A Mathematical Tour of Babbage Difference Engine No. 2

Charles L. Rino
crino@comcast.net
http://chuckrino.com/wordpress/

Abstract

The Babbage Difference Engine No. 2 replica that currently resides at the Sunnyvale Computer History Museum evaluates a 7th-order polynomial. Each cycle of the engine generates a new value. With the luxury of modern computational tools one can generate an approximating polynomial that will demonstrate the intended purpose of the engine, namely to compute and print error-free mathematical tables.

This paper reviews the properties of finite differences that underlie the design of Difference Engine No. 2. An algorithm that converts finite differences to polynomial coefficients is included. A direct emulator of the 31 digit difference engine is described and demonstrated. The paper concludes with the demonstration of initiation processes than might have been used to initiate a difference engine computation.

1 Introduction

A reconstruction of Charles Babbage's Difference Engine No. 2 (DE2) is currently on loan to the Computer History Museum (CHM) through the generosity of Nathan Myhrvold. The Charles Babbage story and how DE2 came to be constructed more than 150 years after it was designed can be found in Doron Swade's book, *The Cogwheel Brain, [1]* and his paper, *The Construction of Charles Babbage's Difference Engine No. 2, [2].* Briefly, evolving industrial-age commerce created a demand for accurate tables of mathematical functions. Mathematicians knew how to perform the computations, [3] but manual execution was time consuming and translating the results to print was error prone. In a popular anecdote, Charles Babbage was comparing tables of functions with his astronomer colleague John Hershel. They found numerous discrepancies, which caused Babbage to exclaim, "I wish to God that these calculations had been executed by steam."

In 1823 Babbage proposed the construction of a calculating engine that would compute and print error-free tables of mathematical functions. He petitioned the British government and received funding to build Difference Engine No. 1. A working fragment was constructed by his machinist Joseph Clement

and demonstrated, but the development came to an abrupt end after a dispute with Clement. Under British law Clement was entitled to keep the tools he constructed. Eleven years into the effort with only the *beautiful fragment* in hand, Babbage turned his attention to the design of an analytic engine capable of general purpose computation under punched card control. He is most famous for his analytic engine design, which foreshadowed modern digital computing. He created the design for DE2 using refinements developed for the analytic engine. He offered DE2 to the British government in 1848 as a fulfillment of his earlier commitment, but because of cost and non delivery, the offer was declined. Charles Babbage died in 1871 bitterly disappointed that his designs were never realized. He is quoted as having said "another age must be the judge."

The London Museum of Science inherited Babbage's designs and papers, which had been collected by his son. The plans resided in the London Museum until a new curator, Doron Swade, following a proposal by Allan Bromley in 1985, started a project to build DE2 from Babbage's plans. The computation section of DE2 was completed and demonstrated in 1991 to celebrate the bicentennial of Babbage's birth. The unfinished DE2 came to the attention of Microsoft's former Chief Technical Officer Nathan Myhrvold. He agreed to fund the completion of the first DE2 replica provided that the London Science Museum would build a second complete replica for his personal use. Upon its completion in 2008, Myhrvold lent his DE2 replica to CHM where it has been regularly demonstrated for public appreciation.

The web site `http://ed-thelen.org` contains tutorial material, references, photographs, computer emulations of DE2's operation, and procedures for initiating a DE2 computation. Polynomial approximation is the underlying principle of DE2's operation, which is designed to evaluate 7th-order polynomials. As we shall see, the method of finite differences makes it possible to reduce sequential evaluation of the approximating polynomial to recursive additions. Students of the DE2 often try to reproduce the addition operations on personal computers, but neither standard spread sheets nor most computation-oriented programming languages will reproduce the full 31-digit precision of DE2. With the luxury of modern symbolic computation languages, we can perform such computations with relative ease. In Babbages's time tedious manual computations were used, but specific details have been "left as exercises for the reader," as Babbage himself was fond of doing. This paper presents a mathematical tour of the design and operation of DE2 with some thoughts on how its intended purpose might have been realized in the 19th century.

To introduce some notation, let $f(x)$ represent a continuous well behaved function, which is to be approximated over a prescribed range of the independent variable x by a polynomial. Let $P(x)$ represent an approximating polynomial of order p. An order p polynomial is defined by $p+1$ coefficients, which are denoted $\widetilde{a}_k$ with $k = 0, 1, \cdots, p$. The approximating polynomial would normally be written as

$$\widetilde{P}(x) = \sum_{k=0}^{p} \widetilde{a}_k x^k. \tag{1}$$

Although $f(x)$ is a continuous function, the independent variable is evaluated only at uniformly spaced values $x_n = n\Delta x + x_0$. Thus, the function values to be used in the approximation are

$$f_n = f(n\Delta x + x_0), \text{ for } n = 0, \cdots, N-1. \tag{2}$$

Note that $\widetilde{P}(0) = \widetilde{a}_0$. Thus, we are effectively approximating a function f_n, with a polynomial

$$\widetilde{P}(n\Delta x) = \sum_{k=0}^{p} \widetilde{a}_k \left(n\Delta x\right)^k, \tag{3}$$

initially starting at $n = 0$. One could offset the independent variable in the polynomial, but it is readily shown that there is an equivalent polynomial of the same order that translates the offset to the first coefficient. For mathematical convenience, the following equivalent redefinition will be used:

$$P_n = \sum_{k=0}^{p} a_k n^k, \text{where } a_k = \widetilde{a}_k \left(\Delta x\right)^k. \tag{4}$$

The dependence on Δx has been absorbed in the polynomial coefficients, but the final form of the approximating polynomial is completely general.

With (4) as a starting point, Section 2 introduces the finite-difference method and demonstrates the properties that support the DE2 operation. Finite differences are differences between two consecutive values of a function or differences of previously computed finite differences. The eight DE2 columns (See Figure 2.3) are numbered 0 to 7. Each column has 31 number wheels that define 31-digit integers. Column 0 on the left contains f_n. Column 7 on the right contains the 7th-order difference. The intermediate columns contain modified differences 1 through 6 that will be described later. However, once initialized by entering 31-digit numbers in each column, the operation of the DE2 proceeds as follows: During the first DE2 half cycle, the odd columns are *simultaneously* added to even columns. With the symbol $\rightarrow$ indicating replacement, the simultaneous additions with replacement are $7+6 \rightarrow 6$, $5+4 \rightarrow 4$, $3+2 \rightarrow 2$, and $1+0 \rightarrow 0$. The new answer, overwritten in column 0, is available at the end of the first half cycle. During the second DE2 half cycle, the modified even columns are simultaneously added to the odd columns and f_n is transferred to the print section. Notationally, $6+5 \rightarrow 5$, $4+3 \rightarrow 3$, $2+1 \rightarrow 1$, and $0 \rightarrow$ printer.

The addition operations themselves are performed in two parts. First digit-by-digit additions are performed with mechanical carry warnings set when single-digit additions exceed 9. After the column wise addition operations are performed, the carry warnings are serviced mechanically by adding one to the next wheel in the column when a carry warning is encountered. This must be a serial operation because the carry operation itself can initiate a carry.

Section 3 describes and demonstrates a direct numerical emulation of the DE2 addition and carry operations. Direct emulation means wheel-by-wheel

additions with carry flags first set and then serviced. Direct emulation can be implemented without special software; moreover, direct emulaton makes it possible to set up the engine with carry levers disengaged. In normal DE2 operation the first four number wheels in each column are set to evaluate the trivial polynomial $P_n = n$. As long as n does not exceed 9999, there is no carry an the counting action does not affect wheels 5 through 31. Disengaging carry levers at any level isolates the lower and upper computation wheels, which can then be initiated to evaluate two polynomials simultaneously.

Attempts have been made to demonstrate a *real* DE2 computation as opposed to an evaluation of a demonstration polynomial. The least-squares method, which was developed by Carl Fedrick Gauss at the end of the 18th century, can be used to determine 8 polynomial coefficients that approximate a function with high precision. With the approximating polynomial as a surrogate for the function, 8 consecutive values are generated, a finite difference table is constructed, and the aforementioned precomputations are performed. This defines 8 starting values to be transferred manually to the number wheels.

Form the work of Brook Taylor in 1715, the intimate relation between polynomial approximation and Taylor series would also have been well known in Babbage's time. Finite-difference tables were commonly used to explore the complexity of a function presented in tabular form. Once higher-order differences cease to be significant, the function segment being investigated can be approximated by a polynomial. As one might expect, the polynomial coefficients that correspond to a finite difference table derived from the function itself are very close to the Taylor series coefficients. This suggests that 8 consecutive values of the function to be approximated could be used as an initiation sequence.

In Section 5 examples of least-squares and direct initiation of a DE2 application to base-ten logarithm tabulation are presented. The results illustrate initiation procedures that could have been used in Babbage's time. However, a mathematician of Babbage's caliber might have devised an initiation procedure uniquely suited to the DE2 operation.

2 Finite Difference Methods

Differential calculus provides a mathematical framework for analyzing functional change. Derivatives, the formal measure of functional change, are constructed from limits of finite differences normalized to the sampling interval as the sampling interval becomes infinitesimally small. The calculus of finite differences is appealing because it has its own set of constructs and theorems that do not involve infinitesimals.

