An efficient broadband deep memory algorithm for computing fractional order operators

Steven Dorsher[1,2a], Gary Bohannan[b]

[a]*Department of Chemistry and Physics, Saint Cloud State University*
720 4th Avenue South, WSB 324, Saint Cloud, Minnesota, 56301, USA
1-952-686-1925
sdorsher@gmail.com
[b]*Department of Chemistry and Physics, Saint Cloud State University*
720 4th Avenue South, WSB 324, Saint Cloud, Minnesota, 56301, USA
gwbohannan@stcloudstate.edu

Abstract

This paper outlines a method to achieve effective bandwidth of five or more decades in the approximation of a fractional order derivative. This constitutes an increase of two decades or more in constant-phase bandwidth (to within 10° of phase ripple) over current algorithms, such the Infinite Impulse Response (IIR) method based on continued fraction expansion, while demanding only slightly more computational steps and processor memory.

Keywords: fractional calculus, numerical methods, computational efficiency
2010 MSC: 65Y02, 65Z02, 65Y04, 65Z04

1. Introduction

Interest in the application of the fractional calculus has been growing at an ever increasing rate. Of long term and continued interest is in the use of fractional order (FO) operators, such as integrators and differentiators, in motion control applications. [1] While wide bandwidth analog controllers have been successfully demonstrated [2], fractional order analog circuit elements are not

[1]Corresponding author
[2]Mailing address: 3152 Blackheath Drive, Saint Cloud, Minnesota 56301

generally available and the prototypes that have been demonstrated do not have the ability to be retuned for a specific desired phase. [3]

Unfortunately, the existing techniques for digital approximation of FO operators are limited in bandwidth to on the order of three and a half decades of frequency response while nonlinear effects in motion control systems can span five decades or more. This places a severe constraint on the design and implementation of digital FO controllers, i.e. how to set the sampling frequency to meet the high speed requirements necessitated by the Nyquist sampling rate while at the same time providing enough deep memory to get low offset error. Additionally, the infinite impulse response type of implementation cannot guarantee stability due to the limitations of finite precision arithmetic. See e.g. [4].

Given the current necessity to implement FO controls in digital form, it is desirable to obtain the most efficient algorithm to compute a fractional order operator while maximizing the numerical stability of the algorithm. Efficiency to be measured in both memory utilization and number of computations per time step. This paper outlines a computational method inspired by the Riemann-Liouville integral definition and the Grünwald summation formula. [5] The essential concept is the rescaling of time by successive accumulation of older data into increasing size bins for deeper memory.

Our motivation is two–fold. First to introduce the idea of "time scaling" for computations requiring coverage of very long time spans, and second to offer an algorithm that might be used in a microprocessor based control unit (mcu) requiring flat phase response over a wide bandwidth. One might wonder why the attempt at such extended bandwidth since integer order derivative and integral operator will typically either decimate or expand the amplitude beyond the range of analog to digital converters due to the $\omega^{\pm 1}$ scaling. Fractional order scaling is much "softer" in the sense that the magnitude scaling only goes as $\omega^{\pm\alpha}$ with the fractional order $|\alpha| < 1$, allowing for effective operation over a much wider bandwidth.

Outline

. We will first describe a novel approach based on successive binning of older data and resource requirements for the algorithm, then we will compare the results with other approaches.

- Section 2 contains the definition of the algorithm.

- Section 3 provides an analysis of the computational resources required for the larger versus the smaller number of memory registers required.
- Section 4 provides a description of the numerical simulation written to verify the algorithm described in Section 2.
- Section 5 describes the amplitude and phase response of these algorithms for a large and small number of bins in the partition.

2. Average Grünwald Algorithm

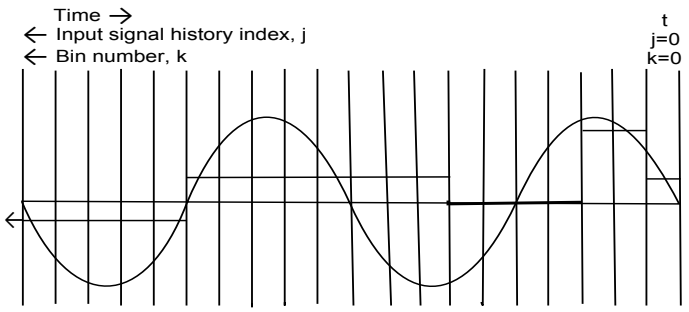
We will be comparing our results with the Infinite Impulse Response (IIR) method based on the continued fraction expansion approximation to s^α. [4] Its benefits are that it has a flat phase response over approximately two and a half decades in phase for a 9th order expansion (10 registers of input signal memory). It would be desirable to find an algorithm with an even broader flat phase frequency bandwidth and with a comparably small amount of memory required. We take the Grünwald algorithm as a starting point. As the signal history retained in the Grünwald sum grows longer, the bandwidth of flat phase response grows broader; however, the memory required also increases with input signal history length. To reduce this memory requirement and retain a long signal history, we propose partitioning the input signal history into bins that are longer further into the past. The value of the binned input signal is represented by the average value of the input signal within that bin. The presence of short bins at recent times maintains sensitivity to high frequencies, while the inclusion of long bins at past times adds sensitivity to low frequencies that would not be present in a truncated Grünwald sum with the same number of terms. (See Figure 1).

