POST'S INVERSION FORMULA AND SEQUENCE ACCELERATION UA VIGRE PROGRAM

JONATHAN CAIN AND BENJAMIN P. BERMAN

Contents

Abstract. The Post Formula is an inherently ill–posed procedure for numerically inverting the Laplace transform. To account for this we developed arbitrarily high precision programs for Faà di Bruno's formula, which calculates the high order derivatives of composition functions arising from our application to wave propagation through disperive media. Furthermore, Post's Formula converges at a logarithmic rate, which we accomodate through the use of high precision sequence accelerators—Wynn's ρ Acceleration. Ultimately, we arrive at the conclusion that Mathematica is the most suitable programming package because of its symbolic and high–precision capabilities.

Date: July 24, 2009.

1. Project Summary

1.1. **Introduction.** The pupose of this project is to consider faster and more efficient methods for inverting the Laplace transform via Post's formula.

$$f(t) = \lim_{q\to\infty} \frac{(-1)^q}{q!} \left(\frac{q}{t}\right)^{q+1} \left(\frac{d^q}{ds^q} F(s)\right)_{s=q/t} \tag{1}$$

The crux of the problem in using this formula is quickly and efficiently computing high-order derivatives of $F(s)$. Another problem that inherently arises is the large numerical coefficients from taking large q, especially when t is small. Thus, a computer program with arbitrary precision and relatively quick computing time is desired.

1.2. **Mathematical Backround.** [1]

The problem with the numerical inversion of the Laplace transform is the amplification of error that arises from the exponental integrand.

$$\mathcal{L}(f(t)) = F(s) = \int_0^\infty e^{-st} f(t)\, dt \tag{2}$$

$$\mathcal{L}^{-}1(G(s)) = \frac{1}{2\pi i} \oint_{\text{Bromwich}} e^{st} F(s)\, ds \tag{3}$$

$$f(t) : \mathbb{R} \to \mathbb{C} \tag{4}$$

$$F(s) : \mathbb{C} \to \mathbb{C} \tag{5}$$

As an alternative to Bromwich contour integration, Emil Post's formula (Equation 1) where $f(t)$ is continuous on $[0, \infty)$ and for some $b \in \mathbb{R}$ such that,

$$\sup_{t>0} \frac{|f(t)|}{e^{bt}} < \infty \tag{6}$$

is particularly attractive due to the absence of tuning parameters (c.f., Weeks and Talbot's methods) and the fact that $s \notin \mathbb{C}$.

There are two problems with using Post's inversion formula: First, when implementing the formula on a computer, amplification of error occurs as described above, especially truncation, discretization, and roundoff errors. The second problem is the logarthimic convergence of the method and the error rate of $\frac{1}{q}$ as $q \to \infty$.

The solution to the first problem is discussed in detial in the section, Struggle for Precision. The solution to the second problem is to use

[1] This subsection and the one that follows is a summary of the research conducted by Patrick O. Kano and Moysey Brio [3]

a series acceleration method. Sequence acceleration produces a more accurate result, much faster than iterating the sequence itself. However, the rate of convergence must first be known in order to choose the appropriate accelerator. The following (Equation 7) defines the rate of convergence $a(n)$, where $\lim_{n\to\infty} S_n = S$.

$$a(n) = \frac{|S_{n+1} - S|}{|S_n - S|} \tag{7}$$

$$\lim_{n\to\infty} a(n) = c, c \in (0, 1) \tag{8}$$

$$\lim_{n\to\infty} a(n) = 1 \tag{9}$$

Equation 8 is the condition for a linear rate of convergence; e.g., $S_n = (5/7)^n$. Equation 9 is the condition for a logarithmic rate of convergence. The rate of our sequence, $\frac{1}{q}$, which arises from using Post's formula for finite q, is logarithmic. Because there is no one algorithm that accelorates all logarthimic sequences [2], it is important to choose a good accelerator by understanding the rate of convergence.

