Queen's Economics Department Working Paper No. 1307

A fast fractional difference algorithm

Andreas Noack Jensen
University of Copenhagen

Morten ÃŸrregaard Nielsen
Queen's University and CREATES

Department of Economics
Queen's University
94 University Avenue
Kingston, Ontario, Canada
K7L 3N6

4-2013

A fast fractional difference algorithm*

Andreas Noack Jensen
University of Copenhagen

Morten Ørregaard Nielsen†
Queen's University and CREATES

First version April, 2013.
This version March 7, 2014.

Abstract

We provide a fast algorithm for calculating the fractional difference of a time series. In standard implementations, the calculation speed (number of arithmetic operations) is of order T^2, where T is the length of the time series. Our algorithm allows calculation speed of order $T \log T$. For moderate and large sample sizes, the difference in computation time is substantial.

JEL Codes: C22, C63, C87.

Keywords: Circular convolution theorem, fast Fourier transform, fractional difference.

1 Introduction

In the estimation or simulation of fractionally integrated (or fractional) time series models, the computational cost is almost exclusively associated with the calculation of fractional differences. Indeed, the computational cost of these calculations can be so great that estimation or simulation of fractional time series models is infeasible when the sample size is very large.

In this paper, we derive an algorithm for the calculation of fractional differences based on circular convolutions. The advantage of our algorithm is that it is designed to exploit very efficient implementations of the discrete Fourier transform, i.e. the fast Fourier transform (Cooley and Tukey, 1965). The number of arithmetic operations required, and hence the calculation speed, of standard implementations of the fractional difference operation is of order T^2, see e.g. Palma (2007, p. 73), where T is the length of the

*We are grateful to the editor, Rob Taylor, two anonymous referees, Jurgen Doornik, Uwe Hassler, Søren Johansen, James MacKinnon, Rocco Mosconi, Peter Robinson, and participants at the 3rd Long Memory Symposium at CREATES for comments. We thank the Canada Research Chairs program, the Social Sciences and Humanities Research Council of Canada (SSHRC), and the Center for Research in Econometric Analysis of Time Series (CREATES, funded by the Danish National Research Foundation) for financial support.

†Corresponding author. Postal address: Department of Economics, Dunning Hall, Queen's University, 94 University Avenue, Kingston, Ontario K7L 3N6, Canada. Email address: mon@econ.queensu.ca

time series, i.e. the sample size. Note that we use "order" to denote the tight (asymptotic) bound, that is, $f(T)$ is of order $g(T)$ if for some T_0 and $K_u > K_l > 0$ it holds that $K_l|g(T)| \leq |f(T)| \leq K_u|g(T)|$ whenever $T > T_0$. In contrast, our algorithm is able to achieve order[1] $T \log T$. For large sample sizes, the difference in computation time is substantial.

As an example, suppose we observe a sample of size $T = 100\,000$, which is not at all unreasonable for applications in, e.g., finance. As illustrated below, in a standard MATLAB implementation (other languages provide similar timings), calculating the fractional difference just one time requires about 2.7 seconds of CPU time on an Intel Core i5-2400 3.1GHz desktop. In comparison, a MATLAB implementation of our algorithm is able to calculate the same fractional difference in only 0.02 seconds of CPU time. In the estimation of even the simplest fractional time series model, based on, e.g., a conditional-sum-of-squares criterion, one would expect to calculate several fractional differences for each iteration in the numerical optimization (one to evaluate the objective function and at least one more for each parameter to evaluate the gradient numerically). If 15-20 iterations are required to locate an optimum of the objective function, that suggests that roughly 100 fractional differences would need to be calculated, although of course this would depend on the number of parameters in the model estimated. Thus, for $T = 100\,000$, the difference in estimation time for the standard implementation versus our implementation of the fractional difference algorithm could very well be of the order of 4.5 minutes versus two seconds. The computational costs with standard implementations seems prohibitive for bootstrap or simulation procedures with large sample sizes. On the other hand, such procedures remain quite feasible with our implementation of the fractional difference operator.