2.1. Modified Grünwald

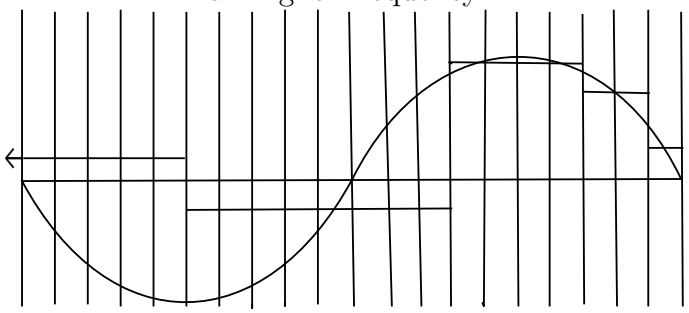
The Grünwald form of the fractional derivative of order α can be written [5]

$$ {}_0D_t^\alpha f(t) = \lim_{N_t \to \infty} \left(\frac{t}{N_t} \right)^{-\alpha} \sum_{j=0}^{N_t - 1} w_j x_j \tag{1} $$

where $f(t)$ is the input signal at time t and the jth value of the input signal history is $x_j = f\left(t - \frac{j\Delta t}{N_t}\right)$. Note that x_0 is the value at the current time and



a. Higher frequency



b. Lower fequency

Figure 1: Let each division represent a single time step and each horizontal bar represent a bin. Larger amounts of time are included in bins with increasing time indices (further into the past). The value of the bin is the average of the input signal (the sine wave) at each time index within that bin. There exists some oldest bin that still responds sensitively to the input signal. This is bin number one (counting from zero) for Figure a and bin number two for Figure b. Since the time step (grid size) and binning structure are held fixed, at lower frequencies, the oldest sensitive bin moves further into the past. This illustrates the sensitivity of short bins at recent times to high frequencies and the sensitivity of long bins in the more distant past to low frequencies. This scaling relation motivates our binning strategies. All of the binning strategies used in this paper, with the exception of the straightforward truncated Grunwald sum, have bins that increase in duration further into the past, in order to achieve greater sensitivity at low frequencies.

Abbreviations and notation used this paper:

IIR	– Infinite Impulse Response
FO	– Fractional Order
GL	– Full Grünwald calculation
GL#	– Truncated Grünwald calculation to # terms
LIN#	– Linear binning, # bins
SQ#	– Squared binning, # bins
EXP#	– Exponential binning with a base of two, # bins
FIB#	– Fibonacci binning, # bins
N_b	– number of memory bins
N_h	– full history capacity of the algorithm
N_t	– number of data items up to time t

increasing indices of x correspond to more distant times in the past. The jth Grünwald weight is

$$w_j = \frac{\Gamma(j-\alpha)}{\Gamma(j+1)\Gamma(-\alpha)}. \tag{2}$$

For the purposes of this paper, the order will be constrained $-1 < \alpha < 1$. Note that a fractional order integral is simply a derivative of negative order. Both derivatives and integrals are computed over a finite interval. It is assumed that the system is non-initialized; that is there are no initial conditions.

To include more distant history at low computational cost, we modify the Grünwald sum of Equation 1 by partitioning its history into N_b bins. In each bin k, the input signal history x_j is represented by its average value over that bin, X_k.

Since we make the assumption that each value is well represented by its average within a bin, we can define a value for the "bin coefficient" by summing the Grünwald coefficients within that bin.

$$W_k = \sum_{j=p_{k-1}+1}^{p_k} w_j \tag{3}$$

where w_j is summed from the lowest index of the input data history within

bin k to the highest index p_k within that bin. Define

$$N_h = p_{N_b} = \sum_{k=0}^{N_b} b_k, \tag{4}$$

for later reference. Here b_k is the number of time indices within the kth bin.

With these definitions, the modified Grünwald differ-integral can be written

$${}_0D_t^\alpha f(t) = (\Delta t)^{-\alpha} \sum_{k=0}^{N_b} \bar{W}_k X_k \tag{5}$$

where Δt is the interval between time samples. There is an additional factor that goes into $\bar{W}$ that will be discussed in Section 2.2. Note that in the average Grünwald algorithm, N_h plays the role of the last virtual time element to effectively enter the modified summation, rather than N_t. Here, N_t plays the role of the number of time steps that have occurred since the algorithm started to run.

2.2. Updating the average input signal history (bin values)

As time steps forward, the input signal history x_j and its binned counterpart, the average input signal history X_k, must change. The zeroth element always corresponds to the current time, so the origin of these histories essentially slides along the time axis as the calculation proceeds. When a new input data element is read, the values of the average input signal history (X_k) are updated. The new data element is transferred into bin zero through a weighted average, updating the value of the average input signal history, X_0, stored in the first bin. Since data elements represent values read at time steps, they should be incompressible– when an element leaves bin zero and enters bin one, if bin one is full, another virtual element should be pushed from bin one into bin two, and so on. This process continues until a bin that is partially full or empty is reached. For an already filled bin, to update the average data stored in the kth bin, we take the weighted average obtained by adding one virtual element from the $(k-1)$th bin to the $b_k - 1$ identical virtual elements remaining in the kth bin, where b_k is the bin capacity. For an unfilled bin, the virtual element that has just been shifted into the bin is averaged with the c_k identical virtual elements already in the bin. This process is illustrated in Figure 2.