2.1 Finite Differences

Once a function has been computed at uniformly spaced values of the independent variable, a finite difference table can be constructed. Forward differences

are defined as follows:

$$d_n^0 = f_n \tag{5}$$

$$d_n^m = d_{n+1}^{m-1} - d_n^{m-1} \tag{6}$$

The m index represents the order of the finite difference. The n index identifies the first contributing sample. A finite difference table to order 7 is constructed as follows:

f_n	d_n^1	d_n^2	d_n^3	d_n^4	d_n^5	d_n^6	d_n^7
f_{n+1}	d_{n+1}^1	d_{n+1}^2	d_{n+1}^3	d_{n+1}^4	d_{n+1}^5	d_{n+1}^6	
f_{n+2}	d_{n+2}^1	d_{n+2}^2	d_{n+2}^3	d_{n+2}^4	d_{n+2}^5		
f_{n+3}	d_{n+3}^1	d_{n+3}^2	d_{n+3}^3	d_{n+3}^4			
f_{n+4}	d_{n+4}^1	d_{n+4}^2	d_{n+4}^3				
f_{n+5}	d_{n+5}^1	d_{n+5}^2					
f_{n+6}	d_{n+6}^1						
f_{n+7}							

. (7)

The first column could be replaced by d_n^0, but it is more transparent to let the value of the function appear explicitly.

If d_n^p is constant at some order p, all higher order differences are zero. This occurs if and only if f_n is a polynomial of order p. Thus, a finite difference table provides a measure of how closely a polynomial of order p approximates the function. If a difference table has precisely constant differences at order p, the function is a polynomial of order p. The proposition is proved by demonstrating that unique polynomial coefficients can be recovered from the finite differences. An algorithm that will recover the polynomial coefficients from the f_n row of the difference table follows. Coefficient recovery is not need for normal DE2 operations, but it will be used in Section 4 for evaluation purposes.

2.2 Polynomial coefficient recovery

Finite differences of order m can be written in terms of $m+1$ values of f_n explicitly:[1]

$$d_n^m = \sum_{k=0}^{m} (-1)^k \binom{m}{k} f_{n+m-k}. \tag{8}$$

where

$$\binom{m}{k} = \frac{m!}{(m-k)!k!}. \tag{9}$$

If f_n is a polynomial, f_{n+m-k} can be written as

$$f_{n+m-k} = \sum_{l=0}^{p} a_l (n+m-k)^l. \tag{10}$$

[1] Abrmowitz and Stegun 25.1[4]

To determine the defining polynomial coefficients, consider the set of finite differences at $n = 0$:

$$f_{m-k} = \sum_{l=0}^{p} a_l (m-k)^l \tag{11}$$

$$= \sum_{l=0}^{p} a_l \sum_{l'=0}^{l} (-1)^{l'} \binom{l}{l'} k^{l'} m^{l-l'} \tag{12}$$

Substituting f_{m-k} into the expression for d_n^m it follows that

$$d_0^{p-j} = \sum_{k=0}^{p-j} (-1)^k \binom{p-j}{k} f_{p-j-k} \tag{13}$$

$$= \sum_{k=0}^{p-j} (-1)^k \binom{p-j}{k} \sum_{l=0}^{p} a_l \sum_{l'=0}^{l} (-1)^{l'} \binom{l}{l'} (p-j)^{l-l'} k^{l'} \tag{14}$$

$$= \sum_{l=0}^{p} a_l \sum_{l'=0}^{l} (-1)^{l'} \binom{l}{l'} (p-j)^{l-l'} S_{l'}^{p-j} \tag{15}$$

where

$$S_{l'}^{m} = \sum_{k=0}^{m} (-1)^k \binom{m}{k} k^{l'}. \tag{16}$$

One can verify that $S_{l'}^{m} = 0$ for $l' < m$. It follows that

$$d_0^{p-j} = \sum_{l=p-j}^{p} a_l \sum_{l'=p-j}^{l} (-1)^{l'} \binom{l}{l'} (p-j)^{l-l'} S_{l'}^{p-j} \tag{17}$$

$$= a_{p-j} (-1)^{p-j} S_{p-j}^{p-j} \tag{18}$$

$$+ \sum_{l=p-j+1}^{p} a_l \sum_{l'=p-j}^{l} (-1)^{l'} \binom{l}{l'} (p-j)^{l-l'} S_{l'}^{p-j} \tag{19}$$

The contributions to d_0^{p-j} are determined by the coefficients from $p-j$ to p. This leads to the following recursion:

$$a_p = \frac{d_0^p}{(-1)^p S_p^p} \tag{20}$$

$$a_{p-j} = \begin{cases} \dfrac{d_0^{p-j} - \sum_{l=p-j+1}^{p} \widetilde{a}_l \sum_{l'=p-j}^{l} (-1)^{l'} \binom{l}{l'} (p-j)^{l-l'} S_{l'}^{p-j}}{(-1)^{p-j} S_{p-j}^{p-j}} & \text{for } j = 1, \cdots, p \end{cases} \tag{21}$$

One could also set up a system of $p+1$ linearly equations whose solution defines the coefficients, but the recursive solution is more in keeping with the properties of finite difference tables.

2.3 Forward DE2 Recursion

The critical finite difference property for DE2 operation is that $p+1$ consecutive values of the approximating polynomial define all subsequent values with only recursive addition operations. From (6) it follows that $d_{n+1}^{m-1} = d_n^{m-1} + d_n^m$. Since d_n^p is constant for all n, the following recursion extends (7) indefinitely:

$$d_{n+(l+1)}^{p-(l+1)} = d_{n+l}^{p-l} + d_{n+l}^{p-(l+1)} \text{ for } l = 0, \cdots, p-1. \tag{22}$$

In (22) d_{n+l}^{p-l} defines an antidiagonal in the difference table that terminates with the value of the function in the first column. Recall that the symbol $\leftarrow$ denotes replacement. Although extension of the difference table illustrates the recursive property of finite differences, for evaluation purposes one would start with the first complete row in (7). Recursive evaluation of f_n, with repetition of 7 values already computed would proceed as follows:

$$d_{n+1}^p = d_n^p \tag{23}$$

$$d_{n+1}^{p-(l+1)} = d_n^{p-l} + d_n^{p-(l+1)} \text{ for } l = 0, \cdots, p-1. \tag{24}$$

A device with $p+1$ storage registers and a single addition unit could execute the recursion serially. With the shorthand notation d_n with $n = 7, 6, \cdots, 0$ used to designate the elements in the nth row of (7), the finite-difference algorithm is defined as follows:

$$\begin{aligned}
d'_7 &\leftarrow d_7 \\
d'_6 &\leftarrow d_7 + d_6 \\
d'_5 &\leftarrow d'_6 + d_5 \\
d'_4 &\leftarrow d'_5 + d_4 \\
d'_3 &\leftarrow d'_4 + d_3 \\
d'_2 &\leftarrow d'_3 + d_2 \\
d'_1 &\leftarrow d'_2 + d_1 \\
d'_0 &\leftarrow d'_1 + d_0
\end{aligned} \tag{25}$$

The primes are introduced to separate the values before (unprimed) and after the replacement operations (primed). To execute the difference operations mechanically, each column would have to be added to the adjacent column.

It is noteworthy that Babbage never used this scheme because of its inefficiency. In a mechanical device the provision for addition must be replicated for each column. However, the serial addition only executes one addition at a time. Six of the sever addition units are idle during each sequential addition. There is a way to do half of the operations in parallel, but the scheme is not at all obvious. To introduce the scheme, it is helpful to start with a construction that might have suggested the parallel algorithm.

In the course of trying to understand what patterns of operations lead to recurrent jams, those involved with keeping the DE2 running have used a reconstruction of the differences that preceded the jam. Starting with the last complete cycle, n, consider reconstructing the differences at preceding values, $n-1$, $n-2$, and so on. This is illustrated in the following table:

f_{n-4}					d'^5_{n-4}	d'^6_{n-4}	d^7_n
f_{n-3}				d'^4_{n-3}	d'^5_{n-3}	$\mathbf{d}'^6_{n-3}$	$\mathbf{d}^7_n$
f_{n-2}			d'^3_{n-2}	$\mathbf{d}'^4_{n-2}$	$\mathbf{d}'^5_{n-2}$	d'^6_{n-2}	d^7_n
f_{n-1}		$\mathbf{d}'^2_{n-1}$	$\mathbf{d}'^3_{n-1}$	d'^4_{n-1}	d'^5_{n-1}	d'^6_{n-1}	d^7_n
$\mathbf{f}_n$	$\mathbf{d}^1_n$	d^2_n	d^3_n	d^4_n	d^5_n	d^6_n	d^7_n

. (26)

The reconstruction is performed by repeated use of (6). The relations that define the row-by-row reconstructions in the table are summarized below:

$n-1$
$d'^6_{n-1} = d^6_n - d^7_n$
$d'^5_{n-1} = d^5_n - \left(d'^6_{n-1}\right)$
$d'^4_{n-1} = d^4_n - \left(d'^5_{n-1}\right)$
$\mathbf{d}'^3_{n-1} = d^3_n - \left(d'^4_{n-1}\right)$
$\mathbf{d}'^2_{n-1} = d^2_n - \left(d'^3_{n-1}\right)$

(27)

$n-2$
$d'^6_{n-2} = \left(d'^6_{n-1}\right) - d^7_n$
$\mathbf{d}'^5_{n-2} = \left(d'^5_{n-1}\right) - \left(d'^6_{n-2}\right)$
$\mathbf{d}'^4_{n-2} = \left(d'^4_{n-1}\right) - \left(d'^5_{n-2}\right)$
$d'^3_{n-2} = \left(d'^3_{n-1}\right) - \left(d'^4_{n-2}\right)$

(28)

$n-3$
$\mathbf{d}'^6_{n-3} = \left(d'^6_{n-2}\right) - d^7_n$
$d'^5_{n-3} = \left(d'^5_{n-2}\right) - \left(d'^6_{n-3}\right)$
$d'^4_{n-3} = \left(d'^4_{n-2}\right) - \left(d'^5_{n-3}\right)$

(29)

$n-4$
$d'^6_{n-4} = \left(d'^6_{n-3}\right) - d^7_n$
$d'^5_{n-4} = \left(d'^5_{n-3}\right) - \left(d'^6_{n-4}\right)$

(30)

Instead of using the differences in the first row, consider the bold differences in (26). Those differences can be evaluated with the following *parallel* additions $7+6 \to 6$, $5+4 \to 4$, $3+2 \to 2$, and $1+0 \to 0$. In preparation for the next computation, the subsequent even-to-odd additions, $6{+}5 \to 5$, $4{+}3 \to 3$, $2{+}1 \to$

1, can also be performed in parallel. This pattern of alternate parallel odd and even additions is easily overlooked, but Charles Babbage discovered it in his design of the original difference engine. A refined mechanical implementation of it was used in the design of DE2. In modern computer terminology, the parallel operations are referred to as pipelining.