In a related strand of literature on the so-called "type I" (or untruncated[2]) fractional processes, there has been some focus on fast algorithms for simulation of fractional processes. An early algorithm by Davies and Harte (1987), see also Craigmile (2003), Chen, Hurvich, and Lu (2006), and Doornik (2006), applies the circulant embedding method and the fast Fourier transform to generate a time series with given autocovariances, say $\gamma_0, \gamma_1, \ldots, \gamma_{T-1}$, and specifically type I fractional processes with $d < 1/2$ such that the autocovariances are well-defined. Also, an idea similar to our Theorem 1 appears in Sowell (1992, p. 170), using the continuous rather than the discrete Fourier transform, as an approximation device for the untruncated fractional difference operator for type I processes. Related approximate approaches for type I processes are the Whittle estimator, which can use the fast Fourier transform, e.g. Beran (1994, p. 116) or Palma (2007, p. 78), and truncation of the fractional filter at a fixed lag, e.g. Hasslet and Raftery (1989).

The remainder of the paper is laid out as follows. In the next section we describe the fractional difference operation in more detail and derive our proposed algorithm.

[1] All logarithms in this paper are to base two. However, since we only derive asymptotic orders, any factor of proportionality, and hence the base of the logarithms via the change of base formula, is irrelevant, and we therefore suppress the base number in the notation.

[2] The convention applied in this paper, see (1) below, is that of a "type II" (truncated) fractional process, see e.g. Marinucci and Robinson (1999). While that is certainly not the only relevant type of fractional process, these definitions are not essential to this paper, since our focus is on fast calculation of the fractional difference of a time series of finite length as in (1).

Section 3 provides numerical results, and some further discussion and conclusions are given in section 4.

2 Fast fractional difference algorithm

Consider the time series X_t, which is observed for $t = 1,\dots,T$. Suppose we want to calculate the fractional difference

$$Y_t = \Delta_+^d X_t = \sum_{j=0}^{t-1} \pi_j(-d) X_{t-j}, \qquad t = 1,\dots,T, \tag{1}$$

where the fractional coefficients $\pi_j(u)$ are defined as the coefficients in an expansion of $(1-z)^{-u}$, which are

$$\pi_j(u) = \frac{u(u+1)\cdots(u+j-1)}{j!}, \qquad j = 0,1,\dots. \tag{2}$$

Note that the summation in (1) is truncated at $t-1$ because we only observe X_t starting at time $t = 1$. The subscript "+" on the fractional difference operator thus indicates that only observations on X_t with a positive time index are included in the summation. If we had pre-sample observations (initial values) on X_t that we wanted to include, then the summation would be extended to include those as well; see Johansen and Nielsen (2012, 2013). However, such considerations are not essential to the developments in this paper, and therefore we do not consider this possibility further.

The standard calculation of the fractional difference in (1) is done as a linear convolution of the two series $X = (X_t)_{t=1}^T$ and $q = (\pi_{t-1}(-d))_{t=1}^T$. That is, the time series $Y = (Y_t)_{t=1}^T$ with t'th element given in (1) can be written as

$$Y_t = \sum_{j=1}^{t} q_j X_{t-j+1}, \qquad t = 1,\dots,T. \tag{3}$$

Because the number of arithmetic operations required in each sum in (3) is of order t, the whole linear convolution operation for $t = 1,\dots,T$ is of order T^2.

Our algorithm for the fractional difference operator takes advantage of a frequency-domain transformation, and we therefore define the discrete Fourier transform $f = (f_j)_{j=1}^T$ of a series $a = (a_t)_{t=1}^T$ as the solution to the equation $a = T^{-1}Ff$, where F is the Fourier matrix with (j,k)'th element $(F)_{jk} = w_T^{(j-1)(k-1)}$ and $w_T = e^{\mathrm{i}2\pi/T}$ with $\mathrm{i} = \sqrt{-1}$ denoting the imaginary unit. Each element of a can therefore be expressed in terms of the Fourier coefficients f_j and powers of w_T as

$$a_t = \frac{1}{T}\sum_{j=1}^{T} f_j w_T^{(t-1)(j-1)}, \qquad t = 1,\dots,T. \tag{4}$$

Since F is symmetric and $F\overline{F} = TI_T$, where the bar denotes complex conjugation, the inverse operation is $(T^{-1}F)^{-1} = \overline{F}$, i.e. the complex conjugate of each element in F, such that $f_j = \sum_{t=1}^T a_t w_T^{-(t-1)(j-1)}$. Thus, the matrix $\overline{F}$ represents the discrete Fourier transform whereas $T^{-1}F$ is the inverse transform.