Currently, the generalized Wynn-ρ algorithm is being used. This method recurisively yeilds an approximation for the function $f(t) = \lim_{n\to\infty} \phi_n$ by ρ^0_{N-2}. The recursive rule,

$$\rho^{(n)}_{-1} = 0 \tag{10}$$

$$\rho^{(n)}_{0} = x_n \tag{11}$$

$$\rho^{(n)}_{k} = \rho^{(n+1)}_{k} + \frac{k}{\rho^{(n+1)}_{k-1} - \rho^{(n)}_{k-1}} \tag{12}$$

produces the matrix:

$$\begin{array}{cccccc} \rho^0_{-1} = 0 & \rho^0_0 = \phi_0 & \rho^0_1 & \rho^0_2 & \cdots & \rho^0_{N-2} \\ \rho^1_{-1} = 0 & \rho^1_0 = \phi_1 & \rho^1_1 & \rho^1_2 & \cdots & \rho^1_{N-2} \\ \rho^2_{-1} = 0 & \rho^2_0 = \phi_2 & \rho^2_1 & \rho^2_2 & & \\ \vdots & \vdots & \vdots & \vdots & & \\ \vdots & \vdots & \vdots & \rho^{N-3}_2 & & \\ \vdots & \vdots & \rho^{N-2}_1 & & & \\ \rho^{N-1}_{-1} = 0 & \rho^{N-1}_0 = \phi_{N-1} & & & & \end{array} \tag{13}$$

We will describe which accelerators is best for which sequences in the section, Sequence Acceleration.

We now turn to the problem of computing high-order derivatives. In our application of Post's formula, we need to analyze a composition of functions. We utilize Faà di Bruno's formula for the composition of two functions.

$$\frac{d^n}{dt^n} f(g(t)) = \sum_{k=0}^{n} f^{(k)}(g(t)) B_{n,k}(g'(t), ..., g^{(n-k+1)}) \tag{14}$$

This formula uses Bell polynomials of the second kind defined recursively as:

$$B_{p,q} = \sum_{m=1}^{q-p+1} \begin{pmatrix} q-1 \\ m-1 \end{pmatrix} \frac{d^m g}{dt^m} B_{q-m,p-1} \tag{15}$$

$$B_{0,0} = 1 \tag{16}$$

$$B_{q,0} = 0 \text{ for } 1 \geq q \tag{17}$$

$$B_{q,1} = \frac{d^q g}{dt^q} \tag{18}$$

$$B_{q,q} = (g(t))^q \tag{19}$$

1.3. **Application Backround.** The application of our project arises from the propagation of light through dielectric materials with nontrival dispersion relations. With sufficient assumptions, Maxwell's equations for relating electromagnetic fields to charge density and current density reduce to:

$$\vec{E}(\vec{k}, s) = \beta(|\vec{k}|, s)\vec{E}(\vec{k}, t = 0) + \alpha(|\vec{k}|, s)\frac{\partial E}{\partial t}(\vec{k}, t = 0) \tag{20}$$

with

$$\alpha(\vec{k}, s) = \alpha(|\vec{k}|, s)\frac{1}{s^2\epsilon_r + c^2|\vec{k}|^2} \tag{21}$$

$$\beta(\vec{k}, s) = \beta(|\vec{k}|, s) = \frac{s}{s^2\epsilon_r + c^2|\vec{k}|^2} = s\alpha(s, k) \tag{22}$$

where $\epsilon_r(s)$ is the dispersion relation for a given material, $|\vec{k}|^2 = k^2$ is one spatial dimension, c^2 is the speed of light squared, and s is the Laplacian analogue of time t. The problem we address involves the numerical inverision of the coefficients, $\alpha(\vec{k}, s)$ and $\beta(\vec{k}, s)$, from Laplacian space to time.

Thus, the necessity for applying Faà di Bruno's formula (Equation 14) for the composition of two functions is apparent for computing the

approximate inverse coefficients via Post's formula,

$$\alpha \approx \frac{(-1)^q}{q!}\left(\frac{q}{t}\right)^{q+1} D^q\alpha(k, q/t) \tag{23}$$

$$\beta \approx \frac{(-1)^q}{q!}\left(\frac{q}{t}\right)^{q+1} D^q\beta(k, q/t) \tag{24}$$

Using this so-called "Bell-Post" method, it is possible to approximte an optical wave propagation through a dieletric media given a dispersion relation $\epsilon_r(s)$ and its derivatives along the real axis for a given time and wave number.

