

5-1-2003


A Recursive Algorithm For Fractionally Differencing Long Data Series

Joseph McCarthy
Bryant College

Robert DiSario
Bryant College

Hakan Saraoglu
Bryant College

Follow this and additional works at: <http://digitalcommons.wayne.edu/jmasm>

 Part of the [Applied Statistics Commons](#), [Social and Behavioral Sciences Commons](#), and the [Statistical Theory Commons](#)

Recommended Citation

McCarthy, Joseph; DiSario, Robert; and Saraoglu, Hakan (2003) "A Recursive Algorithm For Fractionally Differencing Long Data Series," *Journal of Modern Applied Statistical Methods*: Vol. 2 : Iss. 1 , Article 29.

DOI: 10.22237/jmasm/1051748940

Available at: <http://digitalcommons.wayne.edu/jmasm/vol2/iss1/29>

This Algorithms and Code is brought to you for free and open access by the Open Access Journals at DigitalCommons@WayneState. It has been accepted for inclusion in Journal of Modern Applied Statistical Methods by an authorized editor of DigitalCommons@WayneState.

A Recursive Algorithm For Fractionally Differencing Long Data Series

Joseph McCarthy
Finance Department

Robert DiSario
Department of Mathematics
Bryant College

Hakan Saraoglu
Finance Department

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).

Joseph McCarthy is a Professor in the Finance Department at Bryant College. He received a DBA in finance from the University of Colorado in 1983. His academic interests include fixed income instruments, non-linear modeling and wavelets. Robert DiSario is an Assistant Professor in the Department of Mathematics at Bryant College. He received a Ph.D. in statistics from Boston University in 1996. His academic interests include applied statistics and combinatorics. Hakan Saraoglu is an Associate Professor in the Finance Department at Bryant College. He received a Ph.D. in finance from Michigan State University in 1996. His academic interests include investments, international finance, and corporate finance.

Method

Let y_t be obtained by taking the d^{th} difference of a time series x_t ; $t = 0, 1, \dots, n - 1$:

$$y_t = (1 - B)^d x_t, \tag{1}$$

where B is the backshift operator defined by

$$Bx_t = x_{t-1}.$$

If $d=1$, then y_t is the first difference:

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

If $d=2$, then y_t is the second difference:

$$\begin{aligned} y_t &= (1 - B)^2 x_t = (1 - B)(x_t - x_{t-1}) \\ &= x_t - 2x_{t-1} + x_{t-2}. \end{aligned} \tag{3}$$

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

$$\begin{aligned} y_t &= (1 - B)^2 x_t = (1 - 2B + B^2)x_t \\ &= x_t - 2x_{t-1} + x_{t-2}. \end{aligned} \tag{4}$$

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):

$$\begin{aligned} (1 - B)^d &= 1 + \frac{d}{1!}(-B)^1 + \frac{d(d-1)}{2!}(-B)^2 \\ &\quad + \frac{d(d-1)(d-2)}{3!}(-B)^3 + \dots \\ &= \sum_{j=0}^{\infty} \frac{d(d-1)(d-2)\dots(d-(j-1))}{j!} (-1)^j B^j \end{aligned} \tag{5}$$

where the numerator in the above expression has j factors except when $j=0$ where it is unity. Now by multiplying each factor in the numerator by -1 we change the sign of each:

$$(1 - B)^d = \sum_{j=0}^{\infty} \frac{(-d)(1-d)(2-d)\dots((j-1)-d)}{j!} B^j \tag{6}$$

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

$$(1 - B)^d = \sum_{j=0}^{\infty} \frac{(j-1-d)(j-2-d)\dots(j-j-d)\Gamma(j-j-d)}{j!\Gamma(-d)} B^j \tag{7}$$

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:

$$C_j = \frac{\Gamma(j-d)}{\Gamma(j+1)\Gamma(-d)} \quad j=0,1,2,\dots \tag{8}.$$

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:

$$\begin{aligned}
 C_0 &= \frac{\Gamma(0-d)}{\Gamma(1)\Gamma(-d)} = 1 \\
 C_j &= \frac{\Gamma(j-d)}{\Gamma(j+1)\Gamma(-d)} \\
 &= \frac{(j-d-1)\Gamma(j-d-1)}{j\Gamma(j)\Gamma(-d)} \\
 &= \frac{(j-d-1)}{j} C_{j-1}
 \end{aligned} \quad (9).$$

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 $n-1$.

```

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

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

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

```

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

```

Using our approach we have been able to fractionally difference long data series. In the

results section we give an example using a series of 10,000 observations.

Results

In the first example, we fractionally difference a small integer data series using $d=.5$, then fractionally difference the result again using $d=.5$. 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 $d=.5$ 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 $d=.5$. **ZJ** is the fractional difference of **YJ** using Call FDIF with $d=.5$. Next, **YT** is the fractional difference of **XT** using our recursive procedure with $d=.5$. Finally, **ZT** is the fractional difference of **YT** using the procedure with $d=.5$. 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

Beran, J. (1994). *Statistics for long memory processes*. New York: Chapman & Hall.

Geweke, J. & Porter-Hudak, S. (1983). The estimation and application of long memory time series models. *Journal of Time Series Analysis*, 4, 221-238.

Granger, C. W. J. & Joyeux, R. (1980). An introduction to long memory time series models and fractional differencing. *Journal of Time Series Analysis* 1(1), 15-29.

Jensen, M. (1999). Using wavelets to obtain a consistent ordinary least squares estimator of the fractional differencing parameter. *Journal of Forecasting*, 18, 17-32.

Kaplan, W. (1984). *Advanced calculus*. (3rd ed.). Reading, MA: Addison-Wesley.

Mandelbrot, B. B. & Van Ness, J.W. (1968). Fractional Brownian motions, fractional noises and applications. *SIAM Review*, 10, 422-437.

SAS Institute, Inc., (1990). *SAS/IML Software: Usage and reference*, Version 6 Cary, NC: SAS.

SAS Institute, Inc., (2001). *SAS/IML Software: Changes and enhancements*, Release 8.2. Cary, NC: SAS.

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

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

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;
```

```

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
659 * create random data for fractional differencing algorithm *;
660
661 data one ;
662 * for n=171 both methods of fractional differencing work *;
663 * for n=172 call to fdif fails, but convolution works *;
664 do i=1 to 10000;
665 x=rand('NORMAL');
666 output;
667 end;
668
669
670 * fractional differencing algorithm implemented below *;
671

```

NOTE: The data set WORK.ONE has 10000 observations and 2 variables.

NOTE: DATA statement used:

real time 0.00 seconds

```

672 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
701 quit;
NOTE: Exiting IML.
NOTE: 7659 workspace compresses.
NOTE: PROCEDURE IML used:
      real time      3.18 seconds
```