The circular convolution of two series a and b of length T is denoted $a \circledast b$ and defined as

$$(a \circledast b)_t = \sum_{j=1}^{T} a_j b_{t-j+1}, \qquad t = 1, \ldots, T, \tag{5}$$

where the sequences are extended periodically such that $a_{j+nT} = a_j$ and $b_{j+nT} = b_j$ for all $n = 0, \pm 1, \pm 2, \ldots$.

In Theorem 1 below, we state the finite version of the circular convolution theorem, which shows how the circular convolution of finite sequences in (5) can be calculated by application of the discrete Fourier transform. For periodic integrable functions this result can be found in, e.g., Zygmund (2003, Theorem 1.5, p. 36). The finite version has appeared in various forms in the engineering literature as an important application of the fast Fourier transform, see e.g. Stockham (1966, p. 230), Cooley, Lewis, and Welch (1969, p. 32), and Oppenheim, Schafer, and Buck (1999, Chap. 8). The version in Theorem 1 allows a simple proof of our main result. Because the notion of circular convolution of finite sequences seems less known in the econometrics literature we provide a brief proof of the theorem.

Theorem 1 *Let $a = (a_t)_{t=1}^T$ and $b = (b_t)_{t=1}^T$ be two sequences. Then*

$$a \circledast b = T^{-1} F(\overline{F} a \circ \overline{F} b), \tag{6}$$

where the symbol "$\circ$" denotes element-wise multiplication.

Proof. Let $f = \overline{F} a$ and $g = \overline{F} b$ denote the discrete Fourier transforms of a and b, respectively. It then needs to be shown that $a \circledast b = T^{-1} F(f \circ g)$. To do this, insert the expressions for a and b in terms of their discrete Fourier transforms,

$$\begin{aligned}
(a \circledast b)_t &= \sum_{j=1}^{T} a_j b_{t-j+1} = \sum_{j=1}^{T} \left(T^{-1} \sum_{s=1}^{T} f_s w_T^{(j-1)(s-1)} \right) \left(T^{-1} \sum_{u=1}^{T} g_u w_T^{(t-j)(u-1)} \right) \\
&= T^{-2} \sum_{s=1}^{T} \sum_{u=1}^{T} f_s g_u \sum_{j=1}^{T} w_T^{(t-1)(s-1)+(j-1)(s-u)} \\
&= T^{-2} \sum_{s=1}^{T} \sum_{u=1}^{T} f_s g_u w_T^{(t-1)(s-1)} \sum_{j=1}^{T} w_T^{(j-1)(s-u)} = T^{-1} \sum_{s=1}^{T} f_s g_s w_T^{(t-1)(s-1)},
\end{aligned}$$

where the last equality follows because of the well-known result

$$\sum_{j=1}^{T} w_T^{(j-1)k} = \begin{cases} T & \text{if } k \equiv 0 \pmod{T}, \\ 0 & \text{if } k \not\equiv 0 \pmod{T}. \end{cases}$$

This shows that the Fourier coefficients of $a \circledast b$ are given by the elementwise product of the Fourier coefficients of a and b. ∎

The next theorem presents our main result, which is to show how the finite circular convolution theorem can be used to calculate the fractional difference in (1), or equivalently in (3), by the discrete Fourier transform. For any $T \times 1$ vector a, let

$$\tilde{a} = [a', 0'_{T-1}]' \tag{7}$$

denote the $(2T-1) \times 1$ vector consisting of a extended with $T-1$ zeros.

Theorem 2 *The fractionally differenced time series Y in* (3) *can be calculated as the first T elements of the $(2T-1)\times 1$ vector*

$$T^{-1}F(\overline{F}\tilde{q}\circ\overline{F}\tilde{X}). \tag{8}$$

Proof. By equation (6) it holds that the t'th element of (8) is equal to $(\tilde{q}\circledast\tilde{X})_t$. Furthermore, for $t=1,\ldots,T$, we have from definition (5) that

$$(\tilde{q}\circledast\tilde{X})_t=\sum_{j=1}^{2T-1}\tilde{q}_j\tilde{X}_{t-j+1}=\sum_{j=1}^{t}\tilde{q}_j\tilde{X}_{t-j+1}+\sum_{j=t+1}^{2T-1}\tilde{q}_j\tilde{X}_{2T+t-j}=\sum_{j=0}^{t-1}\pi_j(-d)X_{t-j} \tag{9}$$

because $\tilde{q}_j=\pi_{j-1}(-d)$ for $j=1,\ldots,T$ and $\tilde{q}_j=0$ for $j\geq T+1$, while $\tilde{X}_{t-j+1}=X_{t-j+1}$ for $j=1,\ldots,t$ and $\tilde{X}_{2T+t-j}=0$ for $t+1\leq j\leq T$ by definition (7). ∎

