Fractional Differentiation on long-memory Time Series: a case of study in Fractional Brownian Motion Processes

Frank Salvador Ygnacio Rosas

Co-founder and Quantitative researcher at Quantmoon Technologies

The aim of this project is the implementation of an optimal fractional differentiation procedure for time series with 'long-memory'. Moreover, since there is a trade-off between the informative dependence of the time series and the stationarity needed to make statistical inferences, this project proposes a recursive binary search method to get the optimal (minimum) non-integer order of differentiation to make the time series (weakly) stationary, while preserving as much as memory as possible. This procedure will be tested on several Fractional Brownian motion simulations using different values for the Hurst exponent between 1/2 and 1. This will make possible to get an empirical relationship between the latter as a measure of the degree of dependency, and the optimal non-integer order of differentiation. The hypothesis is that this relationship tends to be linear, being the classical d(1) order of differentiation only optimal mostly when H is closer to 1. Thus, the project starts by implementing the non-integer differentiation method for time series developed by Lopez de Prado (2018). Then, a new recursive binary search algorithm to find the optimal non-integer "d" is proposed, due to its efficient computational time, degree of precision and simplicity. This will be tested over the Fractional Brownian Motion simulations, proving the linear relationship between the Hurst exponent and the optimal "d" developed in this project. Finally, some general considerations about the differentiation procedure implemented are outlined, as well as possible improvements.

Why Fractional Differentiation for Time Series?

The first thing necessary to understand is why fractional differentiation might be a better method in the context of time series analysis. Essentially, some time series show a high level of statistical dependence

between two points over time, well-known as “long-memory”. This informational dependency is a consequence of the fact that each new state depends upon a “long” history of previous states. However, statistical modelling faces some difficulties in dealing with this condition since most of its procedures work over the assumption of IID processes. Thus, practitioners are constantly seeking for stationarity by implementing integer differentiations as the standard transformation tool. Then, they try to recap the marginal signal in this new transformed time series, leading with either a restricted set of models or a complex combination of them. In the quantitative finance area, for instance, a time series of prices is transformed by applying a differentiation of order d(1) as a rule of thumb, which leads to work with a series of difference of prices. Although this transformation achieves stationarity, it comes at the expense of removing all memory from the original time series; that is, its trend, which is the key feature of any model’s predictive power. Note that any differentiation on a time series will remove some of its memory by default. However, in the majority of cases, it is not necessary to remove all of them just to achieve the stationarity condition. Additionally, since time series are mostly non-invariant processes, it is also difficult to achieve strong stationarity. This is why, commonly, it is more than enough to reach weak stationarity to afford any statistical inference. Therefore, instead of looking for an integer differentiation, such as d(1), a fractional differentiation procedure could be implemented, such as:

$$d\,(*), \quad 0 < (*) < 1$$

However, since there are infinite values between 0 and 1, there are infinite possibilities to get our new fractional differentiated time series. This is the reason why this project focuses on the implementation of a procedure to find the optimal $d\,(*)$ that makes possible to get a transformed time series weakly stationarity and, at the same time, with a significant autocorrelation as the representative metric of its memory.

In that sense, the project could be divided as follows. In the first part (**Part I**), the Fractional Differentiation Method for Time Series will be formally described, including its implementation using Wolfram language. In the same section, the project is going to define a recursive binary search algorithm to find the optimal order of differentiation given a specific time series. Then, in the second part (**Part II**), this search algorithm will be tested over many Fractional Brownian Motion simulations for different values of the Hurst Exponent H, that, in the context of long-memory processes, lies between 1/2 and 1. Finally, the project finishes with the “Concluding Remarks” section that summarizes the key ideas of all of these procedures.

Part I: The Fractional Differentiation Method

Formal Definition

Say a time series $X = \{x_t,\ x_{t-1},\ x_{t-2}, \ldots\}$ can be differentiated by the *Finite Difference Operator* ∇^n, such as $\nabla^n x_t = (1 - B)^n x_t,\ \forall\, n \in \mathbb{N}$, where $B^n x_t = x_{t-n}$ is the classical *Backward Shift Operator*. Thus, the goal is:

$$n \in \mathbb{N} \to d \in \mathbb{R}$$

So, considering the *Binomial Series Expansion* for non-integer exponents $(1+x)^d = \sum_{k=0}^{\infty} \binom{d}{k} x^k,\ d \in \mathbb{R}$, it is possible to define a *Finite Difference Operator for non-integer exponents* as well, such as:

$$\nabla^d = (1-B)^d = \sum_{k=0}^{\infty} \binom{d}{k} (-B)^k$$

Now, considering $\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$, it is feasible to use $\binom{d}{k} = \frac{\Gamma(d+1)}{k!\,\Gamma(d-k+1)},\ \forall\ d \in \mathbb{R}$, where $\Gamma(*) := (*)! \to \mathbb{R}$. However, since the Gamma Function is a *meromorphic function* when $(*) < 0$, a better option could be the *generalized definition of the binomial coefficients* instead, such as:

$$\binom{d}{k} = \frac{d(d-1)(d-2)\ldots(d-k+1)}{k!} = \frac{\prod_{i=0}^{k-1}(d-i)}{k!} = \prod_{i=0}^{k-1} \frac{d-i}{k-i}$$

So, replacing this term in our difference operator, it is possible to get:

$$\nabla^d = (1-B)^d = \sum_{k=0}^{\infty} \prod_{i=0}^{k-1} \frac{d-i}{k-i} (-B)^k$$

Simplifying, the ***Fractional Difference Operator*** will be:

$$\nabla^d = \sum_{k=0}^{\infty} w_{d,k}\, B^k\,,$$

$$\text{where}: \quad w_{d,k} = (-1)^k \prod_{i=0}^{k-1} \frac{d-i}{k-i} \quad \wedge \quad \nabla^d = 1 - d\,B + \frac{d(d-1)}{2!} B^2 - \frac{d(d-1)(d-2)}{3!} B^3 + \ldots$$

Thus, by applying the above definition in X, such as $\nabla^d x_t = \sum_{k=0}^{\infty} w_{d,k}\, B^k\, x_t$, it is possible to define the ***Fractional Differentiation Method for Time Series***:

$$\nabla^d x_t = \sum_{k=0}^{\infty} w_{d,k}\, x_{t-k}\ ,\ \text{where}: \quad w_{d,k} = \left\{1,\ -d,\ \frac{d(d-1)}{2!},\ \frac{d(d-1)(d-2)}{3!},\ \ldots,\ \frac{\prod_{i=0}^{k-1}(d-i)}{k!},\ \ldots\right\}$$

Finally, Lopez de Prado (2018) proposes an *Iterative Fixed-Window* method to update $\{w_{d,k}\}$ over the time series for a given d, such as:

$$w_{d,k} = -w_{d,k-1}\,\frac{(d-k+1)}{k} \qquad \text{for } k = \{0, \ldots, \infty\}$$

Particularly, we are looking for all the cases:

$$\left| w_{d,k} \right| > \tau\ , \quad \tau \to 0$$

Being τ := minimum weight (tolerance) to consider any data point as useful information to be preserved.

There is something important to mention before move on to the code implementation. Note that the initial standard definition works over the whole data-points in the time series, since $k = \{0, \ldots, \infty\}$. This means that it is going to generate several values for $\{w_{d,k}\}$ until it reaches the total length of the time

series. So, let's suppose the time series has $z+1$ elements. Due to $X = \{x_t, x_{t-1}, x_{t-2}, \ldots, x_{t-z}\}$, there is going to be $z+1$ elements for $\{w_{d,k}\}$ as well. This essentially leads to compute the scalar product of these two huge set of vectors $\{w_{d,k}\}$ and X, being the result the fractional differentiated time series. However, the latter will have a *negative drift*, since its first values (the oldest ones) will have an associated extremely small $w_{d,k}$ compared to the most recent ones. To avoid this effect, the *Iterative Fixed-Window* method allows to restrict the $\{w_{d,k}\}$ up to specific τ value, passing the fixed $\{w_{d,k}\}$ set of elements as a sort of moving window through all the data-points in the time series, being x_{t-b} the starting point, where b is now the length of $\{w_{d,k}\}$ such as $b < z+1$. In that sense, the transformed time series will have a length of $(z+1) - b$. Therefore, the greater value defined for τ, the smaller b and, thus, more data-points will end up the differentiated time series.

