European Society of Computational Methods in Sciences, Engineering and Technology (ESCMSET)

Journal of Numerical Analysis, Industrial and Applied Mathematics (JNAIAM)
vol. 8, no. 1-4, 2013, pp. 1-12
ISSN 1790–8140

Performance Analysis of the Taylor Expansion Coefficients Computation as Implemented by the Software Package TADIFF [1]

Luisa D'Amore[2], Valeria Mele[a] and Almerico Murli[b]

[a]University of Naples Federico II, Naples, ITALY,
[b]Southern Partnership for Advanced Computational Infrastructures (SPACI), c/o University of Naples, Federico II, Naples
and CMCC - Centro Euro-Mediterraneo per i Cambiamenti Climatici, Lecce, ITALY

Received 29 October, 2012; accepted in revised form 24 April, 2013

Abstract: A detailed rounding errors analysis for the computation of Taylor expansion coefficients of an analytic real function with respect to one variable, as implemented by TADIFF, a software package written in C++ specialized for computing Taylor expansion coefficients using Algorithmic Differentiation, is performed. The error analysis is carried out in a finite precision arithmetic system satisfying the IEEE standard 754. Furthermore, time and space complexity of such a computation is discussed. Experimental results aimed to validate both the accuracy and the complexity estimates are presented.

Keywords: Algorithmic Differentiation, Taylor expansion coefficients, TADIFF

Mathematics Subject Classification: AMS: 65G50; 65Y20; 68Q25

1 Introduction

Derivatives play a central role in sensitivity analysis (model validation,...), inverse problems (data assimilation,...) and design optimization (simulation parameter choice,...). At a more elementary level they are used for solving algebraic and differential equations, nonlinear systems, curve fitting and many other applications. Our interest on derivatives arises from the need of computing Taylor expansion coefficients in a mathematical software for inverting Laplace transforms [4, 5, 6].
Here we are concerned with numerical computation of Taylor expansion coefficients by means of Algorithmic Differentiation (AD)[3]. Numerical approximation of derivatives, based on divided differences, is known to be ill posed [1, 3, 10, 13]. Instead, AD strongly reduces the ill posedness of

[1]Published electronically June 15, 2013
[2]Corresponding author. E-mail: luisa.damore@unina.it, Phone: +39 081 675625, Fax: +39 081 7662106,University of Naples Federico II, Naples, ITALY,
[3]There is not even general agreement on the best name for this topic, which is frequently referred to as Automatic or Computational Differentiation in the literature. Following [8] we prefer algorithmic because it emphasized algorithmic structure and viewpoint.

numerical computation of derivatives by divided differences [9].

Over the last decades, several AD tools have been deployed, for pointers on AD literature and AD tools see the web page www.autodiff.org of the AD community. We refer to the software package `TADIFF`, written in C++, and specialized in computing Taylor series expansion coefficients. `TADIFF` implements AD using templates and operator overloading in ANSI C++, no additional library is required and it is free to use [2].

1.1 The present work and motivations

In [9] it is stated that the derivative value of a function defined as a composition of elementary functions obtained by using AD is exact for a perturbation of the elementary components at the level of the machine precision (backward stability á la Wilkinson). Here we analyze how the round-off error affects the computation of the Taylor series coefficients, in a finite precision arithmetic system assuming to satisfy the IEEE 754 standard. Such analysis stems from the observation that the derivatives are explicitly evaluated at a given point, so the result of AD is a purely numerical value[4]. In practical terms this means that AD differentiates any composite function by appropriately combining the derivatives of its constituents, assuming that procedures for their evaluation are already in place. By doing this recursively, AD effectively extends an original procedure for evaluating function values by themselves into one that also evaluates derivatives. This is essentially an arithmetic manipulation, of course, the application of the chain rule does entail multiplications and additions of floating point numbers, which can only be performed with finite precision. In this sense the roundoff error analysis occurs.

Furthermore, the paper addresses the computational cost of Taylor expansion coefficient, both analyzing in details the complexity of the formulae employed by `TADIFF` and measuring the memory usage and the execution time required by `TADIFF` for computing high orders coefficients.

The rest of the paper is organized as follows: next section summarizes the main rules underlying AD, section 3 describes how `TADIFF` computes Taylor expansion coefficients. In section 4 we give preliminaries useful to our analysis and perform the error analysis of the computation of Taylor expansion coefficients. Sections 5 reports performance results both on the memory issues and on the execution time of `TADIFF`, while finally some conclusions are drawn in section 6.