2. Matlab Exchange File — Faà di Bruno

The first project task was to extend the Matlab [6] exchange file IncompleteBellPoly.m to include Faà di Bruno's formula. We immediately encountered the problem of how to input the functions and their derivatives. Matlab cannot compute the derivatives of functions numerically, only symbolically with the Symbolic Toolbox. However, Faà di Bruno's formula, specifically $f^{(n)}(g(t))$ requires that the function f be entered symbolically; i.e., we can enter numbers for $g(t)$ and its derivatives, but not the function f and its derivatives since they depend on the specified values of $g(t)$. So, we chose to have the user provide the functions and their derivatives in cells.

A major advantage of this input method, besides saving some money on the Symbolic Toolbox, is that one can symbolically compute the derivatives in Mathematica [5] or Maple [4], then simply copy and paste the derivatives into Matlab. Furthermore, function handles can be used as an input method, if prefered by the user.

If one has access to the Symbolic Toolbox, they can directly compute $\frac{d^n}{dt^n}f(g(t))$ since the Symbolic Toolbox can easily use the chain rule. Hence, one could avoid the need for Faà di Bruno's formula altogether. Another inherent advantage of the Symbolic Toolbox is that it computes $\frac{d^n}{dt^n}f(g(t))$ much faster that our program.

Below is a table comparing the run times of the two different progams.

TABLE 1. Comparison of run time

Input	Time

Matlab (mp)
`g = {'sin(t)','cos(t)','-sin(t)','-cos(t)','sin(t)'};`
`f = {'exp(t)','exp(t)','exp(t)','exp(t)','exp(t)'};`
`t = 1:10;`
`prec = 100;`
`tic; dbrunostringHighPrecision(t,prec,f,g); toc`
	1.39004 seconds

Matlab (syms)
`syms t;`
`f = exp(sin(t));`
`tic; h = diff(f,t,4); mp(subs(h,t,1:10),100); toc`
	0.310071 seconds

Mathematica
`f[t_] = Exp[Sin[t]];`
`t1 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};`
`prec = 100;`
`Timing[fd = D[f[t], {t, 4}]; N[fd[t1], prec];]`
	1.85615×10^{-16} seconds

The next problem dealt with precision. The first versions of this program directly accessed the IncompleteBellPoly.m file, requiring the user to have downloaded the file to an appropriate directory where both files are stored. However, this program does not run in multiprecision; therefore, we took the relevant parts of this file and encorporated them, along with the Multi Precision Toolbox, into our file. Further discussion is given in the section, Struggle for Precision.

See Appendix A for dbrunostringHighPrecision.m

3. Sequence Accleration — Wynn–ρ

As mentioned, one of the main goals of our project was to experiment with sequence acclerators. The main weakness of Post's Inversion Formula (Equation 1) is that it converges to its limit like $1/n$ converges to 0. We spent a great deal of time trying to understand the underlying mechanism of Wynn's ρ acceleration specifically Equation 12.

3.1. **Rational Interpolation.**

In our literature search, we found that in many cases the derivation of Wynn's ρ algorithm is completely ignored. In the cases where it is not ignored, it is only mentioned that it has to do with Rational Interpolation, and extrapolating the interpolant to infinity. We verified that this is in fact the case using Maple's symbolic manipulation for high degree rational interpolation.

The even columns of the Wynn–ρ matrix (Equation 13) correspond to the rational interpolation of the original sequence. For example, using the first 3 points, and a 1^{st} degree rational interpolation yields the first element of column 2. Using the first 5 points, and a 2^{nd} degree rational interpolation yields the first element of column 4. In general, the best output from Wynn-ρ (the upper–right most element) is exactly equal to using all of the points in the sequence and the highest possible degree rational interpolation.

We have found that Wynn's ρ algorithm is roughly 10 times faster than using the built in RationalInterpolation function in Mathematica, specifically for high precision acceleration of sequences with length between 10 and 1000. This makes some sense, since the acceleration is concerned only with the limit of the interpolant (the coefficients of the highest degree terms), whereas the interpolation must concerned itself with every coefficient of the rational function. We could probably have sped up the algorithm a bit further by having it only work with two columns at a time, rather than constructing the entire matrix.

Our understanding of the connection between Wynn-ρ and rational interpolation is hopefully a stepping stone toward being able to truly derive the algorithm. There is a big gap in the literature at the moment, and if someone would search a bit deeper and derive a handful of sequence accelerators, we think that it would be very worthwhile.