Being saying that, let's use Wolfram language to implement the method defined above.

Wolfram Implementation

First, let's start by implementing $\{w_{d,k}\}$, such as:

```
In[ ]:= WeightRule[diff_, threshold_] := Module[
          {d = diff, t = threshold, k = 1,
           wgtlist = {1.}, wformulae = (-#1[[-1]] * ((#2 - #3 + 1) / #3)) &},
          While[True, lastweight = wformulae[wgtlist, d, k];
           If[Abs@lastweight < t, Break[], k++];
           AppendTo[wgtlist, lastweight]];
          Return@Reverse@wgtlist]
```

The reader might notice that `wformulae` is the *Fixed-Window* method, that is iterated over the **`While`** function. Thus, with this functionality, it is quite simple to define the general *Fractional Differentiation* procedure, such as:

```
In[ ]:= FractionalDSeries[ts_TemporalData, d_, t_ : 0.0001] := Module[
          {timeSeries = ts, diffOrder = d, tolerance = t},
          FractionalDSeries::tol =
           "Argument `1` causes more weights than total observations. Please, increase it.";
          weights = WeightRule[diffOrder, tolerance];
          totalWeights = Length@weights;
          timeSeriesValues = QuantityMagnitude@timeSeries["Values"];
          If[Length@timeSeriesValues < totalWeights,
           Message[FractionalDSeries::tol, "'t'"] && Abort[]];
          idxDiff = Range[totalWeights, Length@timeSeriesValues];
          fdSeries = (weights.timeSeriesValues[[#[[1]] ;; #[[2]]]]) & /@
            Transpose[{Range[1, idxDiff // Length], idxDiff}];
          Return@TimeSeries[fdSeries,
            {timeSeries["Dates"][[totalWeights ;;]]}, TemporalRegularity → True];]
```

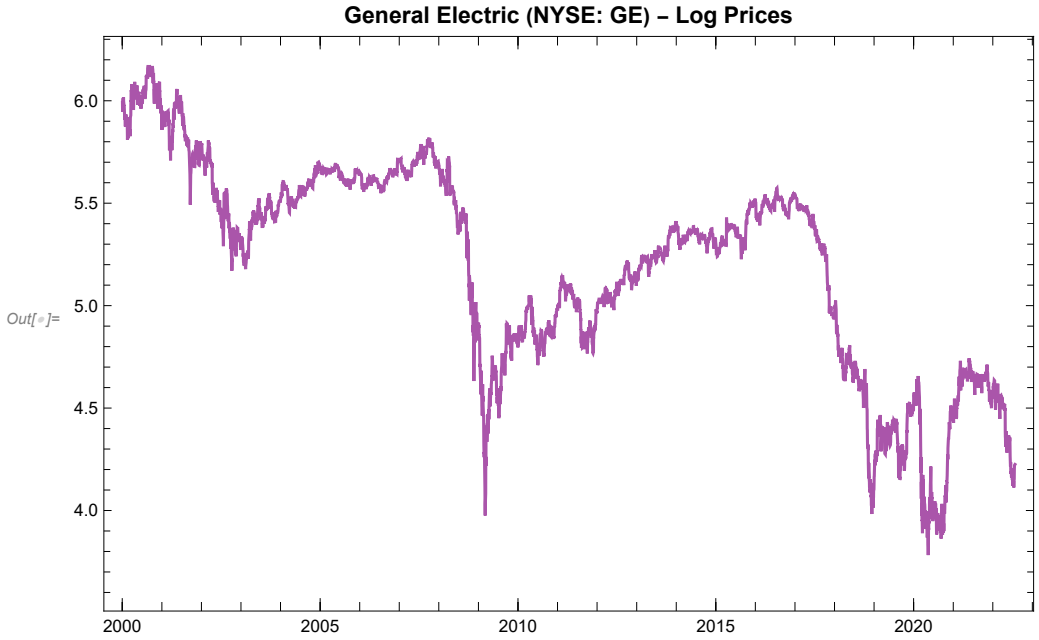
Notice that there is a predefined value of τ =0.0001 in this functionality. This value will not be changed in this project. However, in the conclusions, there is going to be some insights about further modifica-

tions of it, as well as some criteria to define it manually.

So, let's start by testing this function using some financial data; particularly, the time series of logarithmic prices of **NYSE: $GE** (General Electric Inc.) since 2000 until today. This looks like:

In[ ]:=
```
ts = Log@QuantityMagnitude@FinancialData["GE", "Close", "Jan. 1, 2000"];
DateListPlot[ts,
 PlotLabel → Style["General Electric (NYSE: GE) - Log Prices", Black, 12, Bold],
 PlotStyle → {Lighter@Purple}, ImageSize → 500]
```

Out[ ]=



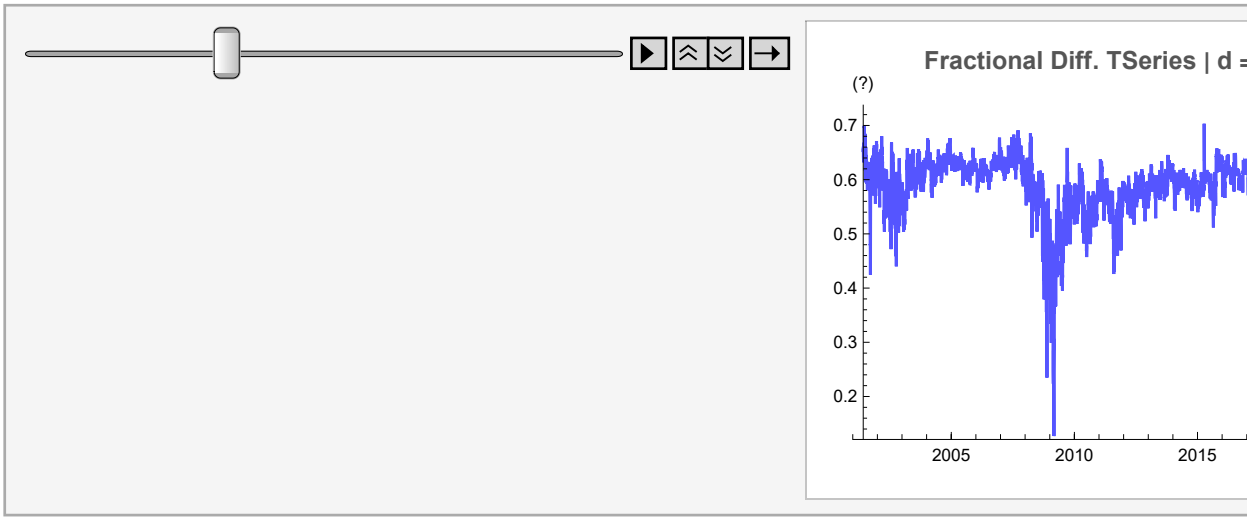
Now, by defining a specific set of {*d*} values between 0 and 1, it is possible to get a nice representation of the effects of the fractional differentiation procedure. Also, to see the consequences of the differentiation on the memory, an Auto-Correlation Function (ACF) is plotted with a maximum lag of 20 datapoints.