2 Algorithmic Differentiation of `TADIFF`

Before starting with the rounding error analysis, we will summarize the main rules of AD. More precisely, we review the basic idea behind AD as carried on by `TADIFF`.

We consider an analytic real function H:

$$H : x \in \Re^n \mapsto y \in \Re^m, \quad n \geq 1,\ m \geq 1$$

where $x = (x_0, \ldots, x_{n-1})$ is the independent variable and $y = (y_0, \ldots, y_{m-1})$ is the dependent variable.
AD is a recursive procedure that computes the value of the derivatives of certain functions at a given point. The functions considered are those that can be obtained by sum, product, quotient, and composition of elementary functions (elementary functions include polynomials, trigonometric

[4]The point that separates algorithmic differentiation from symbolic differentiation is that the chain rule is applied not to symbolic expression but to actual numerical values, at each step [8].

Evaluation Trace of H	
1.	$v_{0,i} := x_i, \quad i = 0, \ldots, n-1$
2.	$v_{l,i} := G_l(W_{l-1})_{l \prec i} \quad l = 0, \ldots, L$
3.	$y_i := v_{L,i}, \quad i = 0, \ldots, m-1$

Table 1: Evaluation trace of H.

functions, real powers, exponentials, and logarithms). The first step is the automatic generation of the so-called *evaluation trace* of H, which is a mathematical description of the sequence of floating point operations and values actually calculated by a particular implementation used to evaluate the function H.

The evaluation trace of H is made of $L+1$ levels[5] (or, equivalently, steps) $l = 0, \ldots, L$ such that:

1. at level $l = 0$ there are defined $n0 := n$ variables:

$$v_{0,0} := x_0;\ v_{0,1} := x_1; \ldots\ldots; v_{0,n-1} := x_{n-1};$$

2. at level $l = 1$ there are defined $n1$ variables:

$$v_{1,0} := g_0(W_0);\ v_{1,1} := g_1(W_0); \ldots\ldots; v_{1,n1-1} := g_{n1-1}(W_0);$$

3. at level $l = 2$ there are defined $n2$ variables:

$$v_{2,0} := g_0(W_1); v_{2,1} := g_1(W_1); \ldots\ldots; v_{1,n2-1} := g_{n2-1}(W_1);$$

4. at level $l = L$ there are defined $nL := m$ variables:

$$v_{L,0} := g_0(W_{L-1}); v_{L,1} := g_1(W_{L-1}); \ldots\ldots v_{L,nL-1} := g_{nL-1}(W_{L-1});$$

where g_i are *elemental* functions[6], and W_i is a subset of the variables introduced at levels l where $l < i$.
By defining G_l, $l = 0, \ldots, L$ to be the set of elemental functions g_i used at the level l, i.e.:

$$G_l := \{g_i\}_{i=0,\ldots,nl-1}, \quad l = 0, \ldots, L$$

and by introducing the following notation:

$$v_{l,i} := G_l(W_{l-1})_{l \prec i}$$

where the relation $j \prec i$ means that v_i depends on v_j, the evaluation trace of H can be briefly described as shown in Table 1.

[5] We may interpret the set of indices as the vertex set of a directed acyclic graph. When vertices are annotated with the elemental functions g_i the evaluation trace is often called *computational graph*. In this case we refer to levels.

[6] Following [9] we define and elemental function to be either intrinsic functions or arithmetic operations provided by the programming environment used.

The generation of the evaluation trace of H defines $Nvar = \sum_{l=0}^{L} nl$ variables and it is made of three main parts: the first copies the value of independent variables x_i to internal variables $v_{0,i}$, the last section copies the resulting values $v_{L,i}$ into *dependent* variables y_i, the actual calculation is carried out in the midsection of the box. The evaluation trace is interpreted as a sequence of elementary operations and the basic differentiation rules can be applied to compute the derivatives of each intermediate function. Subsequently, the chain rule is employed to compute the derivatives of H with respect to x_i. Since the chain rule is associative one can either propagate derivatives together with the function evaluation.
When the computer is used to evaluate the program implementing the function H, the actual operations performed correspond to the operations in the evaluation trace. That is, the computer will interpret the expression into a list of simple operations similar to those in the code-list.

3 Taylor arithmetic in TADIFF

In this section we review how the software package performs the computation of Taylor coefficients of H using the evaluation trace of H as described in section 2.