3.2. **The Mysterious Numerator.** Another goal of the project was to toy around with the numerator from Wynn's ρ algorithm (equation 12). This numerator is fully described in Osada's paper [7], where the term is actually related to the rate of convergence of the sequence. We verified Osada's modification for the sequences:

$$S_1(n) = \left(1 + \frac{1}{n}\right)^n \tag{25}$$

$$S_2(n) = \sum_{k=1}^{n} \left(k + e^{1/k}\right)^{\sqrt{2}} \tag{26}$$

As Osada states, S_1 should converge faster with a numerator of k ($\theta = -1$), and S_2 should converge faster with a numerator of $k-2+\sqrt{2}$. We confirmed that this is true.

Through studying Osada's numerators and the code implementation of Wynn's ρ algorithm, we noticed that the $k-2$ numerator in Kano and Brio 2009 is actually a misnomer. The problem was that $k-2$, which is perfectly accurate in the computer program due to the change

of indices, was carried over to the paper without being changed back to the traditional indices of the algorithm (Equation 12).

See Appendix B for wynnrho.m, wynnrho.nb

3.3. **Aitken's Δ^2 Acceleration.** The Aitken's Δ^2 Method is a acceleration technique for linearly converging sequences. It is defined as follows:

$$A_n = x_n - \frac{(x_{n+1} - x_n)^2}{(x_{n+2} - 2x_{n+1} + x_n)} \tag{27}$$

To better understand sequence acceleration, we compared Wynn's ρ accleration to Aitken's method. We confirmed that Aitken's method is a fantastic accelerator for linearly converging sequences, and a bad accelerator for logarithmically converging sequences by using the following test sequences:

$$S_1(n) = \left(1 + \frac{1}{n}\right)^n \tag{28}$$

$$S_2(n) = \left(\frac{5}{7}\right)^n \tag{29}$$

Furthermore, the Wynn-ρ method was highly ineffective on test sequence S_2 (Equation 29). Which just goes to show Weniger's point that it is important to test various accelerators on a given sequence.

Here, we present a comparison of the error for Aitken's Method, and Wynn's ρ algorithm with the sequence itself. This comparison is for the sequence S_1 (Equation 28):

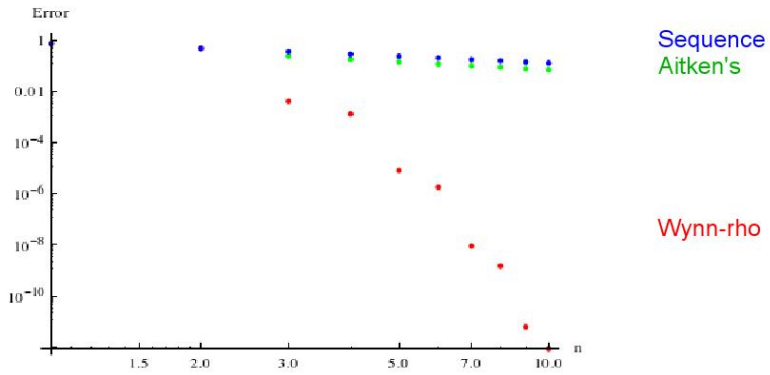

FIGURE 1. Notice that the plot is on a log-log scale. The original sequence is in blue, Aitken's accelerated sequence is in green, and Wynn-ρ is in red. For this sequence with a logarithmic rate of convergence, Wynn-ρ is by far the best, we have found that for this sequence it can be accurate up to 10^{-58} with an original sequence of only 32 values.

3.4. **Modified Aitken's Δ^2 Acceleration.** Investigation into the modified Aitken's method should be conducted. Osada [7] showed that this acceleration algorithm produces results similar to Wynn-ρ for logarithimically convergent sequences. As stated above, further work is needed into understanding how acceleration algorithms are created. In our opinion, the investigation into this acceleration method may prove the most fruitful becuase there is an interesting relationship between modified Aitken's and Wynn-ρ. From Eqaution 12 and Matrix 13, creating the ρ_1^2 term in the sequence (for any k-length: Δx) yields,

$$\rho_1^2 = \phi_2 + \frac{2\Delta x}{\frac{\Delta x}{\Delta\phi_2} - \frac{\Delta x}{\Delta\phi_1}} \tag{30}$$

$$= \phi_2 + \frac{2\Delta\phi_1\Delta\phi_2}{\Delta\phi_1 - \Delta\phi_2} \tag{31}$$

where, $\Delta\phi_1 = \phi_2 - \phi_1$ and $\Delta\phi_2 = \phi_3 - \phi_2$. Thus,

$$\rho_1^2 = \phi_2 + \frac{2(\phi_2 - \phi_1)(\phi_3 - \phi_2)}{-\phi_1 + 2\phi_2 - \phi_3} \tag{32}$$