By testing all these values, it is possible to get:

In[ ]:=
```
baseDs = Range[0, 1, 0.005];
setFSeries = Table[{d, FractionalDSeries[ts, d]}, {d, baseDs}];
```

In[ ]:=
```
ListAnimate[Table[
  dTested = listfseries[[1]];
  tsData = listfseries[[2]];
  autoCorrData = CorrelationFunction[tsData, {1, 20}];
  minfseries = Min@tsData;
  maxfseries = Max@tsData;
  minACF = Min@autoCorrData;
  maxACF = Max@autoCorrData;
  Row[
   {ListLinePlot[
     tsData,
     PlotLabel → Style[
       "Fractional Diff. TSeries | d = " <> ToString@dTested, Lighter@Black, 13, Bold],
     PlotStyle → {Lighter@Blue},
     PlotRange →
      {If[Negative@minfseries, minfseries * 1.05, minfseries * .95], maxfseries * 1.05},
     ImageSize → 320, AxesLabel → {"Time", "(?)"}],
    ListPlot[
     autoCorrData,
     PlotLabel → Style["ACF | d = " <> ToString@dTested, Lighter@Black, 13, Bold],
     PlotStyle → {Lighter@Blue},
     PlotRange → {If[Negative@minACF, minACF * 1.2, minACF * 0.8], maxACF * 1.02},
     ImageSize → 320, Filling → Axis, AxesLabel → {"Lag", "AC"}]},
   "   "],
  {listfseries, setFSeries}],
 AnimationRunning → False, SaveDefinitions → True]
```

Out[ ]=



It is quite obvious that, when $d \to 1$, the correlogram shows less memory in the time series for different lags, in such a way that for $d = 1$ all memory is erased. Thus, the closer $d \to 0$, the better to the aim of memory preservation. Note also that the length of the differentiated time series is not the same for all the values of d. This is because of the effect of the *Iterative Fixed-Window* method. Finally, the reader might notice also there is a "(?)" character that defines the y-axis in the differentiated time series plot.

Just skip it by now, because it will be clarified at the end of this **Part I**.

Now, it is possible to pass a unit root test on each differentiated time series based on the {*d*} values computed above. A statistic (probability p-value) close to 0 would allow to reject H_0:= *presence of unit root*, and, hence, ensure the time series has good chances to be at least weakly stationary. At the same time, it is possible to use an ACF(1) as an indicator of the memory as long as the process get closer to some statistic values that can be useful to reject H_0.

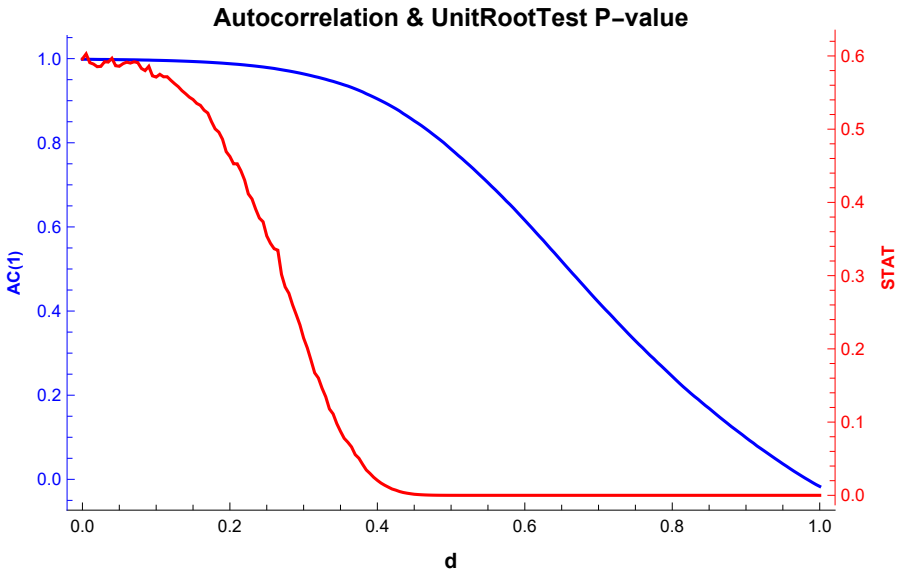
In[ ]:=
```
statACFDataPlot = {#[[1]], CorrelationFunction[#[[2]], 1], UnitRootTest@#[[2]]} & /@ setFSeries;
```

In[ ]:=
```
corrPlot = ListLinePlot[
    {#[[1]], #[[2]]} & /@ statACFDataPlot, ImagePadding → {{50, 50}, {45, 2}},
    PlotStyle → Blue, Frame → {True, True, False, False},
    FrameStyle → {Automatic, Blue, Automatic, Automatic},
    FrameTicks → {None, All, None, None},
    PlotLabel → Style["Autocorrelation & UnitRootTest P-value", Black, 13, Bold],
    FrameLabel → {{Style["AC(1)", Bold], None}, {Style["d", 11, Bold], None}},
    ImageSize → 500];

statPlot = ListLinePlot[
    {#[[1]], #[[3]]} & /@ statACFDataPlot, ImagePadding → {{50, 50}, {45, 2}},
    PlotStyle → Red,
    Frame → {False, False, False, True},
    FrameTicks → {{None, All}, {None, None}},
    FrameStyle → {Automatic, Automatic, Automatic, Red},
    FrameLabel → {{None, Style["STAT", Bold]}, {None, None}},
    PlotLabel → Style[" "],
    ImageSize → 500];
```

In[ ]:=
```
Overlay[{corrPlot, statPlot}]
```

Out[ ]=



So, the reader might notice there is no linear decrease in memory (blue line) for fractional differentiation orders. It tends to settle only when $d \to 1$, which is actually useful since there is a region between 0.4 and 0.6 in which it is possible to say there is a high level of autocorrelation and, at the same time, the H_0 could be rejected (red line), achieving the stationarity condition. Therefore, the criteria for select the optimal non-integer order of differentiation $d(*)$ will be the lower boundary of this region; in this example, a value of 0.40 approximately. Thus, generalizing this idea, this value should leads to the maximum preserved memory, since $d(*)<1$. However, note that $d(*)$ might slightly change based on the type of algorithm used (specially in its precision). In the long run, though, it should be essentially the same.

Now, using this criteria, it is possible to get the optimal $d(*)$ by the implementation of a binary search algorithm.

The Recursive Binary Search Algorithm to find d(*)

A binary search is a search algorithm that allows to find the position of a specific value in a sorted set of elements. Essentially, given the array A of q values, such as $A = \{A_1, A_2, A_3, \ldots, A_q\}$, where $A_1 \leq A_2 \leq A_3 \leq \ldots \leq A_q$, the index position of a target φ value could be found by following the pseudocode defined below:

```
function RecursiveBinarySearch [A, φ, min, max] is
    middlePoint := min + floor ((max - min) / 2)
    while min < max do
        if  φ < A[middlePoint]  then
            max :=  middlePoint + 1
            run RecursiveBinarySearch[A, φ, min, max]
        else if φ > A[middlePoint] then
            min:= middlePoint - 1
            run RecursiveBinarySearch[A, φ, min, max]
        else
            return  middlePoint
    return Error
```

The above is a recursive version of the original binary search algorithm, which is the one implemented in this project. Moreover, the value φ will be replaced by a significance level α, in such a way that the new condition will be **test[ts, middlePoint] < α**, being the function **test** the unit root test over the time series **ts** differentiated with **middlePoint** as the non-integer order. Notice it is not necessary to define a set or array of consecutive values for d, since it is feasible to define just the limits (say, 0 and 1) to initialize the searching process. Finally, to improve the computational search, it is a common practice to define a minimum convergence value between the new best answer (i.e., best d found) and the previous one, in order to cut the process in case the difference between both of them falls below this threshold. Thus, this considerations make possible to do the implementation as follows:

```
In[ ]:= FractionalDOSeries[ts_, lo_, hi_, sig_ : 0.05, t_ : 0.0001, climit_ : 0.0005] :=
 Module[{midOrderDif = N@Midpoint[{{lo}, {hi}}][[1]]},
  convergence = (hi - midOrderDif);
  fDSeries = FractionalDSeries[ts, midOrderDif, t];
  If[
   convergence < climit,
   Return[
    {midOrderDif, fDSeries, CorrelationFunction[fDSeries, 1]}],
   statValue = UnitRootTest@fDSeries;];
  Return[
   Which[
    statValue < sig,
    FractionalDOSeries[ts, lo, midOrderDif, sig, t, climit],
    statValue ≥ sig,
    FractionalDOSeries[ts, midOrderDif, hi, sig, t, climit]]];]
```

Note that the above implementation has a predefined $\alpha = 0.05$, and a convergence limit of 0.0005. Particularly, the former could be modified in case the user requires a more rigorous evaluation of the unit root test.