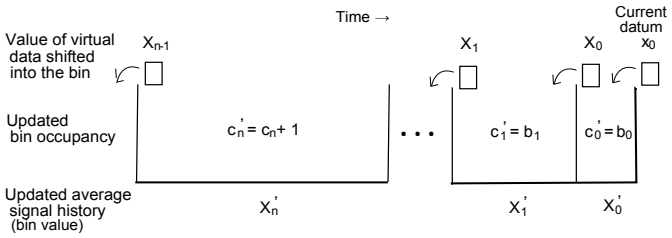


Figure 2: When the input datum x_0 is read at the current time, virtual data elements with average input signal history values X_k are shifted from the kth to the $(k+1)$th bin in the binned input signal history. The first $n-1$ bins are at capacity, but the nth bin gains one data point. The bin averages are updated to value X'_k through a weighted average given by Equation 6. At the next time step, the whole structure will step forward along along the time axis and read the next datum in the sequence, pushing back a new chain reaction of virtual data. Since bins increase in duration at higher time indices and more virtual data elements of identical values share a bin, the influence of an individual datum is diluted as it is shifted further back into memory.

During start-up, it will be necessary to consider a bin that has some set size b_k, but is not filled to that capacity. In that case, it is the current occupation number c_k of that bin that enters the calculation. The algorithm for updating bins can be stated generally if the occupation number of the kth bin is defined to be c_k. In that case, $c_k = b_k$ for all the bins more recent than the bin that is in the process of filling. No updating occurs for older bins.

If the kth bin initially contains c_k elements, updating the history either leaves c_k ($c'_k = c_k = b_k$) or increments the number of elements in the bin such that $c'_k = c_k + 1$ if the bin is not yet at capacity.

Taking into account both weighted average and capacity effects, when a new virtual element is shifted into the kth bin, the bin's new value X'_k is given by

$$X'_k = \frac{c'_k - 1}{c'_k} X_k + \frac{1}{c'_k} X_{k-1}. \tag{6}$$

where X_k is the old value of the kth bin and X_{-1} is taken to be x_0, the input data that has just been read.

During start-up, the partially full bin also factors into the Grünwald weights. To handle the bin that is partially full, we weight the binned Grünwald weights by the ratio of the bin occupation number c_k to its capacity b_k,

$$\bar{W}_k = \frac{c_k W_k}{b_k}. \tag{7}$$

3. Computational efficiency

The computational efficiency of each algorithm concerns both the memory used by the algorithm and the number of computational steps required for the algorithm to execute, which is a measure of the speed. To characterize the operational efficiency of a digital fractor operating as a circuit element in real time, the quantities of interest are the memory or number of processor steps *per time step*.

There are three phases of the computation that factor into the computational cost of the binned Grünwald algorithm: initialization (including summing the binned Grünwald weights), updating the average values of the stored history in the bins as new input data is read, and summing the modified Grünwald differ-integral.

During initialization, several things must happen. The array defining the binning structure, b_k, must be set. The history of the input signal must be zeroed and the first output must be set to zero. Most significantly, the binned weights must be calculated for the chosen binning structure.

The time it takes to calculate or set b_k is moderately dependent upon the details of the binning structure chosen. However, any binning structure will have a processing time that scales as N_b, the number of memory storage bins, rather than N_h, the history capacity, because N_b is the number of bins that must be set.

On the other hand, since each of the N_h individual Grünwald weights must be summed, the time for initialization of the weights scales as N_h. Typical values of N_h are of order $1,000$ to 10^6. Typical values of N_b are 10 to 26. Since $N_h >> N_b$, the bulk of the initialization process occurs during the computation of the weights and the rest of the initialization process may be neglected. The time for initialization as a whole scales as N_h.

By the very design of the algorithm, there is no need to store the full Grünwald weights. The largest arrays that need to be stored are the binned weights and the bin capacity, each of which has size N_b. As a result, memory scales as N_b rather than N_h. Equation 3 gives the algorithm for the calculation of the bin weights. The following recursion formula for Grünwald weights can be used to iterate w_j to a new value with each time step. In the jth time step, w_j has the value

$$w_j = \frac{j - 1 - \alpha}{j} w_{j-1}. \tag{8}$$

In the implementation of an algorithm for initialization of the weights, individual Günwald weights w_j need not be stored in an array after calculation. These individual weights can be summed immediately into the corresponding bin weights W_k according to Equation 3, at which time the value of w_j may be safely overwritten with w_{j+1} in order to minimize the amount of memory required such that it may be possible to later migrate the code to an mcu.

When new input data is read, the time required to update the binned history values scales as at most N_b because there are N_b bins that must be updated. Equation 6 may be used as the basis for an algorithm to accomplish this goal. Care must be taken to account for the filling of the bins until the approximation reaches steady-state.