To summarize the complete DE2 operation, the following precomputations set up the differences that initiate a DE2 operating sequence:

$$\mathbf{f}_n = f_n,\ \mathbf{d}_n^1 = d_1, \mathbf{d}_7 = d_7$$

$$\begin{aligned}
\mathbf{d}_6 &= d_6 - d_7 \\
\mathbf{d}_5 &= d_5 - d_6 \\
\mathbf{d}_4 &= d_4 - d_5 \\
\mathbf{d}_3 &= d_3 - d_4 \\
\mathbf{d}_2 &= d_2 - d_3 \\
\\
\mathbf{d}_6 &= \mathbf{d}_6 - d_7 \\
\mathbf{d}_5 &= d_5 - \mathbf{d}_6 \\
\mathbf{d}_4 &= \mathbf{d}_4 - \mathbf{d}_5 \\
\\
\mathbf{d}_6 &= \mathbf{d}_6 - d_7
\end{aligned} \tag{31}$$

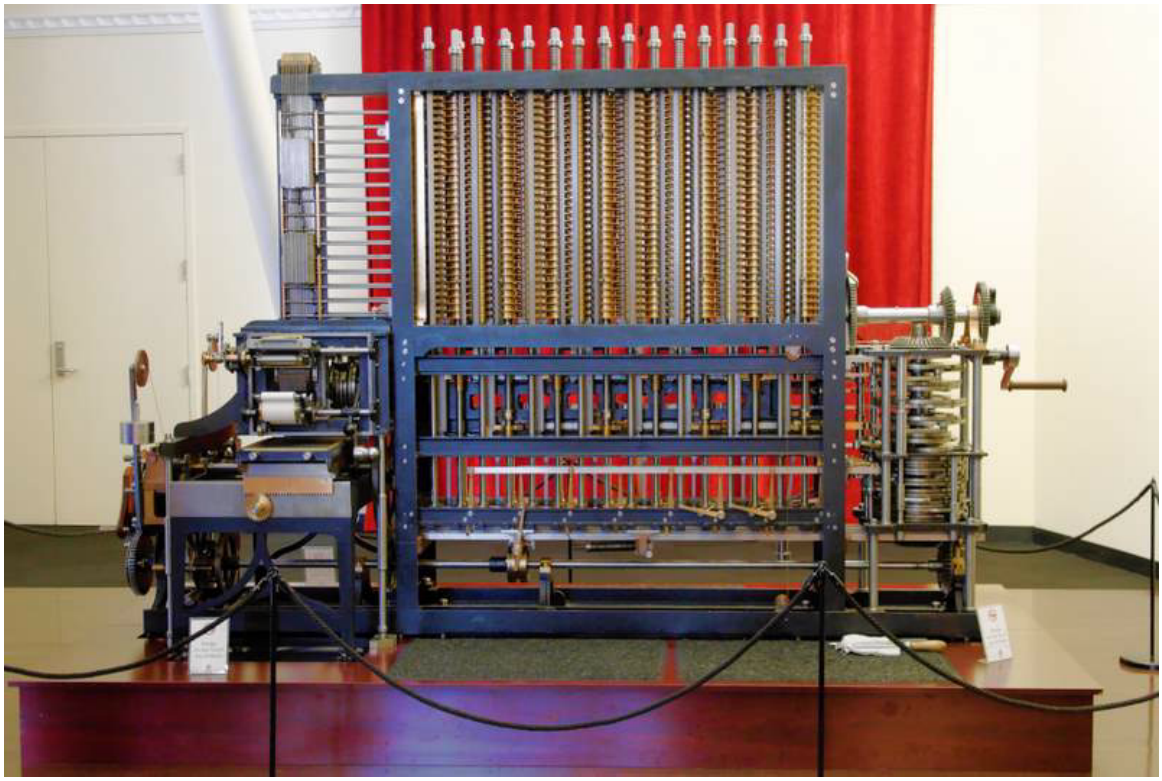
The precomputations are in a slightly different order to reflect their derivation, but the results are identical to those on the website `http://ed-thelen.org`.[2]

With the modified register contents denoted $\mathbf{d}_n$, the elegantly simple DE2 parallel half-cycle operations are defined as follows:

$$\begin{aligned}
&\text{ODD-TO-EVEN} \\
\mathbf{d}'_6 &\leftarrow d_7 + \mathbf{d}_6 \\
\mathbf{d}'_4 &\leftarrow \mathbf{d}_5 + \mathbf{d}_4 \\
\mathbf{d}'_2 &\leftarrow \boldsymbol{d}_3 + \mathbf{d}_2 \\
\mathbf{d}'_0 &\leftarrow d_1 + d_0
\end{aligned} \tag{32}$$

$$\begin{aligned}
&\text{EVEN-TO-ODD} \\
\mathbf{d}'_5 &\leftarrow \mathbf{d}'_6 + \mathbf{d}_5 \\
\mathbf{d}'_3 &\leftarrow \mathbf{d}'_4 + \mathbf{d}_3 \\
\mathbf{d}'_1 &\leftarrow \mathbf{d}'_2 + \mathbf{d}_1
\end{aligned} \tag{33}$$

[2]The development presented here was suggested by CHM docent Lowell Krasner, who said it emerged from discussion with Tim Robinson and the CHM maintenance crew. CHM docent Dick Guertin uses a cardboard cutout to highlight the difference table relation that defines the pipelining operation.



Evaluating (31), followed by repeated evaluation of (32) and (33) will extend the approximating polynomial exactly, until one of the columns generates a carry that propagates to wheel 31.

3 DE2 Emulation

The photograph above shows DE2 ready for demonstration. The crank on the right operates the engine through a 4:1 reduction gear. The computation section is contained in the central rectangular frame. The printing section is on the left. The 8 vertical columns of bronze *figure wheels* are effectively 31-digit *registers* that store the current *state* of the difference engine. The contents of the registers from left to right are denoted d_0, d_1, d_2, d_3, d_4, d_5, d_6, and d_7, with d_0 representing the current value of the polynomial. Indirect emulation of the DE2 operation would perform the mathematical operations defined by (32) and (33) as 31-digit integers. Here direct emulation of the digit-by-digit operations performed by the DE2 will be pursued. The actual mechanical operation of the DE2 will be described only in enough detail to define the emulation. The detailed mechanical operation of the DE2 can be found in documents on the

website `http://ed-thelen.org`.

3.1 Background

The addition operations defined by (32) and (33) formally take the contents of the first designated register and add it to the left adjacent register. Each figure wheel has 40 teeth, representing the digits 0 to 9 repeated 4 times. The DE2 wheel-by-wheel additions are performed through intermediate *sector* gears. Rotation of the adding wheels from their current value to zero transfers the value to the receiving wheels through the sector gears. The sector gears serve two functions. The first is to *remember* the figure wheel values being *given off* to the receiving figure wheels. The sector gears have a stepped thickest. The thicker portion engages the giving off wheel. The thinner portion engages the sector wheel. During the reset phase that follows the giving off phase, the sector column is lifted to disengage it from the receiving wheels while keeping it engaged with the zeroed giving off wheels. The values remembered by the sector wheels are transferred back to the giving off wheels.

The complete addition cycle is performed in two phases. During the giving off phase of the addition cycle, each wheel of the first active column is added to the corresponding wheel of the second active column as just described. If the addition causes a receiving wheel to pass 9, a carry warning is set by a tab on the figure wheel. In the DE2 each wheel that can receive an addition has a carry lever with four positions: unwarranted, warned, carried, and disconnected. The warned state indicates that a carry occurred during the giving off phase of the addition cycle.

During the carry phase of the addition cycle when the receiving wheels are disengaged and free to turn, the carry warning states are *polled* and serviced. The servicing operation is conceptually straightforward. If the nth carry wheel is warned, the $n{+}1^{\text{th}}$ wheel is advanced one digit, and the carry arm is advanced to the carried position. If the carry addition itself initiates a carry, the $n{+}1^{\text{th}}$ carry arm is advanced to the warned position. The DE2 carry levers are attached to a shaft adjacent to the receiving number-wheel column. The mechanical implementation of the service operation uses a separate shaft with 30 fixed carry arms that trip the warned carry levers to initiate the carry operation. A direct sequential implementation of the carry servicing operations would require 30 arms arranged with 12 degree separation. Babbage didn't use this scheme because there is not enough time during a 12 degree rotation to service a carry induced by a carry operation.