Thus, applying this function in the original time series with a $\alpha = 0.01$, it is feasible to get :

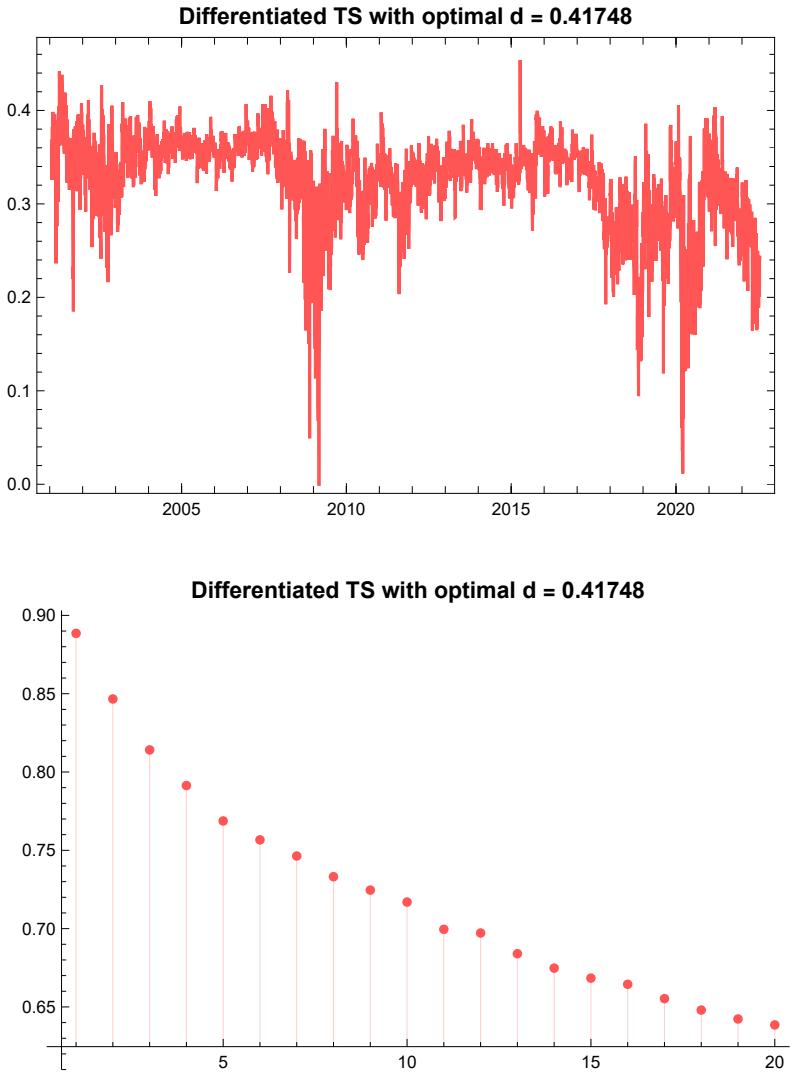
```
In[ ]:= optimalFracDSeries = FractionalDOSeries[ts, 0, 1, 0.01]
```

```
Out[ ]= {0.41748, TimeSeries[  Time: 22 Jan 2001 to 18 Jul 2022
Data points: 5407 ], 0.888517}
```

The function returns a list of three elements: the optimal d **(*)**, the differentiated time series, and its ACF(1). Plotting the transformed time series looks like:

In[ ]:=
```
Row[{DateListPlot[
   optimalFracDSeries[[2]],
   PlotLabel → Style["Differentiated TS with optimal d = "\
      <> ToString@ToString@optimalFracDSeries[[1]], Black, 12, Bold],
   PlotStyle → {Lighter@Red},
   ImageSize → 400
  ], ListPlot[
   CorrelationFunction[
    optimalFracDSeries[[2]], {1, 20}],
   Filling → Axis, PlotRange → All,
   PlotLabel → Style["Differentiated TS with optimal d = "\
      <> ToString@optimalFracDSeries[[1]], Black, 12, Bold],
   PlotStyle → {Lighter@Red},
   ImageSize → 400
  ]}, "      "]
```

Out[ ]=



There are several important considerations here. The first one is that the recursive binary search allows to get a nice precision in the required order of differentiation. This is important, since we can get a

better approximation of the optimal $d(*)$ without too much computational cost. Moreover, in addition to achieve the condition of invariant mean and covariance structure over time, the transformed time series has significance autocorrelation for different lags, from ACF(1) ≈ 0.88 until ACF(20) ≈ 0.64, with a smooth decay along them. This memory preservation condition makes possible to understand the slight trend in the differentiated time series (just compare the first value versus the last one). The reader can also compare this results with the traditional $d(1)$ by moving the console bar all the way to the right in the first dynamic plot computed above—the differences against the optimal $d(*)=0.41748$ are more than evident.

Finally, the reader might remember the "(?)" character of the first dynamic plot computed above. It was not a mistake—there is an important consideration related to it to keep in mind: ***what is the new differentiated time series about?*** This seems to be a silly question. However, since original time series is composed by a record of prices, a $d(1)$ means, consequently, a difference of prices. So, for instance, if someone wants to know the difference between what was paid some day and received back the next one, is more than enough to see this difference assigned to that specific period of time in the $d(1)$ differentiated time series. However, is it possible to make the same interpretation when $d(*)=0.41748$? Although this seems to be a minor consideration, this could have a considerable impact on the uses of this method in certain real applications. The project will recap this topic in the Conclusion Remarks section.

In the next part, the project will start by applying the optimal fractional differentiation procedure computed above in the context of Fractional Brownian motion simulations to understand the relationship between $d(*)$ and H, the Hurst Exponent.

Part II: The Fractional Brownian Motion (fBm) Experiment

An quick understanding of fBm processes and the Hurst exponent

Let's start by the formal definitions. So, considering the probability space $(\Omega, \mathcal{F}, \mathrm{P})$, any stochastic process $\{X_t : t \in \mathbb{R}^+\}$ could be defined as:

$$d X_t = a(t)\, d t + b(t)\, B_H(t)$$

Where $a(t) \wedge b(t)$ are the drift and diffusion coefficients respectively. Thus, a *Standard Brownian motion* process $B(t)$ will satisfy $dX_t \sim N(0, dt)$, which means that the random increments are independent because X_t is $\mathcal{F}_t$-measurable for $t \in \mathbb{R}^+$ in such a way that $\mathbb{E}[X_t \mid \mathcal{F}_s] = X_s$ for $0 \le s \le t$ (i.e., a martingale). So, to achieve this condition, the Hurst exponent H of the one-parameter Gaussian process $B_H(t) : t \in [0, T]$ must be $H = 1/2$ to the make its covariance matrix:

$$\mathbb{E}[B_H(t)\, B_H(s)] = \frac{1}{2}\left(|t|^{2H} + |s|^{2H} - |t-s|^{2H}\right)$$

Equals to $\mathbb{E}\left[B_{1/2}(s)\, B_{1/2}(t)\right] = \min(s, t)$, with an expected value of $\mathbb{E}\left[B_{1/2}(t)\right] = 0$ as well.

Thus, when $H \in [0, 1] - \{1/2\}$, the process is known as ***Fractional Brownian motion*** (fBm).

Now, what does all this mean? In plain English, for $H > 1/2$, the increments in the random process are positively correlated; in contrast, when $H < 1/2$, they are negatively correlated. Thus, the long-memory condition (more formally, the "long-range" dependence) arises in the former case, since the increments $X_t = B_H(t) - B_H(s)$ will show effectively a positive correlation in each new random step since $\mathbb{E}\left[B_{1/2}(s)\, B_{1/2}(t)\right] \neq \min(s, t)$. Therefore, the aim of the project is to evaluate the relationship between the Hurst exponent H for $1/2 \leq H < 1$, and the optimal $d(*)$ of that particular random process. The base hypothesis is that this relationship between them is linear (which is actually pretty obvious). However, **the particular interest of this project is to discover *empirically* the regions of values for $d(*)$ that might correspond to a given H, and how these varies as long as $H \to 1$**. Thus, the project will run n simulations for the same Hurst exponent H, applying the optimal fractional differentiation procedure on each of these simulated paths, and evaluating the set of optimal non-integer orders $\{d(*)_1, d(*)_2, \ldots, d(*)_n\}$ to assess its statistical properties.