This time a memory saving and speed enhancing opportunity is available. Since only one bin can be filling at a time and the rest must be either full ($c_k = b_k$) or empty ($c_k = 0$), the full array of N_b occupancy numbers c_k need not be stored. Instead, it is sufficient to store the index of the bin that is filling, $k_{filling}$ and its occupancy, $c_{filling}$. These unfilled bins with $k > k_{filling}$ and $c_k = 0$ are not updated when a new input signal value is read into the history. With this approach, the algorithm becomes

$$X'_k = \frac{c_{filling} - 1}{c_{filling}} X_k + \frac{1}{c_{filling}} X_{k-1} \tag{9}$$

for $k < k_{filling}$ and

$$X'_k = \frac{b_k - 1}{b_k} X_k + \frac{1}{b_k} X_{k-1} \tag{10}$$

for $k = k_{filling}$ with no shift occurring for $k > k_{filling}$ since there are no elements in those bins. Since no computation occurs for $k > k_{filling}$, the updating algorithm truly scales as $k_{filling}$ per time step, which is at most N_b.

While it is possible to achieve a memory savings by storing only $c_{filling}$ and $k_{filling}$, in the C++ code reported in this paper, the full array of c_k is stored. This does not impact the memory scaling of the updating algorithm. Updating the stored bin history values scales as N_b in memory because the largest arrays that need to be stored are the binned average values, the bin capacity, and the bin occupancy, each of size N_b.

However, we do implement the speed enhancing aspect of the algorithm. In the code reported in this paper, the calculation is truncated at $k_{filling}$ rather than looping over the full N_b bins to obtain the $k_{filling}$ scaling in time.

The two real-time operation stages, updating and differ-integrating, together require less than 400 flops for 26 bins of history. For 26 bins, less than 100 memory registers are required. Initialization, on the other hand, is very resource-consuming. N_h may be larger than a million and still not fill 26 bins. Initialization therefore may require more than a million flops.

The binned Grünwald algorithm may be written in terms of either the number of elements per bin, b_k, or in terms of the maximum time step p_k included within each bin (see Equation 3). While we explain our algorithms in terms of b_k in this paper, the results derived from our C++ code were obtained using the p_k method.

4. Validation methods

While the algorithms described in Section 2 are ultimately intended for use in a digital fractor, in order to characterize their frequency response we have simulated them in C++ on a personal computer. The relative phase as a function of frequency is based upon the fractional derivative or integral of sinusoidal inputs, sweeping from low frequency to high frequency in logarithmic steps. Relative phase is reconstructed by curve fitting (adapted from Numerical Recipes in C [6]), using the last half of the output signal to the sine and cosine components of a sinusoidal signal with the same frequency as the input signal. From that, phase and amplitude are calculated.

To verify phase accuracy, we examine the phase shift of a single crossing in the time domain for an input frequency of 0.7924 Hz (Figures 3). Table 1 shows a comparison of reconstructed phases using the sinusoidal fit method and phases obtained from Figure 3 by measuring the phase difference in a single zero-crossing. The results demonstrate that the reconstructed phases are accurate when the output phase is within about ten degrees of the expected phase. Since we are interested in the deviations of the phase from the expected phase by about ten degrees, this should be sufficient. Transients may account for the remainder of the discrepancy in phase in the cases where the single crossing phase and the fit phase differ more dramatically.

The simulation has successfully been run for up to $1,000,000$ time steps, limited primarily by memory considerations due to the storage necessary