The 30 DE2 polling arms are spaced at 22.5 degrees, with an extra 22.5 degrees between the 15th and 16th wheel. With this arrangement, the polling arms can spiral in two identical groups of 15, which means that the upper and lower halves of the column carry warnings are serviced simultaneously. If a carry occurs at wheel 15 in the first rotation it will not be serviced. A second rotation of the carry arms captures and, if necessary, propagates this carry. Since the first carry warnings from the first 15 wheels have already been polled and serviced, the second rotation only affects wheels 16 through 31. The polling

and carry servicing operations are performed on the receiving columns when they are disengaged after the giving off operation.

3.2 Direct DE2 Emulation

The two-phase addition cycle is easily emulated. The 31 column n wheel settings are stored in an integer array $d_n(m)$, where m represents the figure wheel index from 1, (least significant) to 31, (most significant). An auxiliary array, $C(m)$, keeps track of the the carry lever settings, 0, unwarned, 1 warned, 2 carried. Disarmed carries are flagged by a separate flag that overrides specified carry operations. The giving off emulation is an element-by-element modulo 10 addition of $d_n(m)$ to $d_{n-1}(m)$. Where additions would produce a second digit, the corresponding carry array entry is changed from 0 to 1. The carry phase operation uses $d_{n-1}(m)$ and $C(m)$. Carry polling and servicing starts at the first wheel. If a 1 is present, the carry flag is changed to 2 and 1 is added to the next wheel, $d_{n-1}(m+1)$. If that addition causes a carry, $C(m+1)$ is set equal to 1 before it is polled.

To emulate the DE2 implementation, the carry phase operation is first applied to digits 16 to 31 and then to digits 1 to 15. Stopping at A carry at wheel 31 is not serviced. By performing the segmented polling in reverse order, a carry flag at digit 15 will not be serviced until the second cycle is performed. A repeat of the carry phase operations for digits 16 to 31 picks up an unserviced carry at wheel 15. The emulation can be segmented for independent operation by setting all the carry flags at specified wheel location to disengaged with a separate flag.

The DE2 emulation is implemented with the following two subroutines:

```
[da,carry]=GivingOff(d1,d2)
[dc,carry]=CarryPhase(da,carry,nwstart,nwend,DisableFlags)
```

The `GivingOff` routine replaces da with the modulo 10 addition of $d1 + d2$. A carry flag array is generated with zeros indicating no carry or 1 indicating a carry warning. The CarryPhase subroutine implements the carry operations. Carry operations will propagate. The carry flags are serviced from wheel `nwstart` to `nwend`. If the carry flag corresponding to `nwend` is set, $da(nwend + 1)$ in incremented. To service 31 digits, $nwend = 30$. After executing `GivingOff` the carry entries are 0 or 1. After executing `CarryPhase` all the active carry entry 1 values are replaced by 2.

The DE2 emulator implements the numbered serial operations summarized in the following tables. The DE2 executes operations 1, 2, 3, and 4 in parallel during the first quarter of the full cycle. The serial carry phase operations 5 and 6, 7 and 8, 9 and 10, 11 and 12 are performed in parallel during the second quarter cycle. The results are unchanged if operations 5 through 12 are replaced by a single CarryPhase execution for wheels 1 through 31. The operation sequence is repeated for the corresponding even-to-odd addition and carry operations during the second half of the full cycle. In the actual machine operation the answer is transferred to the print section as part of the even-to-odd

half-cycle. The intermediate values of each column and associated carry levers can be extracted at the end of each quarter cycle of the DE2 operation. Although the PC implementation is serial, the computation is fast enough that the intermediate results could be collected and used to derive a graphic simulation of the DE2.

No.	ODD-2-EVEN OPERATIONS
	GIVING OFF
1	$[\texttt{d6}, \texttt{c6}] = \texttt{GivingOff}(\texttt{d6}, \texttt{d7})$
2	$[\texttt{d4}, \texttt{c4}] = \texttt{GivingOff}(\texttt{d4}, \texttt{d5})$
3	$[\texttt{d2}, \texttt{c2}] = \texttt{GivingOff}(\texttt{d2}, \texttt{d3})$
4	$[\texttt{d0}, \texttt{c0}] = \texttt{GivingOff}(\texttt{d0}, \texttt{d1})$
	CARRY ROTATION 1
5	$[\texttt{d6}, \texttt{c6}] = \texttt{CarryPhase}(\texttt{d6}, \texttt{c6}, 16, 31)$
6	$[\texttt{d6}, \texttt{c6}] = \texttt{CarryPhase}(\texttt{d6}, \texttt{c6}, 1, 15)$
7	$[\texttt{d4}, \texttt{c4}] = \texttt{CarryPhase}(\texttt{d4}, \texttt{c4}, 16, 31)$
8	$[\texttt{d4}, \texttt{c4}] = \texttt{CarryPhase}(\texttt{d4}, \texttt{c4}, 1, 15)$
9	$[\texttt{d2}, \texttt{c2}] = \texttt{CarryPhase}(\texttt{d2}, \texttt{c2}, 16, 31)$
10	$[\texttt{d2}, \texttt{c2}] = \texttt{CarryPhase}(\texttt{d2}, \texttt{c2}, 1, 15)$
11	$[\texttt{d0}, \texttt{c0}] = \texttt{CarryPhase}(\texttt{d0}, \texttt{c0}, 16, 31)$
12	$[\texttt{d0}, \texttt{c0}] = \texttt{CarryPhase}(\texttt{d0}, \texttt{c0}, 1, 15)$
	CARRY ROTATION 2
13	$[\texttt{d6}, \texttt{c6}] = \texttt{CarryPhase}(\texttt{d6}, \texttt{c6}, 16, 31)$
14	$[\texttt{d4}, \texttt{c4}] = \texttt{CarryPhase}(\texttt{d4}, \texttt{c4}, 16, 31)$
15	$[\texttt{d2}, \texttt{c2}] = \texttt{CarryPhase}(\texttt{d2}, \texttt{c2}, 16, 31)$
16	$[\texttt{d0}, \texttt{c0}] = \texttt{CarryPhase}(\texttt{d0}, \texttt{c0}, 16, 30)$

(34)

No.	EVEN-2-ODD OPERATIONS
	GIVING OFF
17	$[\texttt{d5}, \texttt{c5}] = \texttt{GivingOff}(\texttt{d5}, \texttt{d6})$
18	$[\texttt{d3}, \texttt{c3}] = \texttt{GivingOff}(\texttt{d3}, \texttt{d4})$
19	$[\texttt{d1}, \texttt{c1}] = \texttt{GivingOff}(\texttt{d1}, \texttt{d2})$
	CARRY ROTATION 1
20	$[\texttt{d5}, \texttt{c5}] = \texttt{CarryPhase}(\texttt{d5}, \texttt{c5}, 16, 31)$
21	$[\texttt{d5}, \texttt{c5}] = \texttt{CarryPhase}(\texttt{d5}, \texttt{c5}, 1, 15)$
22	$[\texttt{d3}, \texttt{c3}] = \texttt{CarryPhase}(\texttt{d3}, \texttt{c3}, 16, 31)$
23	$[\texttt{d3}, \texttt{c3}] = \texttt{CarryPhase}(\texttt{d3}, \texttt{c3}, 1, 15)$
24	$[\texttt{d1}, \texttt{c1}] = \texttt{CarryPhase}(\texttt{d1}, \texttt{c1}, 16, 31)$
25	$[\texttt{d1}, \texttt{c1}] = \texttt{CarryPhase}(\texttt{d1}, \texttt{c1}, 1, 15)$
	CARRY ROTATION 2
26	$[\texttt{d5}, \texttt{c5}] = \texttt{CarryPhase}(\texttt{d5}, \texttt{c5}, 16, 31)$
27	$[\texttt{d3}, \texttt{c3}] = \texttt{CarryPhase}(\texttt{d3}, \texttt{c3}, 16, 31)$
28	$[\texttt{d1}, \texttt{c1}] = \texttt{CarryPhase}(\texttt{d1}, \texttt{c1}, 16, 31)$

(35)

3.3 Demonstration Polynomial Example

The polynomial

$$f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 + a_6 x^6 + a_7 x^7$$

with

$$\begin{aligned} a0 &= 41 \;\; a1 = 4 \;\; a2 = 7 \;\; a3 = 1 \\ a4 &= \;\; 5 \;\; a5 = 9 \;\; a6 = 2 \;\; a7 = 8 \end{aligned}$$

has been used for DE2 demonstrations. Because the coefficients are small, the initial values of the polynomial do not exceed the accuracy of 64 bit double precision arithmetic. However, once initiated, the DE2 emulator its 100% accurate until one the registers overflows. For this demonstration the initiation values are small, whereby the computation can be initialized with standard software.

The register settings at initiation are shown in Figure 1. The initial values were constructed by using the first 8 values of the polynomial ($n = 0, \cdots, 7$) to form a difference table and then following the prescription for precomputation (31). The wheels numbers are summarized in columns from most significant to least significant. The asterisk marks the end of the polynomial computation (wheels 5 to 31). With d1 wheel 1 set equal to 1, each cycle adds 1 to the d0 column providing a cycle count. Figure 2 shows the state of the difference engine at cycle 1000. Although the polynomial evaluation can be initiated with any sequence of 7 consecutive values of the function, those values must be computed with enough accuracy to support the correct differences to order 7. An initiation using 64 bit double precision arithmetic fails as n approaches 300. For comparison, evaluation of the polynomial with 64 bit double-precision integer arithmetic gives the following result, which is accurate only to 15 decimal places:

$$P(1000) = 8002009005001006900000.$$

The $d0$ row of the emulation in Figure 2 is the exact value. Exact values would continue until a carry occurs at wheel 31.