Optimal Fractional Differentiation (OFD) on fBm simulations for H=1/2 (special case)

First, let's run 500 simulations by defining $H = 1/2$, i.e., by setting the *Standard Brownian motion*, also known as *Wiener Process*. Particularly, the drift and diffusion are defined with 0 and 1 respectively.

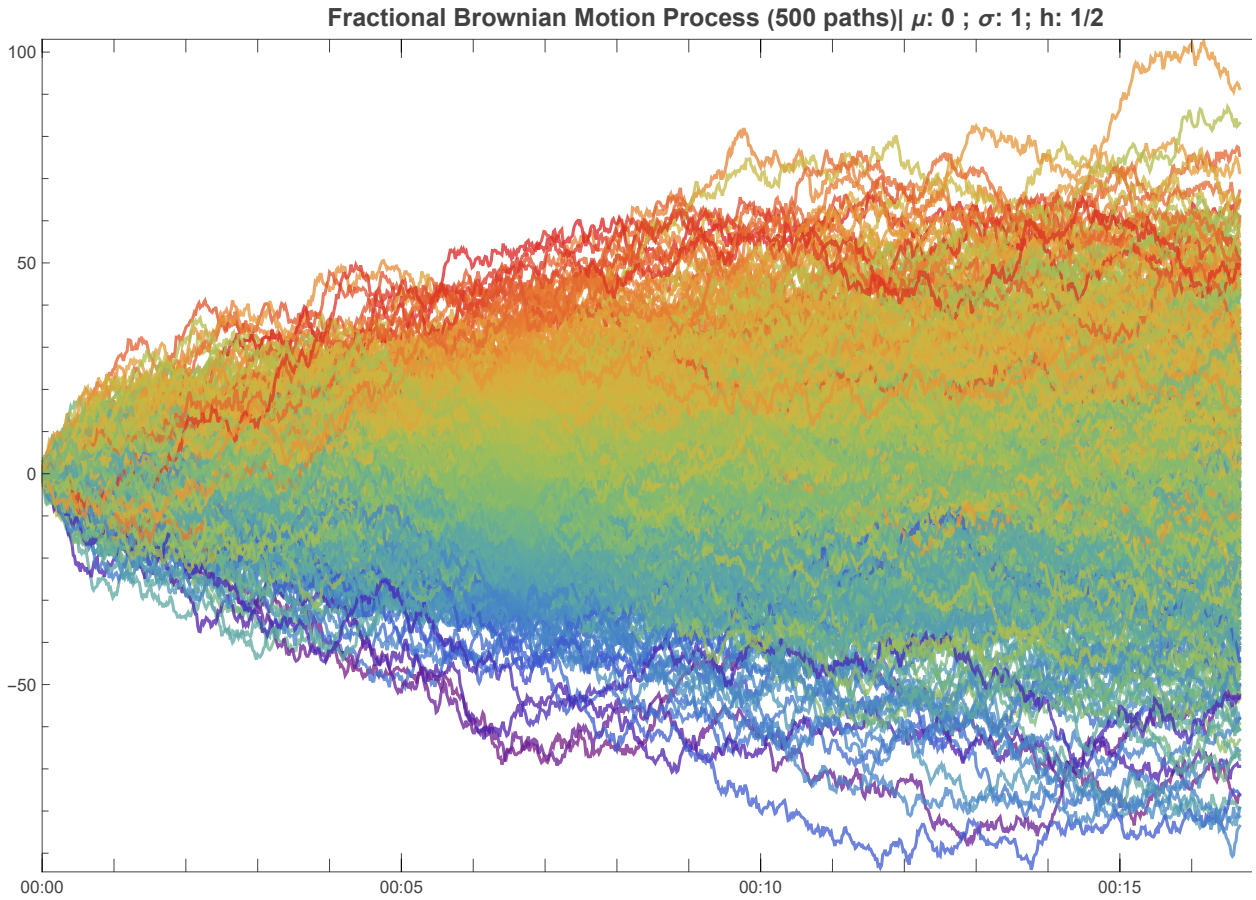
In[ ]:=
```
SeedRandom[15];
fBm = RandomFunction[FractionalBrownianMotionProcess[0, 1, 1 / 2], {1, 1000, 1}, 500];
```

Now, it is possible to plot each simulated path:

In[ ]:=
```
slicefBm = fBm["SliceData", 400];
slicedfBmPlot = BarChart[
      Last[#], Axes → False, BarOrigin → Left, AspectRatio → 6.85,
      ChartStyle → (ColorData["Rainbow"] /@ Rescale[
          MovingAverage[First[#], 2],
          {Min[slicefBm], Max[slicefBm]}, {0, 1}]),
      ImageSize → 63] &[HistogramList[slicefBm,
     {Range[Min[slicefBm], Max[slicefBm], (Max[slicefBm] - Min[slicefBm]) / 20]}]];
```

In[ ]:=
```
DateListPlot[fBm,
 PlotLabel → Style[\
    "Fractional Brownian Motion Process (500 paths)| μ: 0 ; σ: 1; h: 1/2",
   Darker@Black, 13, Bold],
 ImageSize → 720, Epilog → Inset[slicedfBmPlot, {1060, 0}],
 PlotRangePadding → {{0, 125}, {.5, .5}},
 BaseStyle → Directive[Thin, Opacity@0.75],
 PlotStyle → (ColorData["Rainbow"] /@ Rescale[slicefBm])
]
```

Out[ ]=



Notice how the expected value of the paths are essentially the same, since they come from a *Standard Wiener Process* in which effectively $\mathbb{E}\left[B_{1/2}(t)\right] = 0$. Now, let's see how this landscape changes by applying the **FractionalDOSeries** function over each path.

In[ ]:=
```
fdfBm = FractionalDOSeries[TimeSeries@#, 0, 1, 0.001] & /@ fBm["Paths"];
```

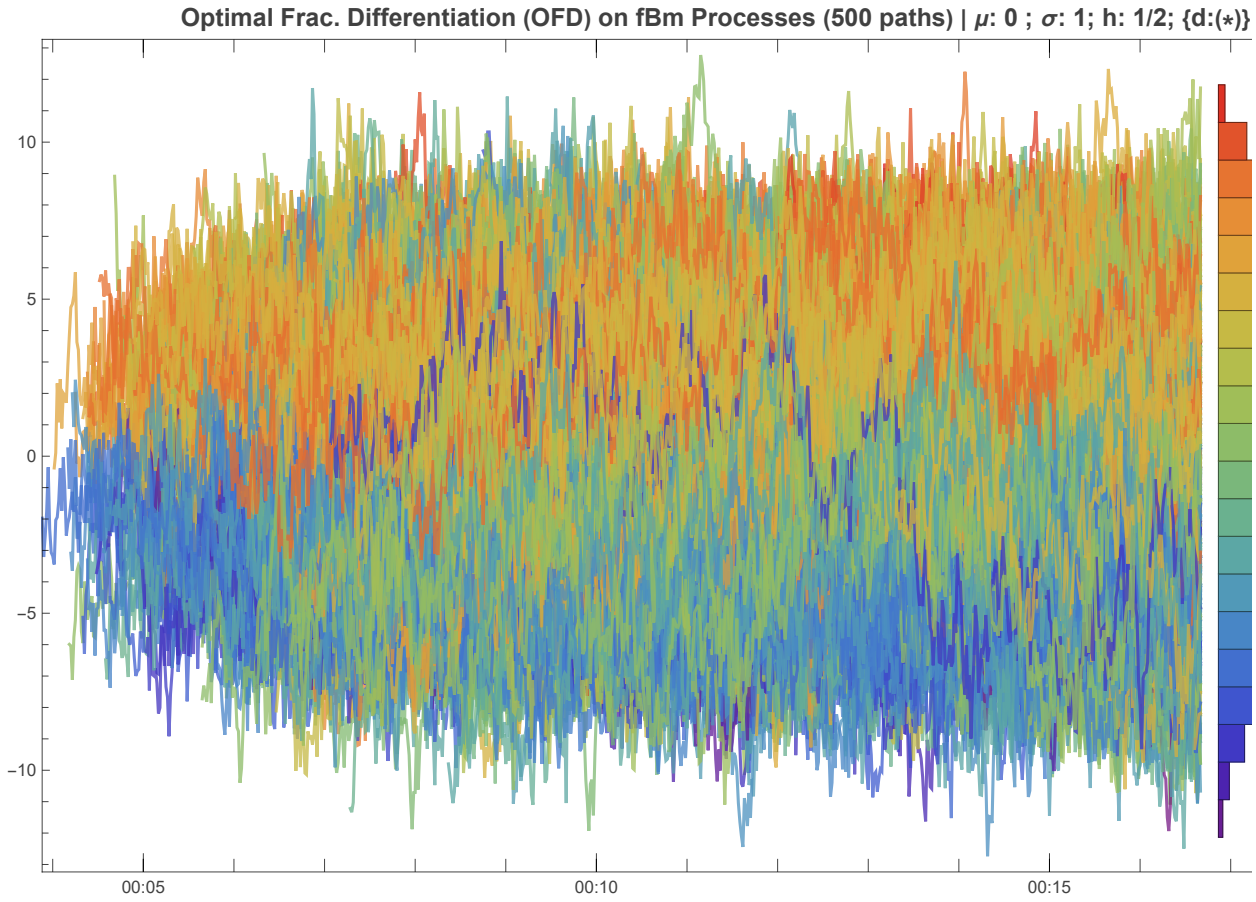
Now, let' s plot its results :