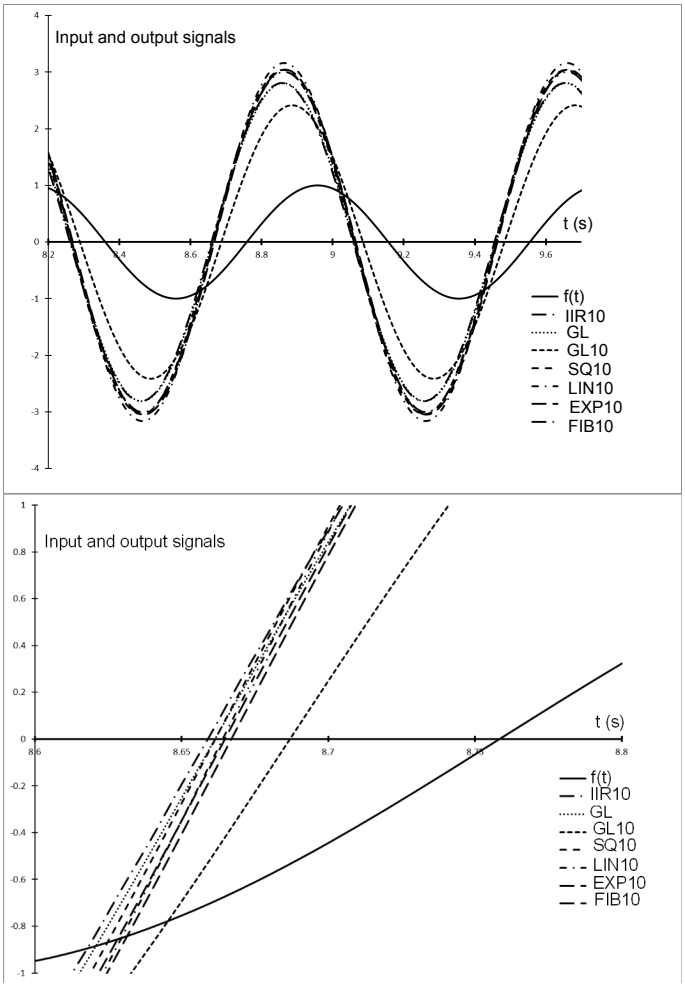


Figure 3: Visual measurements of phase of a 0.5 order derivative of a sine wave input signal with a frequency of 0.7924 Hz, using a variety of fractional derivative algorithms. Phase was evaluated by measuring the separation in zero-crossings and dividing by the period of the input sine wave, assumed to be the same as that of the output signals. Phases reconstructed using the sinusoidal fitting algorithm and phases reconstructed using the zero-crossing method are shown in Table 1. The duration of the simulation shown in this figure was was 10.0 s and there were 1000 time steps such that $\delta t = 0.01$ s. There were 10 terms in the IIR and 10 bins in all binned Grünwald approximations. A variety of binning methods were used in the average Grünwald algorithm: truncated simple Grünwald (GL), linearly increasing bin sizes (LIN), bin sizes increasing as a square (SQ), exponentially increasing bin sizes (EXP), and bin sizes following a Fibonacci rule (FIB). The lower plot shows a zoomed in view around the zero-crossings of the output signals, which were assumed to share the same frequency as the input signa for the purpose of measuring the phase using relative zero-crossing positionsl.

Name	Bin capacity (b_j)	Reconstructed (deg)	Zero-crossing (deg)
IIR40	-	45.0	45 ± 2
GL	$1, 1, 1, \ldots, 1 \times \infty$	44.2	41 ± 2
GL26	$1, 1, 1, 1, \ldots$	23.0	32 ± 2
LIN26	$1, 2, 3, 4, \ldots$	46.1	41 ± 2
SQ26	$1, 4, 9, 27, \ldots$	44.4	41 ± 2
EXP26	$1, 2, 4, 8, \ldots$	42.1	41 ± 2
FIB26	$1, 1, 2, 3, \ldots$	43.0	41 ± 2

Table 1: Phases reconstructed through sinusoidal fits and phases approximated through single zero-crossings of a single frequency (0.7924 Hz) based upon Figure 3, with an expected phase of 45°.

for phase and amplitude reconstruction. At this long simulation duration, amplitudes and phases over six decades in frequency are obtained.

5. Results

In a typical instance of the current state of the art algorithm, the Infinite Impulse Response (IIR), the expansion depends upon the last ten elements in the input signal history [4]. To make a fair comparison between fractional derivative or integral algorithms, it we examine the average Grünwald algorithm with ten bins of input signal history, such that the history memory requirement is the same. Figure 4 contains a bode plot for a fractional derivative of order $\alpha = 0.5$ with ten history registers. Compared to the IIR10, the average Grünwald with an exponential binning structure (EXP10) has approximately a factor of three gain in constant-phase bandwidth with a ten degree variation allowance.

The performance of the average Grünwald algorithm improves dramatically with a small increase in the number of input signal history bins. Computational limitations were the driving reason behind the choice of the maximal number of input history registers. It was not feasible to run the simulation for more than 10^6 time steps due to limitations in phase and amplitude reconstruction. Since the exponential binning structure (EXP) scales most rapidly with history depth, it set the limit on the number of bins that would fill during this number of time steps. Since $\log_2(10^6) \approx 20$, 20 bins of input signal history were included for all average Grünwald binning structures. Figures 5 through 8 compare 20 history memory registers of the IIR to 20

history memory registers of the average Grünwald, for a variety of binning strategies. For both fractional derivatives and integrals there is a gain in the constant phase bandwidth of 2 to 3 decades with a variation of ten degrees allowed.

The specific limitations encountered in generating Figures 5 through 8 would not be present during runtime on a microcontroller unit (mcu) because there would be no need to store and manipulate the entire past output signal history. However, other limitations might be encountered. The wide bandwidth of the EXP20 algorithm requires a higher sampling frequency than used in the simulation, to prevent aliasing, for instance. Additionally, we found that it became difficult to initialize the weights in finite time for more than 26 bins, for instance. Some less well motivated partitioning schemes (not reported here) appeared to go numerically unstable at a lower number of bins, as well.

6. Conclusions

In order to access deep memory while simultaneously conserving computational resources, we have modified the Grünwald algorithm by partitioning it into bins. In these bins, only the average input signal values and summed Grünwald weights are stored– the individual values are discarded to conserve memory and to make the computation more efficient. We have implemented this algorithm in C++ on a desktop computer.

Numerical simulations demonstrate that the average Grünwald algorithm achieves an improvement of about a factor of 3 in constant phase bandwidth ($\pm 10^\circ$) in a fair comparison to the state-of-the-art algorithm for fractional differentiating or integrating in a digital circuit, the Infinite Impulse Response (IIR), with its most commonly used input signal history depth (10 steps). Allowing the number of bins to be limited by the number of time steps our simulation could handle, the average Grünwald algorithm makes a two to three decade constant phase bandwidth improvement over the IIR with only 20 binned input signal history registers.

Furthermore, during runtime, the number of flops per time step and the memory required both scale as N_b, the number of bins, which we have tested at 10 to 20. There is a one time initialization cost where the number of flops scales as N_h, the total number of historical time steps stored within the bins, which may be on the order of millions. Exponential binning strategies performed well at any number of bins.