Wheel	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	*4	3	2	1
d0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	1	0	0	0	0
d1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	6	0	0	0	1
d2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	8	0	0	0	0
d3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	4	6	4	0	0	0	0
d4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	6	0	0	0	0	0
d5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	5	2	4	0	0	0	0	0
d6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	4	4	0	0	0	0	0
d7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	3	2	0	0	0	0	0

Figure 1: Initial state of difference engine for demonstration polynomial evaluation.

Wheel	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	*4	3	2	1
d0	0	0	0	0	0	8	0	0	2	0	0	9	0	0	5	0	0	1	0	0	7	0	0	4	0	4	1	1	0	0	0
d1	0	0	0	0	0	0	0	5	6	1	8	0	3	5	5	4	3	0	3	2	1	1	5	0	0	3	6	0	0	0	1
d2	0	0	0	0	0	0	0	0	0	3	3	6	0	6	0	7	4	0	1	2	0	2	0	8	0	2	8	0	0	0	0
d3	0	0	0	0	0	0	0	0	0	0	0	1	6	8	3	6	0	5	9	4	4	3	8	1	4	6	4	0	0	0	0
d4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6	7	2	0	7	2	7	8	0	0	3	6	0	0	0	0	0
d5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	1	8	1	6	1	5	2	4	0	0	0	0	0
d6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	3	2	1	4	4	0	0	0	0	0
d7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	3	2	0	0	0	0	0

Figure 2: State of difference engine for demonstration polynomial at $n = 1000$.

3.4 Negative Differences

The demonstration polynomial produced an initial state with no negative differences. However, negative differences are not excluded from the finite-difference scheme as described above. Negative numbers must be identified and manipulated accordingly. For example, the initial digit could be used to convey positive or negative values with a 0 or 1. For automated computation, there is a much more efficient procedure. Consider the n-digit negative number

$$N = -a_n 10^n - a_{n-1} 10^{n-1} - \cdots - a_1 10 - a_0. \tag{36}$$

The coefficients a_n represent positive integers. The tens complement is defined as

$$\begin{aligned} NC &= (9 - a_n)\, 10^n + (9 - a_{n-1})\, 10^{n-1} + \cdots + (9 - a_1 10) + (9 - a_0) \\ &= -\,|N| + (10^n - 1)\,. \end{aligned} \tag{37}$$

For numbers in the range $-10^n + 1$ to $10^n - 1$, the complement of a negative number will always have the nth digit equal to 9.

From (37), it follows that adding the complement to any positive number in range will always produce the correct answer with negative number remaining in complement form. Adding negative numbers produces the correct negative number. Thus, to accommodate negative numbers it is only necessary to write the negative numbers in tens complement form and proceed with normal additions. If the polynomial itself produced negative numbers, the difference engine would produce and print the correct answers in tens complement form. Under these conditions a non-zero value of digit 31 would indicate a negative number. Negative logarithms can be avoided by scaling. Negative ranges of trigonometric functions can be avoided by symmetry. In general, negative numbers must be either anticipated and avoided or accepted in tens complement form.

4 Polynomial Approximation

With the complete operation of the DE2 in hand, the issue of polynomial approximation remains. The DE2 will evaluate any polynomial with integer coefficients exactly until an overflow occurs. The functions of interest have a specified ranges for both the function and its independent variable. Scaling must be used to keep the function within the range of the DE2; moreover, some means of converting the cycle count to a value of x must be implemented. These are comparatively straightforward details that can be applied once the approximation procedure is in hand.

As already noted, the DE2 was designed to approximate well behaved mathematical functions such as logarithms and trigonometric functions. The least-squares method is well suited to this kind of problem. It was developed by Carl Fredric Gauss near the end of the 18th century, and it would have been well know in Babbage's time. An approximating polynomial has $p + 1$ unknowns. A least-squares algorithm adapted to the polynomial form used for

demonstrating finite-difference properties is summarized in Section 4.1. The mathematical operations involve Vandermonde matrix manipulations. The size of the Vandermonde matrix is fixed by the number of samples. The Vandermode matrix multiplied by the function is equivalent to computing 8 moments ($8 \times N$ multiply and adds). The desired coefficients are obtained by solving an 8-by-8 system of linear equations (˜8^2 multiply and adds). The scaled form of the Vandermode matrix leads to a well-conditioned system of equations.

Once the polynomial coefficients have been determined, the approximating polynomial must be evaluated 8 times. This is a significant amount of computation at full precision, but the error distribution over the N-sample range of the function is fairly uniform. The procedure might have to be repeated with different coarse samplings to determine the approximation range. That entire operation would have to be repeated until the complete range of the independent variable is covered.

4.1 Least Squares

The least-squares solution to the polynomial approximation problem minimizes the integrated quadratic error measure

$$\epsilon^2 = \sum_{n=0}^{N-1} \left(\sum_{k=0}^{p} a_k n^k - f_n \right)^2 . \tag{38}$$

The solution, obtained by differentiating ϵ^2 with respect to a_k then forcing each differentiation to equal zero, involves a Vandermonde matrix of the form

$$V_N^p = \begin{bmatrix} 1 & 1 & 1 & \cdots & \cdots & 1 \\ 0 & 1 & 2 & & & (N-1)^1 \\ 0 & 1 & 2^2 & & & (N-1)^2 \\ & & & \vdots & & \\ 0 & 1 & 2^p & & & (N-1)^p \end{bmatrix} \tag{39}$$

The system of linear equations that defines the least-squares solution can be written in matrix-vector form as

$$\left[V_N^p V_N^{pT} \right] \overrightarrow{a} = V_N^p \overrightarrow{f} , \tag{40}$$

where $\overrightarrow{a}$ the column vector arrangement of the unknown polynomial coefficients, and $\overrightarrow{f}$ a column vector arrangement of the N values of the function. The formal solution for $N \geq p$ is

$$\overrightarrow{a} = \left[V_N^p V_N^{pT} \right]^{-1} V_N^p \overrightarrow{f} , \tag{41}$$

The matrix $\left[V_N^p V_N^{pT} \right]$, is $p \times p$; however, if $N = p$, the solution reduces to the simpler form

$$\overrightarrow{a} = [V_N^p]^{-1} \overrightarrow{f} . \tag{42}$$

The Vandermonde structure, (41) can be solved more efficiently than the general solution of a system of linear equations. However, with small numbers of function samples, the computation time is dominated by the computation of $\left[V_N^p V_N^{pT}\right]$, and the evaluation of the right-hand side, $V_N^p \overrightarrow{f}$. Each entry of the product vector is a moment of the form

$$M_k = \sum_{n=0}^{N-1} n^k f_n \text{ for } k = 1. \tag{43}$$

Note also that dropping the first column in V_N^p is equivalent to constraining the solution to be zero at $n = 0$.

4.2 Taylor Series

The Taylor series was developed by the English mathematician Brook Taylor in 1715. It would also have been well known in Babbage's time. Taylor's theorem states that any well-behaved function can be represented by an infinite power series of the form

$$f(x) = \sum_{k=0}^{\infty} \frac{1}{k!} \left. f^{(k)}(x) \right|_{x=x_0} (x - x_0)^k . \tag{44}$$

where $f^{(k)}(x)$ denotes the kth derivative of $f(x)$ with respect to the independent variable. The derivatives are evaluated at the starting point of the series. If the Taylor series is truncated at p, the contribution of the neglected remainder can be bounded by the relation

$$R_p = (x - x_0)^{p+1} f^{(p+1)}(\xi) \text{ where } x_0 \leq \xi \leq x. \tag{45}$$

Since derivatives of the functions most commonly tabulated were known, A Taylor series approximation could have been implemented directly. However, because the DE2 effectively increments the index without the offset, the polynomial series must be evaluated in its Maclaurin form

$$f(x + x_0) = \sum_{k=0}^{\infty} \frac{1}{k!} \left. f^{(k)}(x + x_0) \right|_{x=0} x^k. \tag{46}$$

For later reference, consider $\log_{10}(x)$, $\sin(x)$, and $\cos(x)$, where

$$\log_{10}(x) = \ln(x)/\ln(10), \tag{47}$$

with $\ln(x)$ is the natural logarithm of x. The range of $\log_{10}(x)$ logarithm can be extended with decibel scaling:

$$\log_{10}\left(10^{\pm n} x\right) = \pm n + \log_{10} x \text{ for } x \geq 1 \tag{48}$$

Thus, any desired value can be scaled from values within the first decade $1 \leq x < 10$. Trigonometric functions are periodic with repetition at multiples of

π radians, but there is a four-fold symmetry, such that only the range $0 \leq x \leq \pi/2$ need be evaluated. The sine function is a simple translation of the cosine function, but it is instructive to observe how Taylor series preserve known symmetries. One can readily verify the series expansions

$$\log 10(1 + n\Delta x) = \sum_{n=0}^{p} \left[(-\Delta x)^n / \left(n \log(10)\right)\right] n^n, \tag{49}$$

$$\sin(n\Delta x) = \sum_{n=0}^{[p/2]} \left[-(-\Delta x)^{2n+1} / \left(2n+1\right)!)\right] n^{2n+1}, \tag{50}$$

and

$$\cos(n\Delta x) = \sum_{n=0}^{[p/2]} \left[(-\Delta x)^{2n} / \left(2n\right)!)\right] n^{2n}. \tag{51}$$

The square brackets isolate the polynomial coefficients scaled by by a power of the sampling interval. Using scaled coefficients emphasizes the dependence on the sample interval. If Δx is small, the larger coefficients have no effect. If Δx is large, a few large terms dominate the series and the overall fit is poor. This suggests that there is an optimum sample spacing for efficient use of the DE2.

