]= Cj[i-1]*((jj-d-1)/jj) ;
        jj=jj+1;
    end;
* Convolution follows. The arrays are indexed in reverse order to
    implement the *;
* convolution. Also, the symbol for transpose in SAS IML is '          *
    do i=1 to n;
        yt[i]=Cj[1:i]*xt[i:1];
    end;
quit;
```

Appendix II - SAS Program TESTPROG4.SAS

```
***************;
* testprog4.sas *;
*                     *;
***************;
data one ;
* for n=171 both methods of fractional differencing work *;
* for n=172 call to fdif fails, but convolution works *;
    do i=1 to 20;
        x=int(rand('uniform')*1000);
        output;
    end;
proc print data=one ;
    run;
proc iml ;
    use one;
    read all into xx;
    index=xx[,1];
    xt=xx[,2];
    n=nrow(xt);d=.5;
```
* initialization;
yt=j(n,1,0);
zt=j(n,1,0);
yj=j(n,1,0);
zj=j(n,1,0);
Cj=j(n,1,0);
call fdif(yj, xt, .5);
call fdif(zj, yj, .5);
jj=0;
do i=1 to n;
  if i=1 then Cj[i] = 1;
  else Cj[i]= Cj[i-1]*((jj-d-1)/jj) ;
  jj=jj+1;
end;
do i=1 to n;
  * convolution follows *;
  yt[i]=Cj[1:i]`*xt[i:1];
end;
do i=1 to n;
  * convolution follows *;
  zt[i]=Cj[1:i]`*yt[i:1];
end;
print index xt yt zt yj zj;

Appendix III - SAS LOG for FRACDIFF.SAS

653 ***********************;
654 * fracdiff.sas          *
655 *                             *
656 ***********************;
657
658 * create random data for fractional differencing algorithm *;
659
data one ;
660 * for n=171 both methods of fractional differencing work *;
661 * for n=172 call to fdif fails, but convolution works *;
662 do i=1 to 10000;
663 x=rand('NORMAL');
664 output;
665 end;
666
667 * fractional differencing algorithm implemented below *;
668
NOTE: The data set WORK.ONE has 10000 observations and 2 variables.
NOTE: DATA statement used:
    real time       0.00 seconds
671 proc iml ;
NOTE: IML Ready
673   use one;
674   read all into xx;
675   index=xx[,1];
676   xt=xx[,2];
677   n=nrow(xt);
678 * d = fractional differencing parameter *;
679   d=.5;
680
681 * initialization;
682  yt=j(n,1,0);
683  Cj=j(n,1,0);
684
685 * do loop calculates coefficients using recursive method *;
686
687 jj=0 ;
688 do i=1 to n;
689   if i=1 then Cj[i] = 1;
690   else Cj[i]= Cj[i-1]*((jj-d-1)/jj) ;
691   jj=jj+1;
692 end;
693
694 * Convolution follows. Notice that the arrays are indexed in reverse order to implement the
694! *;  
695 * convolution. Also, the symbol for transpose in SAS IML is ' 
695! *;  
696
697 do i=1 to n;
698   yt[i]=Cj[1:i]`*xt[i:1];
699 end;
700 quit;
NOTE: Exiting IML.
NOTE: 7659 workspace compresses.
NOTE: PROCEDURE IML used:
   real time       3.18 seconds