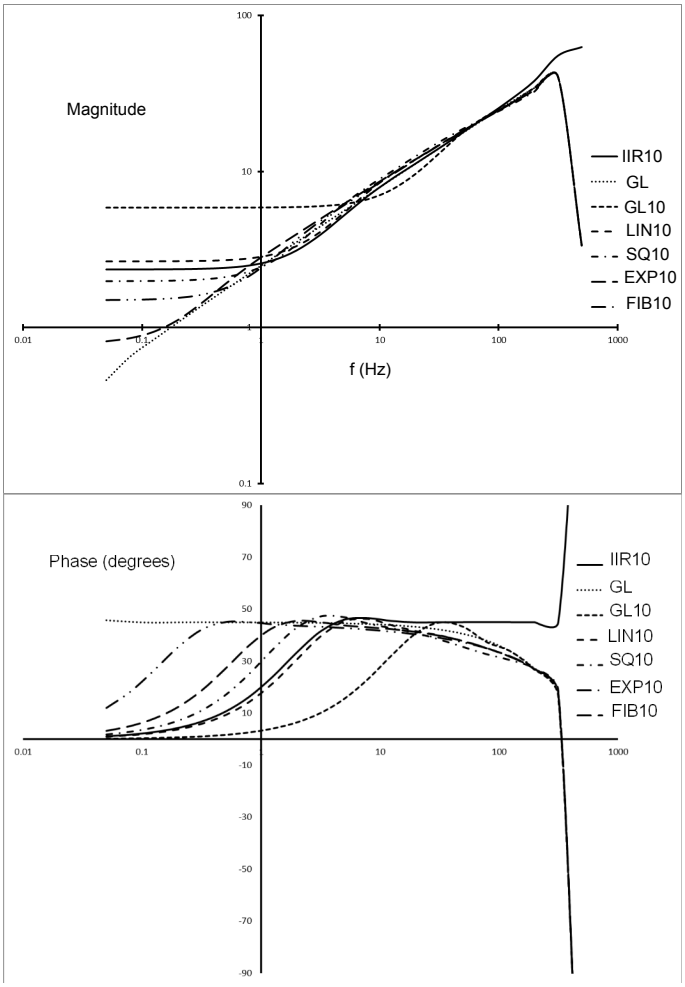


Figure 4: Exponential binning structure in the average Grúnwald algorithm (EXP10) shows a factor of three improvement in constant-phase bandwidth over the accepted infinite impulse response continued fraction expansion (IIR10) algorithm for 10 memory registers of input signal history. This figure shows magnitude and phase as a function of frequency for 10 registers of binned (SQ10, EXP10, FIB10) or unbinned (GL10, IIR10) input signal history. GL is the full Grünwald calculation, for the entire input signal history prior to that time. For this figure, the duration was 10.0 s, $N_t = 10^4$, and $\delta t = 10^{-3}$ s.

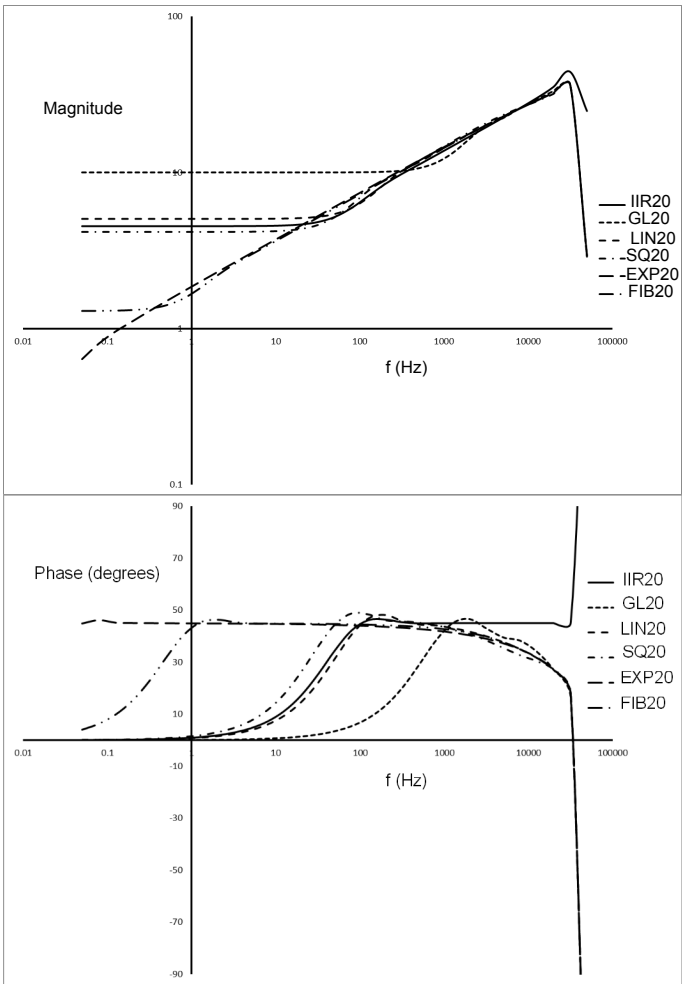


Figure 5: Exponential binning structure in the average Grünwald algorithm (EXP20) shows a two and a half decade improvement in constant phase bandwidth over the infinite impulse response continued fraction expansion algorithm (IIR20) with 20 memory registers of input signal history and $\alpha = 0.5$. The duration was 10.0 s, $N_t = 10^6$, and $\delta t = 10^{-5}$ s.