5 DE2 Initiation Examples

To demonstrate end-to-end DE2 operation, we assume that the function to be approximated can be evaluated with high accuracy. For demonstration purposes, the luxury of symbolic computation software will be exploited. However, the least squares computation was a direct implementation of (41). For expediency, evaluation of the approximating polynomial was used as a DE2 operation surrogate. However, the appropriately scaled finite differences could be used for a DE2 demonstration or emulation. The amount of computational effort required to reproduce the initiation sequence, the accuracy of the results, and the stress on the engine caused by the number of carry operations could be monitored.

5.1 Base-10 Logarithms

A demonstration initiation sequence for base-10 logarithms was presented by CHM docent Tim Robinson using Mathematica software utilities. The results presented here are different only in detail because of the assumed form of the approximating polynomial. With $N = 60$, a least squares fit from $1 \leq x < 1.3$ produced the coefficients summarized in the table below. The Taylor series coefficients from (49) is show for comparison. Coefficients 1, 2, 3, and 4 are close to the Taylor Series values. The computations were perform with 27-digit accuracy.

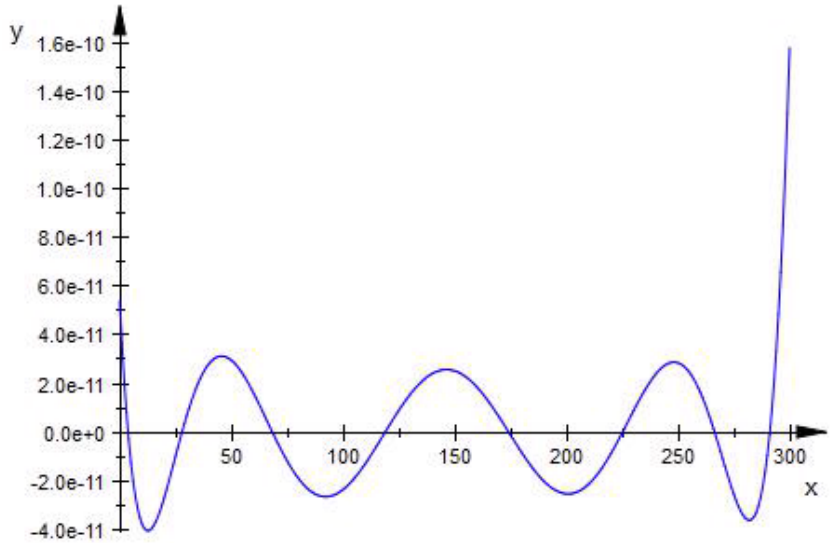


Figure 3: Approximating polynomial errors for base 10 logarithm. The 60 samples spann the range $1 \leq x \leq 1.5$.

#	LLS	Taylor Series (49)
0	0.00000000005341547588036048906052722 34	0
1	0.00217147231645692432214091864	0.00217147240951625913825564459
2	$-0.00000542865023180578876952822433$	$-0.00000542868102379064784563911149$
3	0.000000018091512865872778185235397 9	0.000000018095603412635492818797038 3
4	$-0.000000000067582495019332298871682 6282$	$-0.000000000067858512797383098070488 8936$
5	0.000000000000260977035104324043668396685	0.000000000000271434051189532392281955574
6	$-0.000000090033673518655696064246731 5e \cdot 10^{-8}$	$-0.000000113097521328971830117481489 \cdot 10^{-8}$
7	$0.000000000190292877387147973112440254 \cdot 10^{-8}$	$0.000000000484703662838450700503492097 \cdot 10^{-8}$

To reconstruct the polynomial for DE2 initiation the polynomial is resampled at a finer interval dx,

$$a_k \leftarrow a_k \left(dx/dxLLS\right)^k . \tag{52}$$

Figure 3 shows the errors over the range $1 \leq x \leq 1.3$ at $dx = .001$. The first 8 values could be used to initiate a 300-cycle DE2 operating sequence. The process would have to be repeated approximately 30 times (3000 cycles) to cover the full range of the independent variable.

At this point the hairdressers doing the manual computation might note that the 8 consecutive values of the polynomial they worked so hard to construct are very close to the actual values of the function, as they must be. So why not use 8 the eight easily obtained values of the function itself to initiate the

operation? The first table below shows the differences between the first 8 values from the LLS polynomial and the exact values (to 27 decimal places). To evaluate what happens when exact values of the function are used to construct finite differences, the corresponding polynomial coefficients were computed using (21). For comparison, the Taylor series coefficients are listed. Recall that our definition of the coefficients carry the sampling interval scaling. The values are scaled accordingly from the values in the previous table. As might be expected, the finite difference coefficients differ from the exact Taylor series coefficients only to the extend that the properly normalized finite differences differ from the exact derivatives.

#	Approximating Polynomial	Exact
0	0.000000000053415475880360489060527223 4	0
1	0.00043407751532164290324322528 4	0.00043407747931864066892138777 8
2	0.00086772155209053694655458846 3	0.0008677215312269124928427079
3	0.00130093302823451968511145446	0.00130093302041811880082627886
4	0.00173371280569073833626761169	0.00173371280900052976802710616
5	0.0021660617438310438684756772	0.0021660617565076762304206377 6
6	0.0025979806994718680906174785 5	0.0025979807199085923119629 85
7	0.0030294705268840597448646425	0.0030294705536180071693257673 8

#	Coefficients from (21)	Taylor Series (49)
0	0	0
1	0.00043429448190325129583786022 5	0.00043429448190325182765112891 9
2	$-0.0000002171472409505334691161756 07$	$-0.0000002171472409516259138255644 59$
3	0.000000000144764826492415415155735737	0.000000000144764827301083942550376306
4	$-0.000000000000010857333934724602379310691$	$-0.000000000000010857362047581295691278223$
5	$0.000086810336239111064572505761 6 \cdot 10^{-12}$	$0.000086858896380650365530225783 8 \cdot 10^{-12}$
6	$-0.000000068440262138147467396720458 1 \cdot 10^{-12}$	$-0.000000072382413650541971275188153 2 \cdot 10^{-12}$
7	$-0.000000000053779869666939704067 83 \cdot 10^{-12}$	$0.000000000006204206884332168966444698 84 \cdot 10^{-12}$

6 MATLAB Scripts

The attached program listings include the MATlAB routines used to implement the direct emulator and the mupad scripts used for the symbolic math computations. The symbolic math computations were performed using the MATLAB Symbolic Math Toolbox. The toolbox uses a language called mupad. The attached scripts were written using simple structures that are easily reproduced in other languages. No special routines were used except the implicit linear equation solver were used.

References

[1] Doron Swade. *The Cogwheel Brain.* Little, Brown, and Company, 2000.

[2] Doron D. Swade. The construction of charles babbage's difference engine no. 2. *Annals Hist. Comput.*, (1058-6180/05):70–87, 2005.

[3] I. Grattan-Guinness. Work for the hairdressers: The production of de prony's logarithmic and trigonometric tables. *Annals Hist. Comput.*, 12(3):177–185, 1990.