Taylor expansion method is a generalization of the *forward method* for AD, where instead of computing only the first derivative, we obtain higher order coefficients using recursive rules. These rules, also called *Taylor arithmetic*, are applied on the evaluation trace of function H in order to obtain the coefficients of all intermediate values. Let us explain how this is done.

Let $\mathbf{x_0} \in \Re^n$ and:

$$\frac{H^{(k)}(\mathbf{x_0})}{k!} := (H)_k, \quad k = 0, 1, 2, \ldots$$

denote the k-th Taylor coefficient of H. Observe that it is:

$$(H)_0 := H(\mathbf{x_0})$$

that is the zero-order derivative is the function value at $\mathbf{x_0}$, and:

$$(H)_k = \frac{1}{k}(H')_{k-1}. \tag{1}$$

Let assume that H is a composite function, i.e. $y = H(x(t))$, by introducing the function w such that $\frac{1}{w} = \frac{dH}{dx}$ and by applying the derivative chain rule, then:

$$(H)' = \frac{dH}{dx}\frac{dx}{dt} = \frac{1}{w}x' \tag{2}$$

so, by (1) and (2), $(H)_k$ is expressed as follows [for details see [2]]:

$$(H)_k = \frac{1}{(w)_0}\left((x)_k - \frac{1}{k}\sum_{j=1}^{k-1} j(H)_j(w)_{k-j}\right), \quad k = 1, 2, 3, \ldots, \tag{3}$$

Formula (3) is used extensively by `TADIFF`.
More precisely, once the evaluation trace of H is generated and the variables $v_{l,i}$, $l = 0, \ldots, L$, $i = 0, \ldots, nl-1$ have been defined as explained in section 2, then formula (3) is applied to compute the Taylor coefficients of each one of these variables and at each level l. Such computation is performed in two steps:

Evaluation Trace of Φ	
1.	$v_0 := x_0,$
2.	$v_i := g_i(v_j)_{j \prec i} \quad i = 1, \ldots, Nvar - 1$
3.	$y_0 := v_{Nvar}$,

Table 2: Evaluation trace of Φ.

1. at level $l = 0$, all variables $v_{0,i}$ are constants, then all but the zero order Taylor coefficient is zero;

2. at levels $l > 0$, variables $v_{l,i}$ depend on elemental functions, then a predefined version of formula (3) is employed. This formula is the elementary rule specialized for the sum/subtraction /multiplication/division of functions or of the mathematical built-in functions (exponential, square root, trigonometric, power, ...).

In the next section performance analysis of formula (3) for a scalar real function is carried on.

4 Efficiency and Accuracy

For purposes of illustration we assume that the function under examination is a scalar real function i.e. we assume $n = m = 1$. Let

$$\Phi : x \in \Re \to y \in \Re$$

be a scalar real function.
The evaluation trace generated by `TADIFF` for computing Taylor coefficients of Φ at x_0 is indicated in Table 2 where $y_0 := \Phi(x_0)$. Observe that, in this case, $n0 = nL = 1$.

In order to analyze how the rounding errors propagate during the computation of Taylor coefficients of Φ, by using formula (3), we need to compute the number of floating point operations required by formula (3), i.e. an estimate of the complexity of this formula. Moreover, we measure the execution time required by TADIFF to compute up to N Taylor expansion coefficients of a test function, that is the efficiency of the software TADIFF.

4.1 Time Complexity

Following [12] we will use the symbol $T(N)$ to denote the time complexity function of a computation of size N, that is the number of floating point operations required to perform the specified computation on a problem of size N.

Proposition 4.1 *Let c_k, $k = 1, \ldots, N$ be the sequence of the first N Taylor coefficients of Φ. For computing $N > 1$ coefficients the following time complexity comes out:*

$$T(N)[c_k]_{k=1,N} = O(K\, N^2) \tag{4}$$

where K is a constant depending on the function Φ and measuring the computational cost of Φ.

Proof: At each level $l = 0, \ldots, L$, as explained section 3, for computing the k-th Taylor expansion coefficient `TADIFF` generates:

1. at level $l = 0$, the k-th Taylor coefficient of the variable $v_0 \in \Re$. This computation has the time complexity:
$$T(1)[v_0] = 2 \quad flops$$
2. at level $nl > 0$, the k-th Taylor coefficient of the $N_e := Nvar - 1$ functions g_i, by using formula (3). This computation has time complexity:
$$T(N_e, nl)[g_i]_{i=1,\ldots,N_e} = nl \cdot \sum_{k=1}^{N_e}(2k+1) = nl\,(N_e(N_e+1)+N_e) = nl(N_e^2+2N_e)\,.$$