(33)

Notice the correspondence to the latter term in the formula for the modified Aitken Δ^2 method:

$$s_n^{k+1} = s_n^k - \frac{2k+1-\theta}{2k-\theta}\frac{(s_{n+1}^k - s_n^k)(s_n^k - s_{n-1}^k)}{s_{n+1}^k - 2s_n^k + s_{n-1}^k} \tag{34}$$

Besides giving insight into the derivation of acceleration algorithms, perhaps the modified Aitken's method will porduce faster, more efficient results than Wynn-ρ. However, our computer implementation of this algorithm produced inconclusive results.

4. Struggle for Precision

It is very important for each phase of Post's Inverse Formula—the Bell/Faà di Bruno formula and the sequence accelerator—to be highly accurate. One of our goals was to implement a high precision routine in Matlab. However it seems that this is something that Matlab is simply not built for.

Our first attempt was to use the Matlab–official Fixed Point Toolbox. The struggle here is that this Toolbox is meant to be used to go from Matlab's standard precision down to a fewer digits. Paul Dostert was able to work around this and produce a *tricky* code that allowed us to work in Matlab with high precision using solely the Fixed Point Toolbox. See Appendix D.

Then, we discovered a user submitted toolbox called the Multi-Precision Toolbox. Using this was much more straightforward, and we achieved good results. However, regardless of using either toolbox in Matlab, the run time was much much longer than using Mathematica to do the same problems in the same precision. For this reason we recommend sticking to Mathematica when it comes to high precision calculations; plus, it has the added benefit of being able to compute symbolic derivatives, which may replace the need for a Faà di Bruno and Bell algorithm. Nonetheless, if you can only work with Matlab, our high precision codes combined with the Multi-Precision Toolbox, should get you pretty far.

5. Discussion and Conclusion

Over the course of the last month, we have created a Matlab function for Faà di Bruno's Formula (potentially for the Matlab File Exchange); experimented with various accelerators, ultimately confirming that Wynn-ρ is most suitable; and developed high precision methods to aid with Post's inversion in Matlab, though Mathematica or Maple

may still be more effective. Our overall focus was less on Post's inversion itself, but more on the smaller pieces that may help make it a more reasonable method for numerically inverting the Laplace transform.

The problem of Post's Inversion is that in some ways it lies in the gray area between numerical and symbolic computation. It would be great to have the speed of C++ for the loops of Faà di Bruno's Formula and Wynn-ρ, but unfortunately both also require very high precision and perhaps the symbolic manipulation of Mathematica or Maple. If we were more skilled in C++ we may have been able to investigate this further with the use of the ARPREC package. In the future, investigation into using ARPREC and C++ to perform the computations should be conducted with the purpose of possibly computing faster, more accurate results. However, based on our results (specifally the run times in Matlab and Mathematica), we can conclude that the use of Mathematica or the Symbolic toolbox in finding the derivative of the composition of two functions, is a faster program. While our high precision Faà di Bruno script has its utilities, when it is directly applied to Post's inverse formula in it's current state, it runs into a problem with the symbolic substitution of $s = q/t$ into the derivative (see Equation 1). Patrick Kano's Mathematica script [3] is more suitable due to its ability to quickly compute in a high precision symbolic environment.

In the future, it is important that we gain the ability to derive these sequence accelerators step by step. The key to this may lie in the papers of Wynn [9] and Bjorstad [1]. In general, it is our opinion that the literature on sequence accelerators has neglected the means of creating the algorithms, only to focus on the end results and tests on various sequences. We have made some strides toward this derivation by explicity making a connection between components of Wynn-ρ acceleration and rational interpolation.