The significance of Theorem 2 lies in the fact that the discrete Fourier transform can be calculated very efficiently by means of the fast Fourier transform algorithm, where the number of arithmetic operations required is proportional to $T\log T$, see Cooley and Tukey (1965). Because the operation in (8) only applies three discrete Fourier transforms and one element-wise multiplication of two vectors (which is of order T), the fractional difference algorithm (8) in Theorem 2 is itself of order $T\log T$.

Note that our result in Theorem 2 provides an exact calculation of the fractional difference in (1), and that no approximation is involved. Finally, also note that, depending on the particular implementation of the fast Fourier transformation applied, it may be necessary in practice to extend the series X and q to a length greater than the $2T-1$ used in (7). Specifically, some implementations of the fast Fourier transform require the length to be a power of two, and in that case $\tilde{X}$ and $\tilde{q}$ should be extended with zeros to length equal to the smallest power of two that is at least $2T-1$.

3 Numerical results

In this section we illustrate numerically the difference in computational cost between the standard implementation in (1) or (3) and the algorithm (8) in Theorem 2 for a range of sample sizes, T.

The baseline algorithm computes the linear convolution (3) directly, where the number of required arithmetic operations is of order T^2. The computation time is expected to be proportional to the number of arithmetic operations, and consequently also of order T^2. These timings are compared to our algorithm which is of order $T\log T$ based on the fast Fourier transform. The algorithms differ in the number of arithmetic operations required and therefore the results should be independent of programming language. However, in practice this may not be true because of, e.g., different implementations of the fast Fourier transform. We present results in the three popular languages: MATLAB (MathWorks, 2013), Ox (Doornik, 2007), and R (R Core Team, 2013) in order to exemplify the time gains in practical application.[3]

[3]A distinction between interpreted and compiled routines could be made, suggesting that, in some cases, the comparisons are not really fair. In this light, the comparisons made here could be viewed as conservative, since this point actually goes in favor of our proposed algorithm. We thank a referee for pointing this out.

MATLAB does not ship with a function for fractional differencing. Instead we have written the function `fracfilter` that computes the fractional difference by direct linear convolution through the MATLAB function `filter`. The function `fracdiff` uses the fast Fourier transform to compute the convolution as in Theorem 2, and the time series is padded with zeros such the total length of the series is a power of two. Ox, on the other hand, has the built-in function `diffpow` for fractional differencing, see the `arfima` package v1.04 by Doornik and Ooms (2003), and we use that as the benchmark. The function for the fast Fourier transform in Ox automatically pads the input with sufficient zeros, and hence we do not need to do so in our code, although we still have to pad the series as in (7). Finally, R also has a package for analyzing fractionally differenced data, namely `fracdiff` by Maechler (2012) (not to be confused with our algorithm of the same name), which has the built-in function `diffseries` for fractional differencing. We compare that with our implementation, which again uses the fast Fourier transform and padding with zeros such that the length of the time series is a power of two. All these algorithms are presented in Listings 1, 2, and 3, for MATLAB, Ox, and R, respectively, and are downloadable from the authors' websites.

Listing 1: Matlab code

```
function [dx] = fracfilter(x, d)
    T = size(x, 1);
    k = (1:T-1)';
    b = [1;cumprod((k-d-1)./k)];
    dx = filter(b, 1, x);
end

function [dx] = fracdiff(x, d)
    T = size(x, 1);
    np2 = 2.^nextpow2(2*T-1);
    k = (1:T-1)';
    b = [1;cumprod((k-d-1)./k)];
    dx = ifft(fft(x, np2).*fft(b, np2));
    dx = dx(1:T, :);
end
```