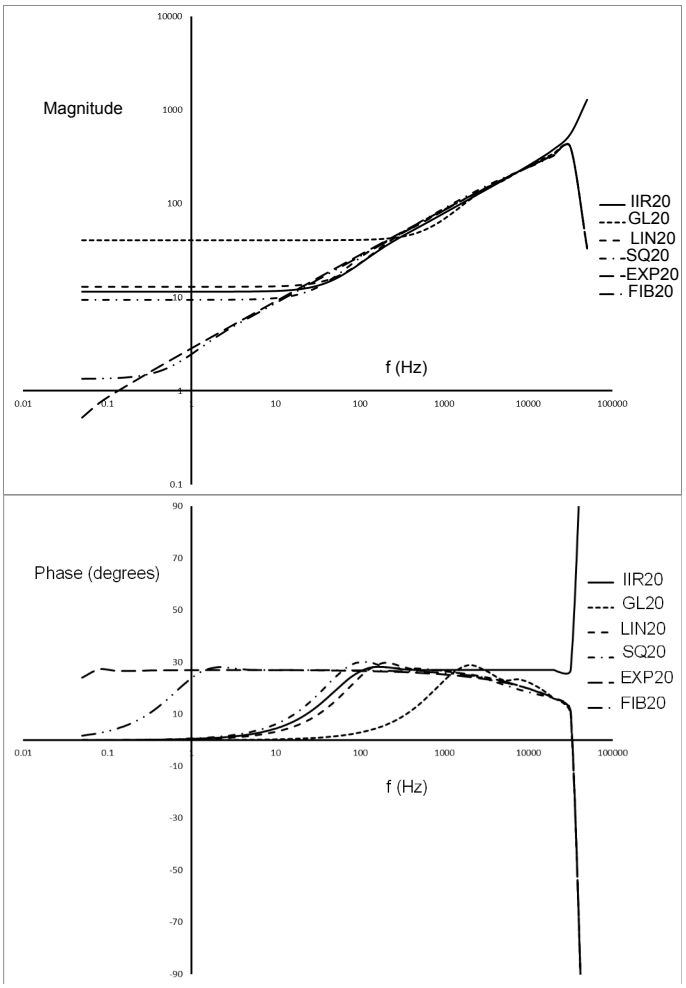


Figure 6: Exponential binning structure in the average Grünwald algorithm (EXP20) shows a three decade improvement in constant phase bandwidth over the infinite impulse response continued fraction expansion algorithm (IIR20) with 20 memory registers of input signal history and $\alpha = 0.3$. The duration was 10.0 s, $N_t = 10^6$, and $\delta t = 10^{-5}$ s.

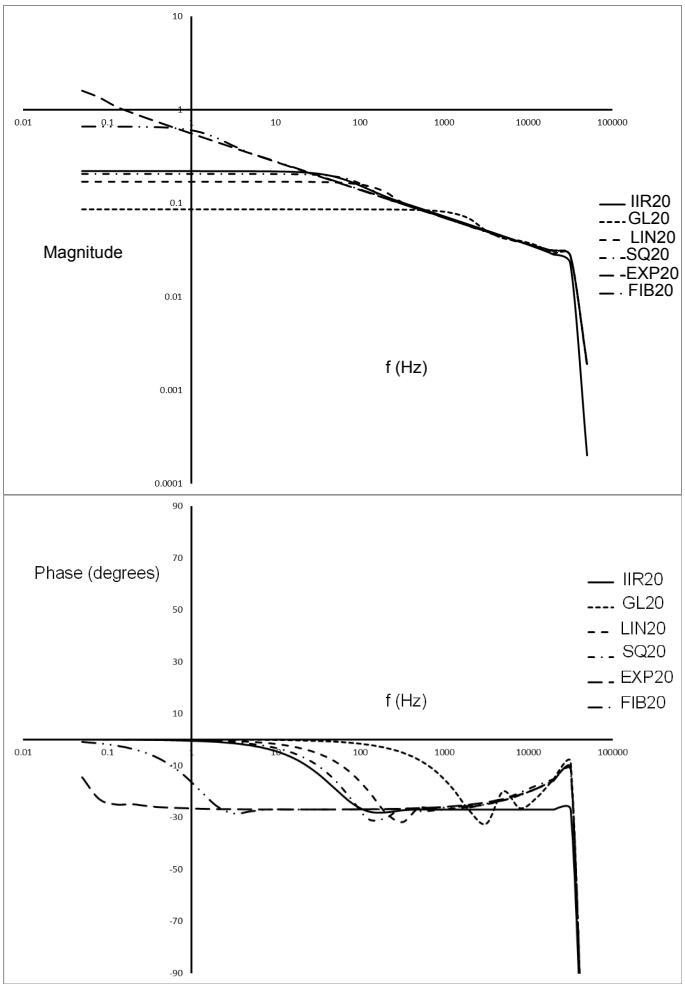


Figure 7: Exponential binning structure in the average Grünwald algorithm (EXP20) shows a two and a half decade improvement in constant phase bandwidth over the infinite impulse response continued fraction expansion algorithm (IIR20) with 20 memory registers of input signal history and $\alpha = -0.3$. The duration was 10.0 s, $N_t = 10^6$, and $\delta t = 10^{-5}$ s.