[4] Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover Publications, Inc., 1965.

```
function DE2Emulator(varargin)
%Emulator for Babbage Difference Engine 2
%Computer History Museum
%Author Chuck Rino
%CHM Docent
%April 2012

%USER INPUTS
Disable1=ones(1,31);  %Carry Levers disabled by replacing 1's with 4

%fprintf('DISABLED CARRY LEVERS D1 12, 15, 16 \n')
%Disable1(12)=4; Disable1(15)=4; Disable1(16)=4;
%nDisable1= Disable1==4;

%Identify cycles for summary output
ncycleP1=1;
ncycleP2=9999;  %MAXIMUM # OF CYCLES BEFORE COUNTER OFERFLOWS
fprintf('DUMP CYCLES %8i TO %8i \n',ncycleP1,ncycleP2)

if isempty(varargin)
    ID=[];
else
    ID=input('Input CR for screen dump only, 1 for file dump ');
end
if isempty(ID)
    fid=1;
else
    fid=fopen('DE2out.txt','w');
end
%Precomputation of finite differences: d(nvals,norder+1)

%Order 7 polynomial:
norder=7;
a0=41.0; a1=4.0; a2=7.0; a3=1.0;
a4=5.0;  a5=9.0; a6=2.0; a7=8.0;
a=[a0;a1;a2;a3;a4;a5;a6;a7];

if isempty(varargin)
    ncycle=[];
else
    ncycle=input('Input current cycle 0<ncycle<=200 CR=>1 ');
end
if isempty(ncycle)
    ncycle=1;
end
n=ncycle-1:ncycle+norder;  %Setup for previous cycle
f=polynomial(n,a);
nvals=length(f);

%Generate finite difference table
d=zeros(nvals,norder+1);
```

```
d(:,1)=f;
for ndiff=2:norder+1
    for nval=1:nvals-ndiff+1
        d(nval,ndiff)=d(nval+1,ndiff-1)-d(nval,ndiff-1);
    end
end
D=d(1,:);

%Precomputation for parallel operation
D(7)=D(7)-D(8);
D(6)=D(6)-D(7);
D(7)=D(7)-D(8);
D(5)=D(5)-D(6);
D(4)=D(4)-D(5);
D(6)=D(6)-D(7);
D(3)=D(3)-D(4);
D(5)=D(5)-D(6);
D(7)=D(7)-D(8);

%Load the 31 digit figure wheel registers
%Offset first noff wheels for cycle count
noff=4;
d0=num2DE2col(D(1),noff);
d1=num2DE2col(D(2),noff);
d2=num2DE2col(D(3),noff);
d3=num2DE2col(D(4),noff);
d4=num2DE2col(D(5),noff);
d5=num2DE2col(D(6),noff);
d6=num2DE2col(D(7),noff);
d7=num2DE2col(D(8),noff);
d1(1)=1;
strdm=fliplr(num2str(ncycle-1));
nn=length(strdm);
for nc=1:nn
    d0(nc)=str2num(strdm(nc));
end
DE=[d0',d1',d2',d3',d4',d5',d6',d7'];
DisplayRegisters2(DE,ncycle-1,fid,f)

CRANK=[];
z=zeros(1,31);
while isempty(CRANK) %Execute one cycle
    DE=[d0',d1',d2',d3',d4',d5',d6',d7'];
    %Odd to Even 1/2 Cycle  (Parallel)
    [d6,c6]=GivingOff(d6,d7);
    [d4,c4]=GivingOff(d4,d5);
    [d2,c2]=GivingOff(d2,d3);
    [d0,c0]=GivingOff(d0,d1);

    %Odd to Even 1/2 Cycle Carry (Parallel)
    %Single pass carry phase
```

```
    [d6,c6]=CarryPhase(d6,c6,1,31);
    [d4,c4]=CarryPhase(d4,c4,1,31);
    [d2,c2]=CarryPhase(d2,c2,1,31);
    [d0,c0]=CarryPhase(d0,c0,1,31);

    %Even to odd 1/2 Cycle  (Parallel)
    [d5,c5]=GivingOff(d5,d6);
    [d3,c3]=GivingOff(d3,d4);
    [d1,c1]=GivingOff(d1,d2);
    %Even to odd 1/2 Cycle Carry (Parallel)
    [d5,c5]=CarryPhase(d5,c5,1,31);
    [d3,c3]=CarryPhase(d3,c3,1,31);
    [d1,c1]=CarryPhase(d1,c1,1,31,Disable1);
    DE=[d0',d1',d2',d3',d4',d5',d6',d7'];

    if ncycle>=ncycleP1 && ncycle<=ncycleP2
        DisplayRegisters2(DE,ncycle,fid,polynomial(ncycle,a))
    end
    fprintf('End Cycle %8i 64-BIT POLYNOMIAL=%27.0f \n',ncycle,polynomial
(ncycle,a));
    %CRANK=input('CRANK ');
    CRANK=1;  %TEMPORARY REPLACEMENT FOR KEYBOARD INPUT
    ncycle=ncycle+1;
end
if fid~=1
    fclose(fid);
end
    function [result,carry]=GivingOff(d1,d2)
        % GIVING OFF PHASE
        % Right to left modulo10 add and replace (d1<=d2+d1) with carry flags warned

        carry=zeros(1,31);                      %carry==1 => warned carry==2 =>
carried  0=> unarmed
        dtemp=d2+d1;                            %Add d2 figure wheels to d1 figure
wheels
        carry(dtemp>9)=1;                       %Set carry warnings
        result=mod(dtemp,10);                   %Truncate to 1st digit (d1
intermediate result)
        %End of first half of add cycle
    end

    function [result,carry]=CarryPhase(dtemp,carry,nwstart,nwend,varargin)
        % CARRY PHASE
        % service carry flags
        if isempty(varargin);
            DE=ones(1,31);
        else
            DE=varargin{1};
        end
        for nwheel=nwstart:nwend                        %Pole carries from giving
off phase
```

```
            if carry(nwheel)==1 && DE(nwheel)==1    %Check for pending carry if
~disabled
                dtemp(nwheel+1)=dtemp(nwheel+1)+1;  %Service pending carry
                carry(nwheel)=2;                    %Completed
                if dtemp(nwheel+1)>9
                    carry(nwheel+1)=1;
                end
                dtemp(nwheel+1)=mod(dtemp(nwheel+1),10);
            end
        end
        result=dtemp;
    end

    function f=polynomial(x,a)
        x=x(:);
        f=zeros(length(x),1);
        nn=length(a);
        for n=0:nn-1
            f=f+a(n+1).*x.^n;
        end
    end

    function DisplayRegisters2(DE,~,fid,varargin)
        %
        %    DE is 31x8 <= wheel 1,2,..,31 by
        %
        if isempty(varargin)
            poly=[];
        else
            poly=varargin{1};
        end
        DID=cell(1,8);
        DID{1}='d0'; DID{2}='d1'; DID{3}='d2'; DID{4}='d3';
        DID{5}='d4'; DID{6}='d5'; DID{7}='d6'; DID{8}='d7';
        fprintf(fid,...
            '\nWheel  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12
11 10  9  8  7  6  5 *4  3  2  1\n');
        for nd=1:8
            fprintf(fid,'  %s  ',DID{nd});
            fprintf(fid,'%3i',flipud(DE(:,nd)));
            fprintf(fid,'\n');
        end
        %if mod(N+1,3)==0
        %    fprintf(fid,'\n\n\n\n\n\n\n\n');
        %end
        %fprintf(fid,'\n64-BIT POLYNOMIAL=%27.0f \n',poly);
    end
end
%REPLACEMENT CODE FOR 2-Cycle Carry Phase Emulation
%[d6,c6]=CarryPhase(d6,c6,16,31);
%[d6,c6]=CarryPhase(d6,c6,1,15);
```

```
%[d4,c4]=CarryPhase(d4,c4,16,31);
%[d4,c4]=CarryPhase(d4,c4,1,15);
%[d2,c2]=CarryPhase(d2,c2,16,31);
%[d2,c2]=CarryPhase(d2,c2,1,15);
%[d0,c0]=CarryPhase(d0,c0,16,31);
%[d0,c0]=CarryPhase(d0,c0,1,15);
%Odd to Even 1/2 Cycle Carry 2
%[d6,c6]=CarryPhase(d6,c6,16,31);
%[d4,c4]=CarryPhase(d4,c4,16,31);
%[d2,c2]=CarryPhase(d2,c2,16,31);
%[d0,c0]=CarryPhase(d0,c0,16,31);
%[d5,c5]=CarryPhase(d5,c5,16,31);
%[d5,c5]=CarryPhase(d5,c5,1,15);
%[d3,c3]=CarryPhase(d3,c3,16,31);
%[d3,c3]=CarryPhase(d3,c3,1,15);
%[d1,c1]=CarryPhase(d1,c1,16,31,Disable1);
%[d1,c1]=CarryPhase(d1,c1,1,15,Disable1);

%Even to odd 1/2 Cycle Carry 2
%[d5,c5]=CarryPhase(d5,c5,16,31);
%[d3,c3]=CarryPhase(d3,c3,16,31);
%[d1,c1]=CarryPhase(d1,c1,16,31,Disable1);
```

```
DUMP CYCLES        1 TO     9999

Wheel  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7
6  5 *4  3  2  1
  d0    0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
4  1  0  0  0  0
  d1    0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
3  6  0  0  0  1
  d2    0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
2  8  0  0  0  0
  d3    0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  4
6  4  0  0  0  0
  d4    0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  3
6  0  0  0  0  0
  d5    0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  5  2
4  0  0  0  0  0
  d6    0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  4
4  0  0  0  0  0
  d7    0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  4  0  3
2  0  0  0  0  0

Wheel  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7
6  5 *4  3  2  1
  d0    0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
7  7  0  0  0  1
```

```
  d1    0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  5
2  8  0  0  0  1
  d2    0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  4
9  2  0  0  0  0
  d3    0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  7  0
6  4  0  0  0  0
  d4    0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  5  6
0  0  0  0  0  0
  d5    0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  5  7  0
0  0  0  0  0  0
  d6    0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  4  1  7
6  0  0  0  0  0
  d7    0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  4  0  3
2  0  0  0  0  0
End Cycle        1 64-BIT POLYNOMIAL=                        77
```

Published with MATLAB® 7.14

Babbage Difference Enginge 2
Least squares estimation of polynomial approximation to f=log10(1+x)

```
reset: p:=7: DIGITS:=27:
```

Generate y=f(x) for x=x_start to x_stop step dx
Adjust x_stop for nsamp from x_start to x_stop

```
x_start:=0: x_stop:=0.3: dxLLS:=.005:
nsamp:=round((x_stop-x_start)/dxLLS): x_stop:=x_start+nsamp*dxLLS:
delete y:           y=hfarray(0..nsamp-1):
n:=0:
for x from x_start to x_stop step dxLLS do
    y[n]:=ln(x+1)/float(ln(10)): n:=n+1:
end_for: nsamp:n-1: print("Number of samples=",nsamp)
```

"Number of samples=", 60

fn to vector-indexed 1 to nsamp

```
delete fn: fn:=matrix(nsamp,1):
for n from 0 to nsamp-1 do
    fn[n+1]:=y[n]:
end_for:
```

V matrix (nsamp x p+1)

```
delete V: V:=matrix(nsamp,p+1):
```

```
for ncol from 0 to nsamp-1 do
     for nrow from 0 to p do
        V[ncol+1,nrow+1]:=ncol^nrow:
  end_for:
end_for:
```

Least Squares Solution for polynomial coefficients

```
delete A: A:=(transpose(V)*V)^(-1)*(transpose(V)*fn):
```

Change array variable

```
delete a: a=hfarray(0..p):
for n from 0 to p do
    a[n]:=A[n+1]:
end_for: a
```

0	0.000000000053415475880360489060527223 4
1	0.002171472316456924322140918 64
2	−0.000005428650231805788769528 22433
3	0.000000018091512865872778185235 3979
4	−0.000000000067582495019332298871682 6282
5	0.000000000000260977035104324043668396685
6	−0.000000000000000900336735186556960642467315
7	0.000000000000000001902928773871479731124402 54