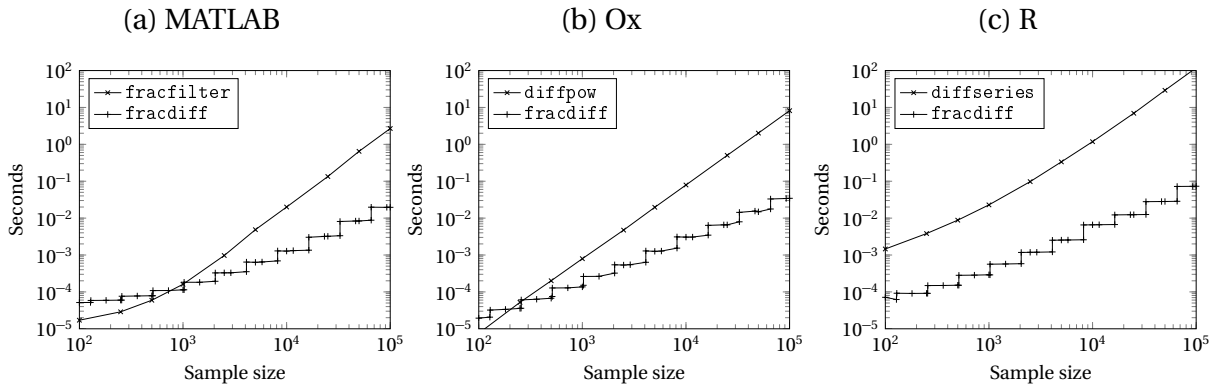
Listing 2: Ox code

```
fracdiff(const x, const d)
{
    decl T, k, b, dx;
    T = rows(x);
    k = range(1, T-1)';
    b = 1|cumprod(((k-d-1)./k));
    dx = fft(cmul(fft(b'~zeros(1, T-1)), fft(x'~zeros(1, T-1))), 2);
    return dx[:T-1]';
}
```

Listing 3: R code

```
fracdiff <- function(x, d){
  iT <- length(x)
  np2 <- nextn(2*iT - 1, 2)
```

Figure 1: Computation times in seconds against sample size



Note: The figure displays computation times in seconds for a range of sample sizes. Panels (a), (b), and (c) show the timings for MATLAB, Ox, and R, respectively. In each panel, both axes are logarithmic.

```
  k <- 1:(iT-1)
  b <- c(1, cumprod((k - d - 1)/k))
  dx <- fft(fft(c(b, rep(0, np2-iT)))*
        fft(c(x, rep(0, np2-iT))), inverse=T)/np2;
  return(Re(dx[1:iT]))
}
```

The computations are run on a desktop with an Intel Core i5-2400 3.1GHz processor running Ubuntu 13.10. The software versions are MATLAB 2013b, Ox Professional 7.0, and R 3.0.0. The timings are computed for sample sizes ranging from 100 to 100 000. For each sample size the fractional difference is repeatedly computed for approximately ten seconds and the average computation time is calculated. A minimum of ten repetitions has also been imposed to improve the precision for the largest sample sizes.[4] The resulting computation times are plotted with logarithmic axes in Figure 1. This clearly shows the different orders of the algorithms. The graphs for the benchmark algorithms are nearly straight lines with slope two except for the shortest samples. For our `fracdiff` algorithm, the graphs appear like step functions with jumps at each power of two, due to the application of the fast Fourier transform and the padding with zeros to a length of powers of two. Overall, Figure 1 clearly shows the advantage of the algorithm in Theorem 2 in terms of computation speed, especially when recalling that the axes are logarithmic.

In Table 1 we give some examples of the actual computing time in milliseconds required to calculate one fractional difference for sample sizes ranging from $T = 100$ to 100 000 using both the standard implementations as well as our fast fractional difference algorithm, `fracdiff`, presented in Listings 1, 2, and 3.

For samples of $T = 100$ the computation times are all very small, at least in MATLAB and Ox, and even though the new `fracdiff` algorithm is actually slower than the benchmark in MATLAB and Ox, they are all very fast and in practice there will hardly be

[4] To avoid differences in computation time caused by parallelization or multi-threading, the computations in MATLAB and Ox are done with a single-thread flag. Since R is single-threaded no special flags are used for the calculations.