Summing $T(N_e, nl)[g_i]_{i=1,\ldots,N_e}$ over the L levels $nl = 1, \ldots, L$, we get

$$T(N_e, L)[g_i]_{i=1,N_e} = \sum_{nl=1}^{L} nl\,\left(N_e^2+2N_e\right) = (N_e^2+2N_e)\sum_{nl=1}^{L} nl = (N_e^2+2N_e)N_e \quad .$$

Because the complexity of the computation of N coefficients is

$$T(N)[c_k]_{k=1,\ldots,N} = N \cdot \sum_{k=1}^{N} T(N_e, L)[g_i]_{i=1,\ldots,N_e} = N\sum_{k=1}^{N}(N_e^3+2N_e^2) = N(N_e^3+2N_e^2)\sum_{k=1}^{N} 1 \quad ,$$

for $K = N_e^3 + 2N_e^2$ formula (4) follows.♣

This result says that the time complexity of the computation of N Taylor expansion coefficients grows quadratically as N, depending linearly on the complexity of the function Φ.

4.2 Efficiency

We performed many experiments aimed to analyze the computational cost - both in terms of memory usage and in terms of execution time - of evaluating derivatives by using `TADIFF`. We consider the function $\Phi(x) = \frac{12.5}{1-x} H\left(\frac{12.5}{1-x} - 3.75\right)$ where

$$H(x) = \frac{8a^3x^2}{(x^2+a^2)^3} \quad (x_0 = 0)$$

We experimentally found that [7] the execution time needed for computing one coefficient is about[8]

$$Time = 1.21e-05 \quad secs$$

and this is also the time required for computing $k+1$ coefficients once k were already computed. This measure does not take into account the time required to the function evaluation, because it depends on the function itself.
Furthermore, in order to measure the growth factor of the execution time as a function of the parameter k (i.e. the number of Taylor coefficients) we measure the overall execution time required for the computation of up to k coefficients, where $k = 1, \ldots, 100$. In figure 1 we show the measured execution time compared with the function $y(k) = k^{1.5}10^{-6}$ suggesting that the growth factor of the execution time as a power function of k is about 1.5, and this is in agreement with the theoretic estimate (4) derived in section 4.1.

[7] Timing results are the medium value of several runs.
[8] We report execution time on an Intel(R) Quad-Core(TM) i7-950, 3.07 GHz, L3 cache size 8 MB, RAM size 12GB.

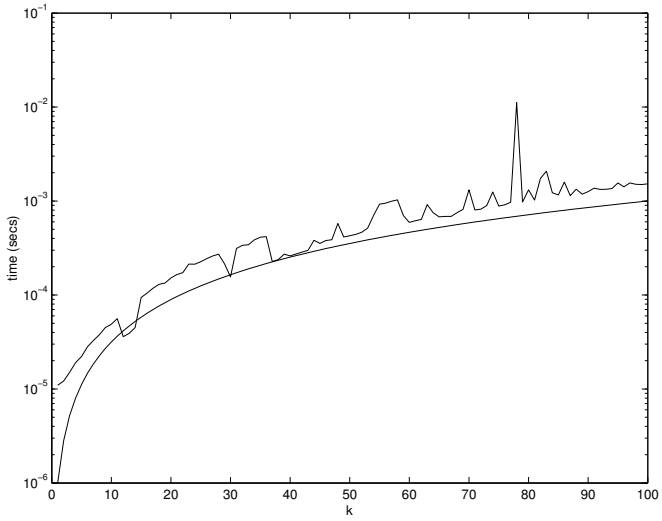

Figure 1: Measured execution time of the computation of k Taylor coefficients, where $k = 1, \ldots, 100$. The time for computing 1 coefficient is $1.21e-05$ secs. The execution time is compared with the function $y = k^{1.5}10^{-6}$.