Taylor Coeffieients

```
delete aa: aa=hfarray(0..p):
for n from 1 to p do
    aa[n]:=-(-dxLLS)^n/n/float(ln(10)):
end_for: aa[0]:=0:
```

```
aa
```

0	0
1	0.00217147240951625913825564459
2	$-0.00000542868102379064784563911149$
3	0.000000018095603412635492818797038 3
4	$-0.0000000000678585127973830980704888936$
5	0.000000000000271434051189532392281955574
6	$-0.0000000000000011309752132897183011748 1489$
7	0.000000000000000004847036628384507005034920 97

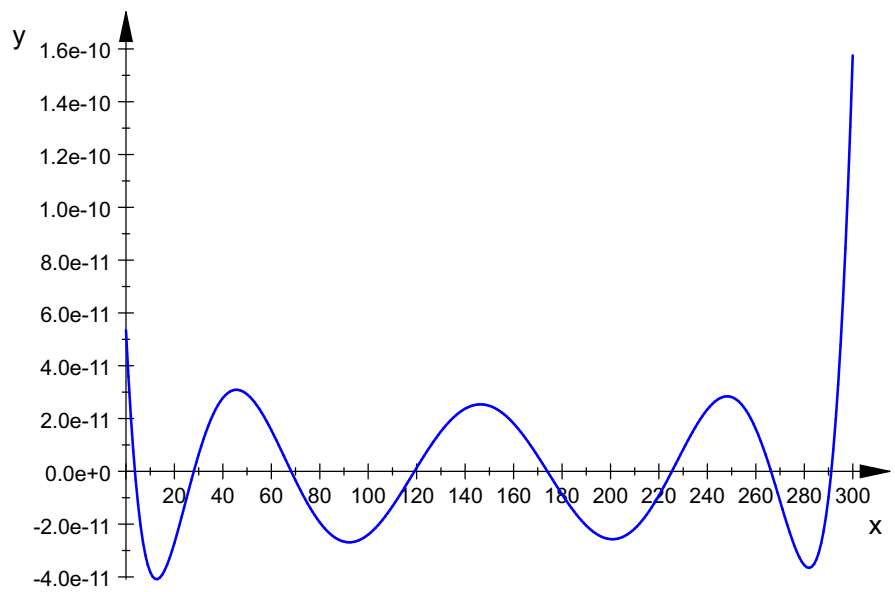
Compute the error at dx

```
x_start:=0: x_stop:=0.3: dx:=.001:
nsamp:=round((x_stop-x_start)/dx): x_stop:=x_start+nsamp*dx:
delete z:            z=hfarray(0..nsamp-1):
n:=0:
for x from x_start to x_stop step dx do
    z[n]:=ln(x+1)/float(ln(10)): n:=n+1:
end_for: nsamp:=n-1: print("Number of samples=",nsamp):
```

"Number of samples=", 300

```
delete g: g=hfarray(0..nsamp):
for nn from 0 to nsamp do
      g[nn]:=0:
      for mm from 0 to p do
         g[nn]:=g[nn]+a[mm]*(nn*dx/dxLLS)^mm:
      end_for:
      g[nn]:=(g[nn]-z[nn]):
end_for:
```

```
plot(g)
```




```
delete d: d=hfarray(0..p):
for nn from 0 to p do
      d[nn]:=0:
      for mm from 0 to p do
         d[nn]:=d[nn]+a[mm]*(nn*dx/dxLLS)^mm:
      end_for:
end_for: d
```

0	0.000000000053415475880360489060527223 4
1	0.00043407751532164290324322528 4
2	0.00086772155209053694655458846 3
3	0.00130093302823451968511145446
4	0.00173371280569073833626761169
5	0.00216606174383104386847567 72
6	0.00259798069947186809061747855
7	0.00302947052688405974486464 25

Babbage Difference Enginge 2
Direct DE2 Estimation of f=log10(1+x)

```
reset: p:=7: DIGITS:=27:
```

Generate y=f(x) for x=x_start to x_stop step dx
Adjust x_stop for nsamp from x_start to x_stop

```
x_start:=0: x_stop:=0.3: dx:=.001:
nsamp:=round((x_stop-x_start)/dx): x_stop:=x_start+nsamp*dx:
delete z:           z=hfarray(0..nsamp-1):
n:=0:
for x from x_start to x_stop step dx do
    z[n]:=ln(x+1)/float(ln(10)): n:=n+1:
end_for: nsamp:=n-1: print("Number of samples=",nsamp):
```

"Number of samples=", 300

Generate finite difference table

```
delete d: d:=hfarray(0..p,0..p):
for m from 0 to p do
    d[m,0]:=z[m]:
end_for:
for n from 1 to p do
    for m from 0 to p-n do
        d[m,n]:=d[m+1,n-1]-d[m,n-1]:
    end_for:
end_for: d[0..p,0..p]:
```

Extract difference d0,d1,...,d7

```
delete dd: dd=hfarray(0..p):
for n from 0 to p do
    dd[n]:=d[0,n]:
end_for: dd
```

0	0.0
1	0.0004340774793186406572791102 84
2	−0.0000004334274103688032925285 61144
3	0.0000000008646933032335595015 19422952
4	−0.0000000000025850328821963586278798175
5	0.0000000000000102931443750342577203 58255
6	−0.0000000000000000050090140368830304851144 3194
7	−0.0000000000000000000271050543121376108501 8632

Recover polynomial coefficients

```
delete a: a=hfarray(0..p):
S:=0:
for n from 0 to p do
      S:=S+(-1)^n*(fact(p)/fact(p-n)/fact(n))*n^p:
end_for:
a[p]:=dd[p]/S/(-1)^p:
for j from 1 to p do
    Sum:=0:
    for l from p-j+1 to p do
        Sum0:=0:
        for lp from p-j to l do
            S:=0:
            for n from 0 to p-j do
                S:=S+(-1)^n*(fact(p-j)/fact(p-j-n)/fact(n))*n^lp:
            end_for:
            Sum0:=Sum0+(-1)^lp*(fact(l)/fact(l-lp)/fact(lp))*(p-j)^(l-lp)*S:
        end_for:
        Sum:=Sum+a[l]*Sum0:
    end_for:
    S:=0:
```

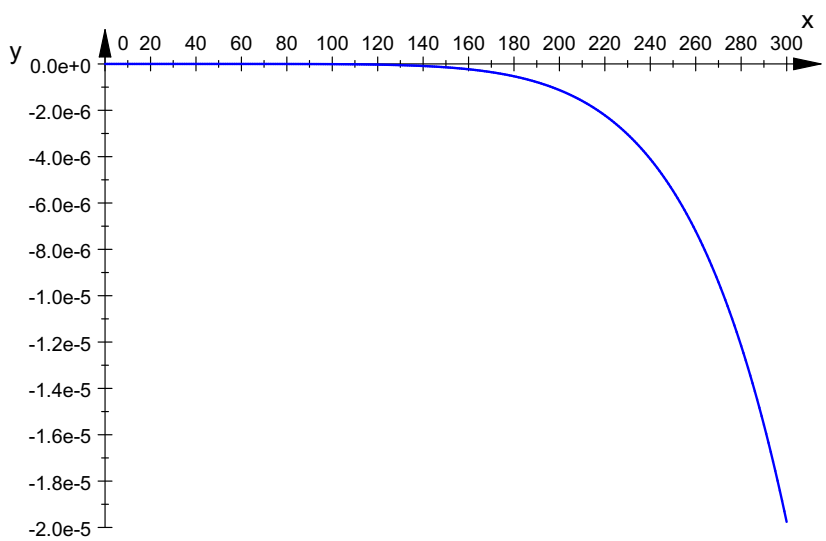
```
    for n from 0 to p-j do
      S:=S+(-1)^n*(fact(p-j)/fact(p-j-n)/fact(n))*n^(p-j):
    end_for:
    a[p-j]:=(dd[p-j]-Sum)/((-1)^(p-j)*S):
end_for:
```

```
a
```

0	0.0
1	0.00043429448190325129583786022 5
2	−0.00000021714724095053346911617 5607
3	0.00000000014476482649241541515 5735737
4	−0.00000000000010857333934724602379310691
5	0.00000000000000008681033623911106457250576 16
6	−0.00000000000000000006844026213814746739672 04581
7	−0.00000000000000000000005377986966693970406783

```
delete g: g=hfarray(0..nsamp):
for nn from 0 to nsamp do
      g[nn]:=0:
      for mm from 0 to p do
         g[nn]:=g[nn]+a[mm]*nn^mm:
      end_for:
      g[nn]:=g[nn]-z[nn]:
end_for:
```

```
plot(g)
```



Taylor Coeffieients

```
delete aa: aa=hfarray(0..p):
for n from 1 to p do
    aa[n]:=-(-dx)^n/n/float(ln(10)):
end_for: aa[0]:=0:
```

```
aa
```

0	0
1	0.00043429448190325182765112891 9
2	−0.00000021714724095162591382556 4459
3	0.00000000014476482730108394255037 6306
4	−0.00000000000010857362047581295691278223
5	0.00000000000000008685889638065036553022 57838
6	−0.00000000000000000007238241365054197127518 81532
7	0.00000000000000000000006204206884332168966444 69884

```
delete d: d=hfarray(0..p):
for nn from 0 to p do
    d[nn]:=ln(nn*dx+1)/float(ln(10)):
end_for: d
```

0	0.0
1	0.00043407747931864066892138777 8
2	0.00086772153122691249284270 79
3	0.0013009330204181188008262788 6
4	0.0017337128090005297680271061 6
5	0.0021660617565076762304206377 6
6	0.0025979807199085923119629 85
7	0.0030294705536180071693257673 8