In[ ]:=
```
fdoptimals = #[[1]] & /@ fdfBm; fdseries = #[[2]] & /@ fdfBm; fdautcorrs = #[[3]] & /@ fdfBm;
```

In[ ]:=
```
tempFdfBmp = TemporalData@fdseries;
sliceFdfBmp = #["Values"][[400]] & /@ fdseries;
slicedFdfBmpPlot = BarChart[Last[#], Axes → False, BarOrigin → Left, AspectRatio → 6.85,
      ChartStyle → (ColorData["Rainbow"] /@ Rescale[MovingAverage[First[#], 2],
          {Min[sliceFdfBmp], Max[sliceFdfBmp]}, {0, 1}]), ImageSize → 63] &[
   HistogramList[sliceFdfBmp, {Range[Min[sliceFdfBmp], Max[sliceFdfBmp],
      (Max[sliceFdfBmp] - Min[sliceFdfBmp]) / 20]}]];
```

In[ ]:=
```
DateListPlot[tempFdfBmp, PlotLabel → Style[\
    "Optimal Frac. Differentiation (OFD) on
      fBm Processes (500 paths) | μ: 0 ; σ: 1; h: 1/2; {d:(*)}",
   Darker@Darker@Black, 13, Bold],
 ImageSize → 720, Epilog → Inset[slicedFdfBmpPlot, {1050, 0}],
 PlotRangePadding → {{0, 125}, {.5, .5}}, BaseStyle → Directive[Thin, Opacity@0.75],
 PlotStyle → (ColorData["Rainbow"] /@ Rescale[sliceFdfBmp])]
```

Out[ ]=

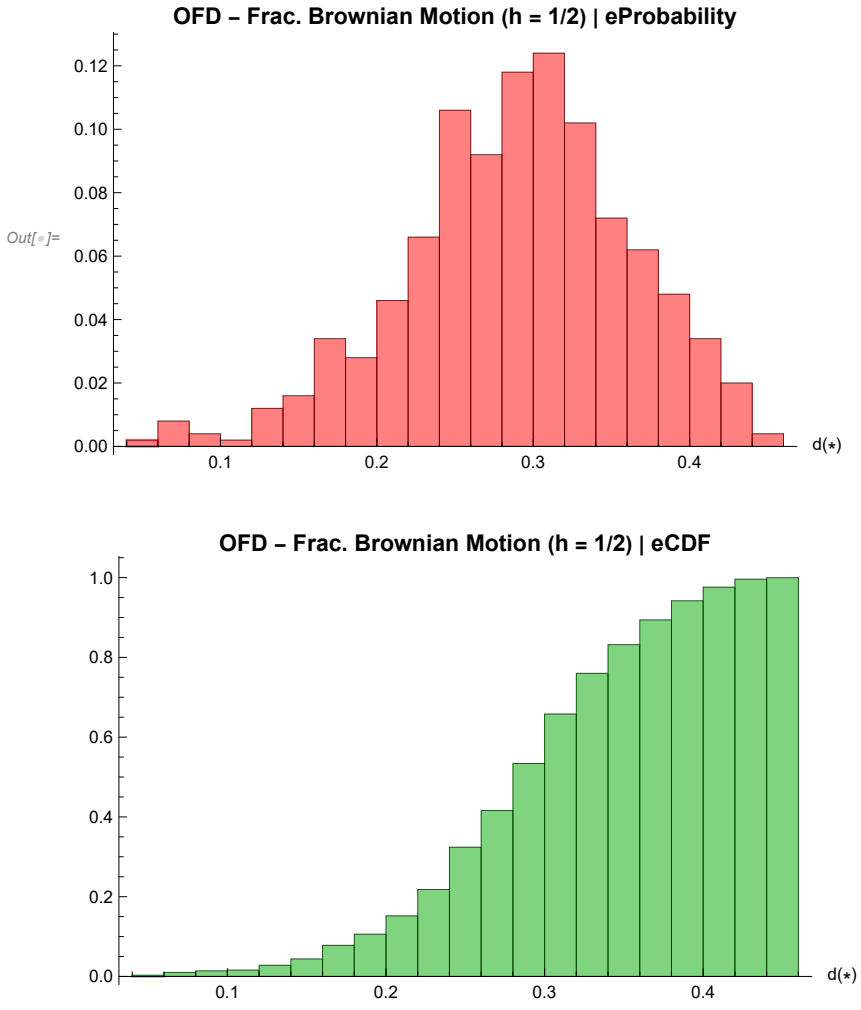


The results are interesting for two reasons: first, notice how, although {d (*)} has different values for each path, there is a sort of consistency between them, in such a way that the transformed time series fall within a specific range of differentiated values; and, second, the sample bar chart for the transformed simulations is now totally different from the previous one, essentially due to the fact that each path has a particular associated d (*).

Now, let's plot the empirical Probability distribution and CDF.

In[•]:=
```
Row[Table[Histogram[\
    fdoptimals, Automatic, info[[1]], PlotLabel →
    Style["OFD – Frac. Brownian Motion (h = 1/2) | e" <> info[[1]], Black, 12, Bold],
  ChartStyle → {Directive[info[[2]], Opacity[.5]]},
  ImageSize → 400, AxesLabel → {"d(∗)"}],
 {info, {{"Probability", Red}, {"CDF", Darker@Green}}}], "    "]
```

Out[•]=



These two plots are simple, but extremely informative. The first one (red one) allows to get the empirical probability distribution of the set of $\{d(*)\}$ values found for the given $H = 1/2$ in all the random paths. Based on that result, it is possible to say, for instance, that the expected value for $d(*)$ in a *Standard Brownian motion* process, with drift 0 and diffusion 1, could be around $d(*) = 0.3$. In the same matter, the second plot (green one) allows to get, for a specific $d(*)$ value, its probability to makes the simulated path weakly stationarity. Thus, based on that outcome, it is feasible to say that, empirically, a $d = 0.4$ can achieve the condition of invariant mean and covariance over time for (almost) any fBm process with $H = 1/2$, $\mu = 0$ and $\sigma = 1$.

Finally, let's extend this procedure for $1/2 \leq H < 1$ by defining $\{H\}$ as a set of values to test.