Concerning the memory issues we found that the amount of memory - measured in terms of number of double precision (8 byte) storage locations - in use by `TADIFF` grows linearly as the number of Taylor expansion coefficients, N, and as the total number of variables introduced by the evaluation trace of the function. More precisely, we experimentally found that:

$$S(N) = N \cdot (Nvar - C) \quad double$$

where $S(N)$ denote the space complexity function as defined in [12], $Nvar$ is the total number of variables used by `TADIFF` and C is the number of constants. For the test function $Nvar = 19$, $C = 9$ then:

$$S(N) = 10k$$

4.3 Rounding Error analysis

In this section we first analyze rounding error propagation during the computation of each one Taylor expansion coefficient by using formula (3), then we derive an upper bound of the rounding error propagated on N Taylor expansion coefficients of the function Φ as introduced in Section 3. We make the following model assumptions on the finite precision arithmetic system[9]:

$$\begin{aligned} &\mathcal{A}_1: \quad fl(x_1 \; op \; x_2 \ldots op \; x_n) = (x_1 \ldots \; op \; x_n)(1+\delta_n), \; |\delta_n| \leq (n-1)u, \\ &\mathcal{A}_2: \quad fl(fun(x)) = fun(x) \cdot (1+\delta_{fun}) \quad , \; |\delta_{fun}| \leq u, \end{aligned} \tag{5}$$

where $n \geq 2$, $fl(z)$ denotes the floating point representation of z, $fun(x)$ denotes an elementary function, $x_i \in \Re$, u is the roundoff unit, *op* denotes a floating point operation.
To analyze the roundoff error propagation on the computed value of c_k we need the following

[9]The IEEE 754 standard supports these model assumptions.

lemmas:

Lemma 4.2 ([11]) *If $|\delta_i| \leq u$ for $i = 1, n$ and $n \cdot u < 1$ then:*

$$\Pi_{i=1}^{n}(1+\delta_i) = 1+\theta_n$$

where

$$|\theta_n| \leq \frac{nu}{1-nu} := \gamma_n \leq 1.01nu$$

♣

In the subsequent analysis it is necessary to manipulate the $1+\theta_k$ and γ_k terms. The next lemma provides the necessary rules.

Lemma 4.3 ([11]) *For any positive integer k let θ_k be such that:*

$$|\theta_k| \leq \gamma_k = \frac{ku}{1-ku}$$

The following relations hold:

$$(1+\theta_k)(1+\theta_j) = 1+\theta_{k+j}$$
$$\gamma_k\gamma_j \leq \gamma_{min(k,j)}, \quad i\gamma_k \leq \gamma_{ik},$$
$$\gamma_k + u \leq \gamma_{k+1}, \quad \gamma_k + \gamma_j + \gamma_k\gamma_j \leq \gamma_{k+j}$$

Next two propositions analyze roundoff propagation of formulae (2) and (3) respectively, for a given value of k.

Proposition 4.4 *Let us consider the Taylor coefficient c_k of the generic elementary function g_l obtained by using formula (1):*

$$c_k := (g_l)_k = \frac{1}{k}(g_l')_{k-1}$$

then it is:

$$|fl(c_k) - c_k| \leq \gamma_2\Gamma(c_k)$$

where $\Gamma(c_k) = \frac{1}{k}|(g_l)_{k-1}|$

Proof: It follows by assumption $\mathcal{A}_1$ taking into account that in order to compute c_k there are needed $T = 2$ $flops$. ♣

Proposition 4.5 *Let us consider the Taylor coefficient c_k of the scalar real function $\Phi : \Re \mapsto \Re$ obtained by using formula (3):*

$$c_k \equiv (\Phi)_k = \frac{1}{(g)_0}\left[(x)_k - \frac{1}{k}\cdot\sum_{j=1}^{k-1} j(H)_j(g)_{k-j}\right]$$

then it is:

$$|fl(c_k) - c_k| \leq \gamma_{2k+1}\Upsilon(c_k)$$

where $\Upsilon(c_k)$ *depends only on* c_k*.*

Proof: We get

$$fl(c_k) = fl\left[\frac{1}{(g)_0}\left((x)_k - \frac{1}{k}\sum_{j=1}^{k-1} j(H)_j(g)_{k-j}\right)\right] =$$

$$= \frac{1}{(g)_0}(1+\delta_1)(1+\delta_2)\left[(x)_k(1+\delta_3) - \frac{1}{k}(1+\delta_4)\cdot\sum_{j=1}^{k-1} j(H)_j(g)_{k-j}(1+\delta_j)(1+\delta_{k-j})\right]$$

where $|\delta_i| \leq u$, for $i = 1, \ldots, k-1$. For our purposes it is not necessary to distinguish between the different δ_i terms, so to simplify the expressions let us drop the subscripts on the δ_i and write $1 + \delta_i = 1 \pm \delta$, where $|\delta| \leq u$. We have:

$$fl(c_k) = \frac{1}{(g)_0}(1\pm\delta)^2\left[(x)_k(1\pm\delta) - \frac{1}{k}(1\pm\delta)\sum_{j=1}^{k-1} j(\Phi)_j(g)_{k-j}(1\pm\delta)^2\right] =$$

$$= \frac{1}{(g)_0}(1\pm\delta)^3\left[(x)_k - \frac{1}{k}\sum_{j=1}^{k-1} j(\Phi)_j(g)_{k-j}(1\pm\delta)^2\right] =$$