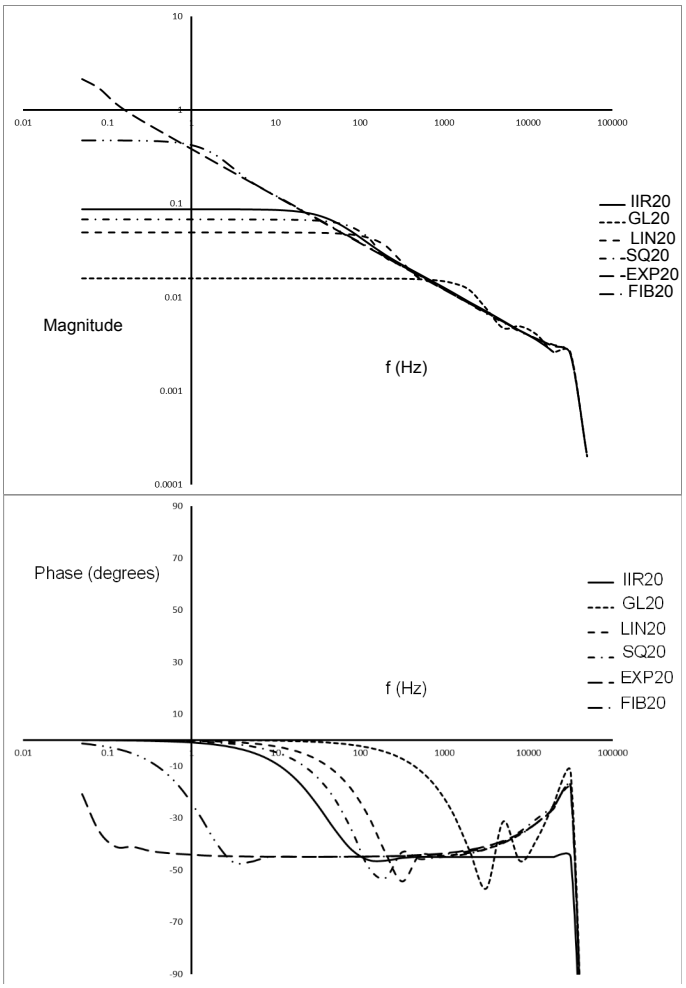


Figure 8: Exponential binning structure in the average Grünwald algorithm (EXP20) shows a two decade improvement in constant phase bandwidth over the infinite impulse response continued fraction expansion algorithm (IIR20) with 20 memory registers of input signal history and $\alpha = -0.5$. The duration was 10.0 s, $N_t = 10^6$, and $\delta t = 10^{-5}$ s.

Beyond the additional bandwidth, the average Grünwald algorithm has a second advantage over the IIR algorithm. Commonly a low-pass filter is used to prevent aliasing. The broad roll-off in phase of the average Grünwald algorithm at high frequencies would allow the Nyquist frequency to be set high when chosing an oversampling region, minimally truncating the bandwidth with the low-pass filter. Alternatively a lower order filter could be chosen, minimally disrupting the phase response. This is in contrast to the sharp plumet in phase of the IIR algorithm at high frequencies, which causes aliasing unless the highest half decade in frequency is filtered.

The ultimate goal of this work is to prepare the algorithm to be used in an mcu. Currently average Grünwald algorithm has been implemented as a numerical integration and differentiation package in C++. Our continuing goal will be to package it for an mcu and use it as a control element in a mechanical environment.

The algorithm developed in this paper is well-suited for these purposes because of its wide constant-phase bandwidth (important for non-linear control) in comparison to the IIR algorithm that is the current commonly accepted standard for use in mcu's. To obtain this extra bandwidth, it requires only a few additional registers of input signal memory. As is necessary for any real-time algorithm, the processing load per time step is constant in the average Grúnwald algorithm.

Since the time the research presented in this paper began, a couple of new numerical fractional integration and differentiation algorithms have been published. Algorithms with increasing costs per time step such as [7] are unsuitable for implementation in mcu's. On the other hand, the numerical moment method presented in [8] could be truncated at 20 registers of input signal memory, just as we have done, to achieve a constant processor load per time step. No bandwidth analysis of the moment algorithm has yet been published.

Acknowledgments

Thanks to Chad Bohannan for his contributions to the C++ code base and for his thoughts on computational efficiency. The authors also thank Saint Cloud State University for the use of computational resources.

References

[1] Luo Y, Chen YQ. Fractional Order Motion Controls. Chichester, West Sussex: John Wiley & Sons; 2013.

[2] Bohannan G. Analog fractional order controller in temperature and motor control applications. Journal of Vibration and Control 2008;14(9-10):1487–98.

[3] Monje CA, Chen Y, Vinagre BM, Xue D, Feliu V. Fractional-order Systems and Controls: Fundamentals and Applications, a monograph in the Advances in Industrial Control Series. Berlin: Springer; 2010.

[4] Chen YQ, Vinagre BM, Podlubny I. Continued fraction expansion approaches to discretizing fractional order derivatives – an expository review. Nonlinear dynamics 2004;38:155–70.

[5] Oldham K, Spanier J. The Fractional Calculus. New York: Academic Press; 1974.

[6] Press WH, Teukolsky SA, Vetterling WT, Flattery BP. Numerical Recipes in C: The art of Scientific Computing. Second ed.; Cambridge University Press; 1992.

[7] Fukunaga M, Shimizu N. A high-speed algorithm for computation of fractional differentiation and fractional integration. Phil Trans 2013;371:20120152.

[8] Pooseh S, Almeida R, Torres DFM. Numerical approximations of fractional derivatives with applications. Asian J Control 2013;15:698–712.