Optimal Fractional Differentiation (OFD) on fBm simulations for H ∈ [1/2, 1[

The first step is to define $\{H\}$. In this case, $\{H\} = \{0.5, 0.52, 0.54, \ldots, 98\}$, i.e, a set of 25 different values. In each of them, 100 simulations will be computed. In total, there will be 2,500 simulations to implement the **FractionalDOSeries** function.

Thus, defining the set $\{H\}$:

In[•]:=
```
stepH = 0.02; setHs = Range[0.5, 1, stepH][[ ;; -2]];
```

Now, computing the fBm simulations for each of these H values:

In[•]:=
```
SeedRandom[15];
fBmSimulations =
 RandomFunction[FractionalBrownianMotionProcess[0, 1, #], {1, 1000, 1}, 100 ] & /@ setHs;
```

Finally, passing the **FractionalDOSeries** function over each path, will be possible to get the results of the optimal fractional differentiation for each of them. Be aware this procedure could take some minutes (less than 10, though).

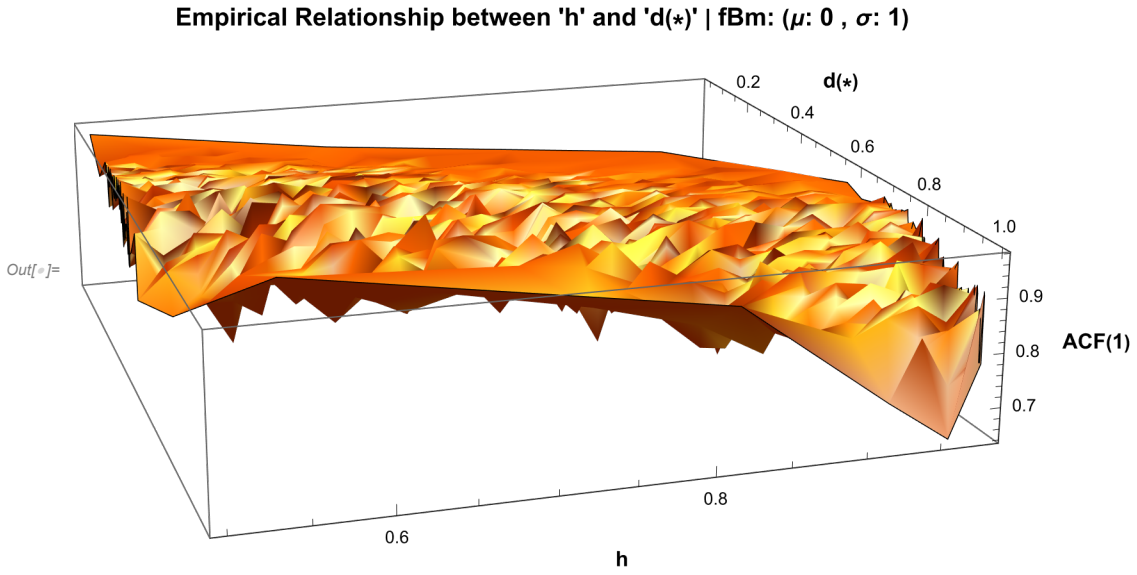
In[•]:=
```
fdofBmExperiment =
  (FractionalDOSeries[TimeSeries@#, 0, 1, 0.001]) & /@ (#["Paths"]) & /@ fBmSimulations;
```

Finally, with this information, would be very informative to deploy a 3D plot between $d(*)$, H and the ACF(1) achieved in this optimal fractional differentiation. In that sense, this plot looks like:

In[•]:=
```
fdoHExperiment = Table[#[[1]] & /@ x, {x, fdofBmExperiment}];
fdsHExperiment = Table[#[[2]] & /@ x, {x, fdofBmExperiment}];
fdaHExperiment = Table[#[[3]] & /@ x, {x, fdofBmExperiment}];
```

In[ ]:=
```
coordFdfBmLst = Flatten[#, 1] &@Table[
    Transpose[{
      fdoHExperiment[[idx]],
      Flatten@Transpose[{setHs[[idx]]}[[ConstantArray[1, Length@fdoHExperiment[[idx]]]]],
      fdaHExperiment[[idx]]}],
    {idx, Range[setHs // Length]}];
ListPlot3D[
 coordFdfBmLst,
 PlotStyle → Directive[Orange, Specularity[White, 10]],
 InterpolationOrder → 5, ImageSize → 550,
 AxesLabel →
  Table[Style[labelName, 11, Bold, Black], {labelName, {"d(*)", "h", "ACF(1)"}}],
 PlotLabel → Style[\
    "\n Empirical Relationship between 'h' and 'd(*)' | fBm: (μ: 0 , σ: 1) \n",
   Darker@Black, 13, Bold],
 Mesh → None, BoxRatios → {12, 12, 3}]
```

Out[ ]=

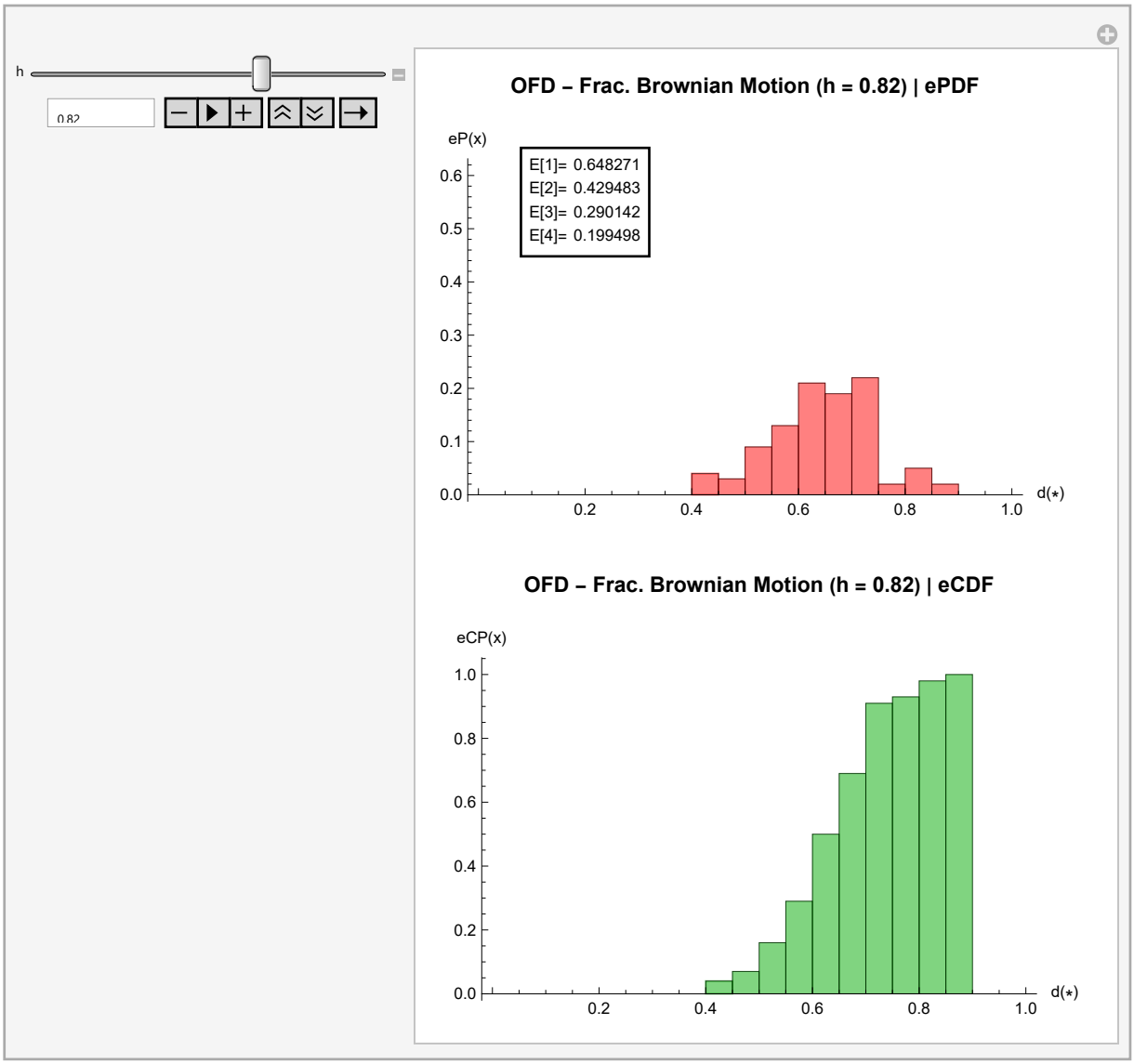


Finally, let's generate a dynamic plot to get its empirical statistical properties, such as:

In[ ]:=
```
fdHExpPlotFunction = (fdoHExperiment[[Position[setHs, #][[1]][[1]]]]) &;
```

In[ ]:=
```
Manipulate[
 dValues = fdHExpPlotFunction[h];
 statMom = Moment[dValues, #] & /@ Range[1, 4];
 Row[
  {Histogram[
    dValues, Automatic, "Probability",
    PlotLabel → Style["OFD – Frac. Brownian Motion (h = " <>
       ToString@h <> ") | ePDF\n", Darker@Black, 12, Bold],
    ChartStyle → {Directive[Red, Opacity[.5]]}, ImageSize → 350,
    AxesLabel → {"d(*)", "eP(x)"},
    PlotRange → {{0, 1}, {0, .6}},
    Epilog → Inset[Framed[
       Grid[{"E[" <> ToString@# <> "]=", statMom[[#]]} & /@ Range[1, statMom // Length],
        Alignment → {{Left}}], Background → White
      ], {.20, 0.55}]],
   Histogram[
    dValues, Automatic, "CDF",
    PlotLabel → Style["OFD – Frac. Brownian Motion (h = " <>
       ToString@h <> ") | eCDF\n", Darker@Black, 12, Bold],
    ChartStyle → {Directive[Darker@Green, Opacity[.5]]}, ImageSize → 350,
    AxesLabel → {"d(*)", "eCP(x)"},
    PlotRange → {{0, 1}, {0, 1}}]}, "   "],
 {h, setHs[[1]], setHs[[-1]], stepH, Appearance → "Open"}, SaveDefinitions → True]
```

Out[ ]=



There are a couple of considerations to mention before the final *Concluding remarks*. The first idea is based on the last dynamic plot: it is quite obvious that, as long as H increases, the empirical PDF/CDF for $d(*)$ changes, requiring a higher non-integer order of differentiation to achieve weak stationarity. This proves the empirical linear relationship between these two parameters. However, what seems to be even more interesting is the 3D plot, which shows the regions or areas that support this linear relationship between $d(*)$ and H. The reader may notice that there is a kind of diagonal behavior as H increases, such that, for instance, there is no possibility of achieving weak stationarity when $H = 0.9$ for $d(*) < 0.6$. Also, the 3D graph allows to see the memory decay when $H \to 1$, because in those scenarios a higher $d(*)$ is required and, therefore, more memory is erased.

Finally, in the general picture and for more extensive uses, this information could be useful in the context of time series modeling and stochastic analysis as a way to understand the implications of fractional differentiation over random processes.

Concluding remarks

Based on the previous results, the main conclusion is that, effectively, **there is a linear relationship between the optimal fractional order differentiation $d^{(*)}$ and the Hurst exponent H in the context of Fractional Brownian motion simulations**. Note that this conclusion works only when $1/2 < H < 1$, since they are precisely long-memory random processes. Moreover, the work carried out allows us to observe, specially in the 3D plot, how the relationship between these two parameters changes linearly, being a greater $d^{(*)}$ necessary when H approaches to 1. In addition, the effect of memory decay can be clearly seen in this upper limit of the random process ($H \to 1$), in which, since a larger $d^{(*)}$ is required to reach stationarity, more memory is eliminated. These conclusions can be useful for time series modeling and stochastic analysis, particularly by dealing with the memory-stationarity dilemma. Finally, there are some considerations that can be drawn from the methods, implementations and results obtained throughout the project. These are:

- *A typical unit root test (i.e., Dickey Fuller F test) is a good starting point to assess stationarity, but it is not enough*.

Keep in mind that the non-existence of a unit root does not necessarily mean that the evaluated time series will be stationary: it only implies that there is more or less strong evidence that allows us to postulate a stationarity condition as a *possibility*. Therefore, it is, perhaps, important to complement this evaluation with another statistical test, such as the *Phillips-Perron test* or the *ADF-GLS test*. This is important, since the results can vary considerably depending on how thorough the stationarity evaluation methods are to find the optimal $d^{(*)}$.

- *Getting results for d < 0 (fractional integration) in time series is possible, but...*

It is possible to make an extension of the proposed differencing method for d < 0 (i.e., integration). However, it is difficult to find a practical use in this case, because the stationarity-memory dilemma would no longer exist as the key problem to apply an optimization algorithm to find $d^{(*)}$. Additionally, the method for update $\{w_{d,k}\}$ will work differently: the rate of change between each $w_{d,k}$ will be much longer, requiring more elements to reach a given τ. Thus, any new integration procedures, despite being possible, should be formulated in a more generalized way.

- *The main drawback of this procedure is the lack of interpretability for the differentiated time series with* $0 < d^{(*)} < 1$.

This is probably the most relevant drawback of this method for real implementations. As already explained, a fractional differentiation procedure for time series does not allow to understand exactly the type of information available after the transformation. In some particular contexts, such as modelling, this could be a problem. For instance, considering the point of view of Explainable Machine Learning, working with this transformed time series as the base dataset would not contribute to being able to understand the results obtained by the modeling from a discretionary point of view, since, essentially, the new differentiated series is practically incomprehensible. This requires a delicate judgment of the practitioner to find the appropriate environment for the implementation of this

method.

- *Recursive binary search is a fashion way to find* d^**, but it is not the only way.*

Finally, it is important to mention the binary search is convenient due to the fact it just requires the definition of the search boundaries and the time series. However, more complex search procedures, or optimization algorithms as well, could be implemented in this matter. In this new case, the parameter τ could have an important role, since there is an option to find a sort of maximum value for it (which means, more data-point in the transformed time series) that allows to fit the stationarity condition, as well as preserving as much as memory as possible. Thus, now there would be 2 key parameters, d^* and τ, to be evaluated.

Keywords

- Fractional Differentiation
- Fractional Brownian motion
- Stochastic analysis
- Time Series

Acknowledgment

The author wishes to express his immense gratitude to all the people who made the development of this project possible. First of all, to Gosia Konwerska, for her kindness, support and brilliant ideas on this subject that were useful to accomplish the goal of the program—definitely, this project would not have gotten to where it is without her. Likewise, the author would like to express his gratitude to Oleg Marichev, for his courtesy and brilliance, as well as for being an inspiration in the deeper branch of this subject that the author hopes to learn. Likewise, the author also wishes to express his gratitude to his mentor Jesse Galef, for providing a practical and adequate guidance for an appropriate development within the program. Last but not least, the author expresses his enormous gratitude to Stephen Wolfram, for having insisted (and how right he was!) on the development of this topic as a project. Of course, the author extend this thanks to the entire team behind the organization of the WSS22, especially to Mads Bahrami, for being the architect who allowed the author to have this amazing opportunity.

References

- Lopez de Prado, M. (2018). *Advances in Financial Machine Learning*. Wiley, New Jersey, (75-88).
- Samko, S., Kilbas, A. and Marichev, O. (1993). *Fractional integrals and derivatives: theory an applications*. Gordon and Breach Science Publishers, Philadelphia, (7-23).

- Walasek, R. and Gajda, j. (2021). “Fractional differentiation and its use in machine learning”. *International Journal of Advances in Engineering Sciences and Applied Mathematics*, Vol.13, No. 3, pp. 270-277. https://doi.org/10.1007/s12572-021-00299-5
- Dai, W. and Heyde, C.C. (1996). “Ito’s formula with respect to Fractional Brownian motion and its application”. *Journal of Applied Mathematics and Stochastic Analysis*, Vol. 9, No. 4, pp. 439-448. https://doi.org/10.1155/S104895339600038X