This summer we focused on exploring and improving two components of the Post Inversion Formula (Equation 1): numerically with Faà di Bruno's Formula, and analytically with Wynn-ρ. We feel that we have made a significant contribution, even if we simply confirmed that Patrick Kano's previous work in Mathematica is the fastest and most efficient. After experimenting with with various sequence accelerators, we find the Wynn-ρ is still most suitable, but further tests using the Modified Aitkin's method is necessary. We are also much closer to a fundamental understanding of sequence accelerators and how they are derived.

6. Appendix A: Faà di Bruno's Formula

Our matlab exchange file for Faà di Bruno's Formula:
NOTE: The symbol "(*)" signals a LaTex fomatting line break

```
function fout = dbrunostringHighPrecision(domin,prec,fins,gins)
%DBRUNOSTRINGHIGHPRECISION  Calculates the n-th derivative of the composition of two functions
(*)using Faa' di Bruno's formula.
%   [fout] = DBRUNOSTRINGHIGHPRECISION(domin,prec,fins,gins) returns a cell array giving the n-th
(*)derivative of [f(g(t))] at each value in the domain.
%
%   INPUTS: domin          -   the domain over which at each point the functions and their n
(*)derivatives will be computed.
%           prec           -   the bit precision for all computations.
%           fins           -   the function f and n derivatives as strings in a cell array.
%           gins           -   the function g and n derivatives as strings in a cell array.
%
%   EXAMPLE 1:
%      t = 1:.01:2*pi;                                          % domain
%      prec = 128;                                              % 128 bit precision (double-
(*)double precision)
%      f = {'exp(t)','exp(t)','exp(t)','exp(t)','exp(t)'};   % f=exp(t) and four derivatives
%      g = {'sin(t)','cos(t)','-sin(t)','-cos(t)','sin(t)'}; % g=sin(t) and four derivatives
%      fout = dbrunostringHighPrecision(t,prec,f,g)          % runs the program
%
%   [fout] = DBRUNOSTRINGHIGHPRECISION(domin,prec,fins) returns a cell array giving the n-th
(*)derivative of [f(g(t))] at each value in the domain.
%
%   INPUTS: domin          -   the domain over which at each point the functions and their n
(*)derivatives will be computed.
%           prec           -   the bit precision for all computations.
%           fins           -   the handle to a function that yields the derivatives of
(*)f and g as strings in a cell array
%
%   EXAMPLE 2:
%      t = 1:.01:2*pi;                                 % domain
%      prec = 128;                                     % 128 bit precision (double-double precision)
%      fg = @derivativefinder();                       % fg is a handle to a function that will
(*)find the derivatives of f and g and format appropriately
%      fout = dbrunostringHighPrecision(t,prec,fg)   % runs the program
%
%   NOTE 1: This program requires the Multi Precision Toolbox (mp.m) by Ben Barrowes
(*)found at www.mathworks.com
%
%   NOTE 2: The incomplete Bell polynomials section is inspired by the file
(*)exchange IncompleteBellPoly.m by Moysey Brio and Patrick O. Kano found at
%           www.mathworks.com
%
%   Authors:
%   Benjamin Berman and Jonathan Cain
%   University of Arizona
%
%   Email:
%   benpb@umich.edu
%   cain1@email.arizona.edu
%
%   Latest Modification Date:
%   July 24, 2009
```

```
% Process inputs and do error-checking
if nargin == 3
    %function is called
    [fin, gin] = fins();
    domin = gins;
elseif nargin == 4
    fin = fins;
    gin = gins;
else
   error('Either 3 inputs (function handle, vector, precision), or 4 inputs (cell vector, cell
*vector, vector, precision) are needed.');
end

if length(fin) ~= length(gin)
    error('Need same number of derivatives for f and g.');
end

% Initailize input matrices in prec -bit precision
f = mp(zeros(length(fin),length(domin)),prec);
g = mp(zeros(length(gin),length(domin)),prec);

% Evaluate the input at the appropriate points
for k = 1:length(fin)
    % Convert cell vector to characters
    gt1 = char(gin(k));
    % Creat inline function and vectorize operations
    gtemp = vectorize(inline(gt1));
    % Fill in initialized matrix
    g(k,:) = gtemp(domin);
    ft1 = char(fin(k));
    ftemp = vectorize(inline(ft1));
    % f evaluated at g
    f(k,:) =ftemp(g(1,:));
end

% Initialize computation matrices in prec -bit precision
B = mp(zeros(length(domin),length(gin),length(gin)),prec);
fseq = mp(zeros(1,length(fin)),prec);
fout = mp(zeros(1,length(domin)),prec);

Nin = length(gin)-1;
Kin = length(gin)-1;


for t = 1:length(domin)

    % Compute high precision Bell Polynomials
    DataList = g(2:end,t);
    if(Nin==0 && Kin==0)
        OutMatrix = 1;
    elseif(Nin>0 && Kin==0)
        OutMatrix = mp(zeros(Nin+1,1),prec);
        OutMatrix(1,1) = 1;
    else
        Bm = mp(zeros(Nin,Kin),prec);
        Bm(1:Nin,1) = mp(DataList(1:Nin),prec);
        for nidx=2:Nin
```

```
            for kidx=2:Kin
                for midx=1:nidx-kidx+1
                    Bm(nidx,kidx) = Bm(nidx,kidx) + nchoosek(nidx-1,midx-1)*DataList(midx)
(*)*Bm(nidx-midx,kidx-1);
                end
            end
        end
        OutMatrix = mp(eye(Nin+1,Kin+1),prec);
        OutMatrix(2:Nin+1,2:Kin+1) = Bm;
    end

    B(t,:,:) = OutMatrix;

    % Use di Bruno's formula
    for k = 1:length(fin)
        fseq(k) = f(k,t)*B(t,length(gin),k);
    end
    fout(t) = sum(fseq);
end
end %function definition
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

7. Appendix B: Wynn's ρ Acceleration

Our matlab file for Wynn-ρ acceleration:

```
function [wynnrho2 wrbest] = wynnrhoHighPrecision(sequence,prec)

ls = length(sequence);
wynnrho = mp(zeros(ls,ls+1),prec);
wynnrho(:,2) = sequence;
numiter = 0;

for k = 3:(ls+1)
    for n = 1:(ls+2-k)

        wynnrho(n,k) = (k-2)/(wynnrho(n+1,k-1)-wynnrho(n,k-1))+wynnrho(n+1,k-2);

    end
end

wynnrho2 = wynnrho(:,2:2:end);
wrbest = max(wynnrho2(:,end));

end
```

Our Mathematica file for Wynn-ρ:

```
g[n_] = (1 + 1/n)^n;
prec = 40;
nlength = 7;
c = N[Table[g[i], {i, 1, nlength}], prec];
El = ConstantArray[0, {nlength, nlength + 1}]; El[[All, 2]] = c;
Table[ El[[m, k]] =
   El[[m + 1,
     k - 2]] + (k - 2)/(El[[m + 1, k - 1]] - El[[m, k - 1]]), {k, 3,
   nlength + 1}, {m, 1, nlength + 2 - k}];
g[n_] = n; j = Table[g[k], {k, 2, nlength + 1, 2}];
wynnrhobest = Max[El[[All, Max[j]]]];
```

```
wynnrho1 = N[MatrixForm[El[[All, All]]], prec];
wynnrho2 = N[MatrixForm[El[[All, j]]], prec];
```

8. Appendix C: Aitken's Δ^2 Acceleration

Our matlab file for Aitken's method:

```
function aout = aitkens(seq)
% input a sequence, output aitken's accelerated sequence
ls = length(seq);
aout = zeros(1,ls);

for i = 1:(ls-2)
    aout(i) = seq(i) - (seq(i+1) - seq(i))^2/(seq(i+2)-2*seq(i+1)+seq(i));
end

end
```

9. Appendix D: P. Dostert's High Precision in Matlab

This code requires only the fixed point toolbox and achieves arbitrarily high precision.