Table 1: Examples of computing time

		Sample size									
		100	250	500	1000	2500	5000	10000	25000	50000	100000
MATLAB	`fracfilter`	0.0171	0.0288	0.0594	0.161	0.970	4.861	20.00	134.21	646.12	2683.3
	`fracdiff`	0.0515	0.0598	0.0786	0.114	0.329	0.635	1.28	3.22	8.36	19.6
Ox	`diffpow`	0.0076	0.0521	0.2007	0.795	4.736	19.669	79.10	503.49	2018.24	8180.8
	`fracdiff`	0.0194	0.0357	0.0674	0.137	0.535	1.260	3.06	6.57	14.84	34.4
R	`diffseries`	1.4584	3.8173	8.8022	22.861	97.580	334.638	1180.79	6956.00	29178.90	118603.5
	`fracdiff`	0.0713	0.0916	0.1518	0.291	1.184	2.542	6.59	12.40	28.24	73.2

Note: Entries are computing times in milliseconds for the calculation of one fractional difference for a variety of sample sizes and for the algorithms given in Listings 1, 2, and 3 as well as the benchmark algorithms. The reported times are averages of repeated calculations of the fractional difference for approximately ten seconds or at least 10 repetitions.

any noticeable difference between the implementations. In MATLAB, our algorithm becomes faster than the traditional method for samples of size approximately $T = 750$, and for Ox the crossover between the two methods is approximately $T = 180$. The traditional implementation in R is very slow and our method is much faster even for $T = 100$.

Because the algorithms scale differently, our method rapidly becomes much faster as the sample size grows, and even though the crossover in MATLAB is around $T = 750$, our method is already three times faster for $T = 2500$. For the other two languages the relative differences are even larger. For a sample size of $T = 100\,000$, the differences in computation times are enormous: compared to the benchmark algorithms, our proposed `fracdiff` algorithm is about 137 times faster in MATLAB, about 239 times faster in Ox, and about 1620 times faster in R.

Of course, the most important reason for having a fast fractional differencing algorithm is its repeated application in, for example, estimation, simulation, or bootstrap procedures. Clearly, the relative timings of our algorithm compared with the benchmark algorithms may differ in such settings, but the crossover points remain the same as those given above for the calculation of the fractional difference itself. The reason, of course, is that the proposed algorithm is exact, so that everything else is identical to the benchmark case; e.g. in the estimation procedure, the same number of iterations will be required in the numerical optimization, and within each iteration the same number of fractional differences and other calculations will be performed.

4 Discussion and conclusions

In this paper we have provided a fast algorithm for calculating the fractional difference of a time series based on the circular convolution theorem and the fast Fourier transform. The required number of arithmetic operations for our algorithm is of order $T \log T$ compared to T^2 for standard implementations, and similarly for the computation time. For large sample sizes, the difference in computation time is very substantial and can easily be the difference between feasible and infeasible estimation with moderate to large sample sizes. Moreover, the much faster calculation of the fractional difference achieved by our algorithm opens up new possibilities for bootstrap or simulation methods to be applied to fractional time series models with moderate to large sample sizes.

Of course, large data sets are common in many fields such as meteorology and finance. For example, in Carlini, Manzoni, and Mosconi (2010) and Bollerslev, Osterreider, Sizova, and Tauchen (2013), the authors apply the fractional cointegration model of Johansen and Nielsen (2012) to large data sets in finance. More specifically, Carlini *et al.* (2010) analyze supply and demand imbalances on stock prices using high-frequency observations. In the estimation, the authors use only a small subset consisting of $T = 110\,000$ observations from a data set with a total of $T = 5.8$ million observations, citing the "extreme computational burden" of the estimation. Indeed, we found that the time required to compute just one fractional difference with $T = 5.8$ million using the standard `fracfilter` implementation in MATLAB is about seven hours. On the other hand, with our `fracdiff` algorithm, the same calculation of one fractional difference takes only 1.7 seconds. Thus, the computation time of our algorithm is about 15 000 times faster, which is likely sufficiently fast to allow estimation with the full sample.

We note that the computationally intensive part in our method is the calculation of

the discrete Fourier transform. When the fractional differencing is applied to multiple times series, the series coefficients only need to be transformed once and in consequence our algorithm scales well with the number of variables. In addition, the fast Fourier transform can be computed in parallel. In our simulations, we have explicitly not taken advantage of this feature, so as to obtain more reasonable comparison with the traditional implementation. However, in practice computing the fast Fourier transform in parallel will make our algorithm even faster on most modern computers since they are typically equipped with several cores.