```
function [wholeS,decS]=showNumberFull(fiIn);


binIn = fiIn.bin;
WordLength = fiIn.WordLength;
FractionLength = fiIn.FractionLength;

% Do the whole number part
for i=1:WordLength-FractionLength
    BinWhole(i) = str2num(binIn(i));
end
WholeFac = 2.^(length(BinWhole)-1:-1:0);

% Do in chuncks of 48, for now...
MaxChunkSize = 48;

% Now we want to create a group of doubles, each using 48 bits. So we
% create a double from the 1st 48, a double from 49 - 96, etc...
i=0;
for j=1:floor(length(BinWhole)/MaxChunkSize)
    i=j;
    index = (i-1)*MaxChunkSize+1:i*MaxChunkSize;
    wholeS(i) = dot(BinWhole(index),WholeFac(index)) ;
end

% Do the rest separately (the "end" part of the bit string)

index = i*MaxChunkSize+1:length(BinWhole);
if(length(index)>1)
    wholeS(i+1) = dot(BinWhole(index),WholeFac(index));
end



% Do the decimal part
```

```
for i=1:FractionLength
    BinDec(i) = str2num(binIn(length(binIn)-FractionLength+i));
end
DecFac = 2.^(-(1:length(BinDec)));

% Do in chuncks of 48, for now...
MaxChunkSize = 48;

% Now we want to create a group of doubles, each using 48 bits. So we
% create a double from the 1st 48, a double from 49 - 96, etc...
i=0;
for j=1:floor(length(BinDec)/MaxChunkSize)
    i=j;
    index = (i-1)*MaxChunkSize+1:i*MaxChunkSize;
    decS(i) = dot(BinDec(index),DecFac(index)) ;
end

% Do the rest separately (the "end" part of the bit string)
index = i*MaxChunkSize+1:length(BinDec);
if(length(index)>1)
    decS(i+1) = dot(BinDec(index),DecFac(index));
end

% -- DISPLAY THE NUMBER --
% Note: =0 is fine here... since each component is below double precision
counter = 0;
for i=1:length(wholeS)
    if(wholeS(i) ~= 0)
        if(counter == 0)
            fprintf('%g',wholeS(i));
        else
            if(wholeS(i)>0)
                fprintf('+%g',wholeS(i))
            else
                fprintf('%g',wholeS(i))
            end
        end
        counter = counter+1;
    end
end

for i=1:length(decS)
    if(decS(i) ~= 0)
        if(counter == 0)
            fprintf('%g',decS(i));
        else
            if(decS(i)>0)
                fprintf('+%g',decS(i))
            else
                fprintf('%g',decS(i))
            end
        end
        counter = counter+1;
    end
end
fprintf('\n');
```

10. Acknowledgements

We express our deepest graditude to Professor Moysey Brio, Dr. Paul Dostert, and Dr. Patrick O. Kano for their continued support throughout this project. Their work leading up to this project and their help this summer were an integral part of our research experience.

Thank you to the University of Arizona, Program in Applied Mathematics and the VIGRE Program for funding our research.

This project was funded by the University of Arizona's NSF Grant: EMSW21–VIGRE Award 0602173.

References

[1] Bjorstad, P., Dahlquist, G. and Grosse E. *Extrapolation of asymptotic expansions by a modified Aitken Δ^2-formula*. BIT Numerical Mathematics, vol. 21, no. 1, pp. 56-65. 1981.

[2] Delahaye, J.P. and Germain-Bonne, B. *The set of logarithmically convergent sequences cannot be accelerated*. SIAM Journal on Numerical Analysis, vol. 19, no. 4, pp. 840-844. 1982.

[3] Kano, P. and Brio, M., *Application of Post's Formula to Optical Pulse Propagation in Dispersive Media*. Paper has been submitted and is under review. 2009.

[4] Maple 12. `http://www.maplesoft.com`

[5] Mathematica 7.0. `http://www.wolfram.com`

[6] Matlab R2008B. `http://www.mathworks.com`

[7] Osada, N. *A Convergence Acceleration Method for Some Logarithmically Convergent Sequences* SIAM Journal on Numerical Analysis, vol. 27, no. 1, pp. 178-189. 1990.

[8] Weniger, E. *Letter to Saul Teukolsky*. `http://www.nr.com/webnotes/nr3webR1.pdf`, 30 October. 2007.

[9] Wynn, P. *On a procrustean technique for the numerical transformation of slowly convergent sequences*. Proceedings of the Cambridge Philosophical Society, vol. 52, no. 4, pp. 663-671. 1956.

J. Cain, Program in Applied Math, University of Arizona, Tucson, AZ

E-mail address: `cain1@email.arizona.edu`

B. P. Berman, Program in Applied Math, University of Arizona, Tucson, AZ

E-mail address: `bpberman@email.arizona.edu`