Finally, we note that "smoother" lines, compared to the step function-type lines in Figure 1, could be achieved by padding the series such that the total length is a product of small prime numbers, i.e. $2^k 3^m 5^n$, as long as the particular implementation of the fast Fourier transform supports this feature. In the calculation of the fast Fourier transform, this prevents the steps when $2T-1$ is slightly greater than a power of two. Indeed, in unreported MATLAB calculations, we found that additional (very modest) decreases in computation time were attained with this procedure compared to the `fracdiff` algorithm in Listing 1 above. However, to keep focus on the (fast) calculation of the fractional difference, rather than efficient calculation of the fast Fourier transform, we have not included an in-depth discussion of the benefits of padding the series to a length given as a product of small primes.

5 List of references

1. Beran, J. (1994). *Statistics for Long-Memory Processes*, Chapman and Hall, London, England.
2. Bollerslev, T., D. Osterreider, N. Sizova, and G. Tauchen (2013). Risk and return: long-run relations, fractional cointegration, and return predictability. *Journal of Financial Economics* 108, 409–424.
3. Carlini, F., M. Manzoni, and R. Mosconi (2010). The impact of supply and demand imbalance on stock prices: An analysis based on fractional cointegration using Borsa Italiana ultra high frequency data. Working paper, Politecnico di Milano.
4. Chen, W.W., C.M. Hurvich, and Y. Lu (2006). On the correlation matrix of the discrete Fourier transform and the fast solution of large Toeplitz systems for long-memory time series. *Journal of the American Statistical Association* 101, 812–822.
5. Cooley, J.W., P.A.W. Lewis, and P.D. Welch (1969). The fast Fourier transform and its applications. *IEEE Transactions on Education* 12, 27–34.
6. Cooley, J.W. and J.W. Tukey (1965). An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation* 19, 297–301.
7. Craigmile, P.F. (2003). Simulating a class of stationary Gaussian processes using the Davies-Harte algorithm, with application to long memory processes. *Journal of Time Series Analysis* 24, 505–511.
8. Davies, R.B. and D.S. Harte (1987). Tests for Hurst effect. *Biometrika* 74, 95–101.
9. Doornik, J.A. (2006). Efficient ARFIMA modelling when the sample size is large. Unpublished manuscript, University of Oxford.
10. Doornik, J.A. (2007). *Object-Oriented Matrix Programming Using Ox*, 3rd ed., Timberlake Consultants Press, London, England.
11. Doornik, J.A. and M. Ooms (2003). Computational aspects of maximum likelihood

estimation of autoregressive fractionally integrated moving average models. *Computational Statistics and Data Analysis* 41, 333–348.
12. Haslett, J. and A.E. Raftery (1989). Space-time modelling with long-memory dependence: assessing Ireland's wind power resource (with discussion). *Journal of the Royal Statistical Society Series C* 38, 1–21.
13. Johansen, S. and M.Ø. Nielsen (2012). Likelihood inference for a fractionally cointegrated vector autoregressive model. *Econometrica* 80, 2667–2732.
14. Johansen, S. and M.Ø. Nielsen (2013). The role of initial values in nonstationary fractional time series models. QED working paper 1300, Queen's University.
15. Maechler, M. (2012). The `fracdiff` package for R, version 1.4-2. URL: `http://cran.r-project.org/web/packages/fracdiff/`.
16. Marinucci, D. and P.M. Robinson (1999). Alternative forms of fractional Brownian motion. *Journal of Statistical Planning and Inference* 80, 111–122.
17. MathWorks (2013). *MATLAB 2013b*, The MathWorks, Inc., Natick, MA.
18. Oppenheim, A.V., R.W. Schafer, and J.R. Buck (1999). *Discrete-Time Signal Processing*, 2nd ed., Prentice Hall, New Jersey.
19. Palma, W. (2007). *Long-Memory Time Series: Theory and Methods*, John Wiley and Sons, New Jersey.
20. R Core Team (2013). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL: `http://www.R-project.org`.
21. Sowell, F. (1992). Maximum likelihood estimation of stationary univariate fractionally integrated time series models. *Journal of Econometrics* 53, 165–188.
22. Stockham, T.G. (1966). High-speed convolution and correlation. *Proceedings of the Spring Joint Computer Conference* 28, 229–233.
23. Zygmund, A. (2003). *Trigonometric Series*, vol. I and II, 3rd rev. ed., Cambridge University Press, Cambridge, England.