Applying Lemma 3,

$$= \frac{1}{(g)_0}(1+\vartheta_3)\left[(x)_k - (1+\vartheta_{2k-2})\frac{1}{k}\sum_{j=1}^{k-1} j(\Phi)_j(g)_{k-j}\right]$$

Then:

$$|fl(c_k) - c_k| \leq \gamma_{2k+1}\frac{1}{|(g)_0|}\left[|(x)_k| + \frac{1}{k}\sum_{j=1}^{k-1} j|(\Phi)_j||(g)_{k-j}|\right]$$

Thesis follows by letting

$$\Upsilon(c_k) = \frac{1}{|(g)_0|}\left[|(x)_k| + \frac{1}{k}\sum_{j=1}^{k-1} j|(\Phi)_j||(g)_{k-j}|\right]$$

♣

Next theorem concerns the rounding error propagation on the Taylor expansion coefficients c_k, $k = 1, \ldots, N$. The error analysis takes into account the error propagation from the beginning of the computation of each coefficient c_k; this means that the upper bound mainly depends on the propagated error on the last coefficient c_N. In other words we get to an uniform upper bound for all the coefficients c_k, $k = 1, \ldots, N$.

Theorem 4.6 *Let $fl(c_k)$ $k = 1, \ldots, N$ be the computed values of expansion coefficients c_k, $k = 1, \ldots, N$, of Φ using evaluation procedure described in Section 3. The following upper bound of the roundoff error introduced in the computation of c_k holds:*

$$|fl(c_k) - c_k| \leq \gamma_{N^2 N_e} \mathcal{G}(c_N), \quad k = 1, \ldots, N \tag{6}$$

where $\lim_{N\to\infty} \mathcal{G}(c_N) = 0$.

Proof:
From the evaluation procedure given in Section 3, we get:

- for computing the coefficient c_k of the N_e functions g_i , by using Proposition 4.4, formula (1) leads to the error amplification factor on c_k:

  $$\gamma_{2N_e} \Gamma(c_k)$$

  where $\Gamma(c_k) = \frac{1}{k}|(g_l)_{k-1}| \quad l = 1, \ldots, N_e$

- for computing the coefficient c_k of the $N_c = l - N_e$ functions g_i, by using Proposition 4.5, formula (3) leads to the error amplification factor:

  $$\gamma_{(2k+1)(l-N_e)} \Upsilon(c_k)$$

In conclusion, for computing N coefficients c_k of Φ the error growth factor is:

$$\sum_{k=1}^{N} \left[\gamma_{2N_e} \Gamma(c_k) + \gamma_{(2k+1)(l-N_e)} \Upsilon(c_k)\right] \leq \gamma_{N^2 N_e} \mathcal{G}(c_N)$$

♣

Figure 1 shows the error introduced by `TADIFF` on the k-th Taylor coefficient, versus the order k. The error is computed assuming as "reference" value the first 32 digits of the k-th Taylor coefficient computed by using the problem solving environment `Mathematica`©, that uses multiprecision.
Test function comes from [7] and it is:

$$\Phi(x) = \frac{12.5}{1-x} H\left(\frac{12.5}{1-x} + 2.5 - 6.25\right)$$

where

$$F(x) = \frac{1}{x(x+1)}\left[\frac{1}{2x} - \frac{e^{-2x}}{1-e^{-2x}}\right]$$

5 Conclusions

The effects of finite number representation are familiar to anyone who has done scientific computing. Rounding error, if it manages to accumulate sufficiently, can destroy a numerical solution. We demonstrate, both theoretically and experimentally, that the rounding error introduced on the k-th derivative, computed by `TADIFF` and using AD, grows quadratically with N, the number of Taylor expansion coefficients. As a consequence, as N grows the rounding error may be amplified significantly. This may happen in presence of Taylor series slowly convergent. Regarding the computational cost, we found that the amount of memory - measured in terms of number of double precision (8 byte) storage locations - in use by `TADIFF` grows linearly as the number of Taylor expansion coefficients, N, and as the total number of variables introduced by the evaluation trace of the function. Finally, the growth factor of the execution time, as a function of N, is 1.5.

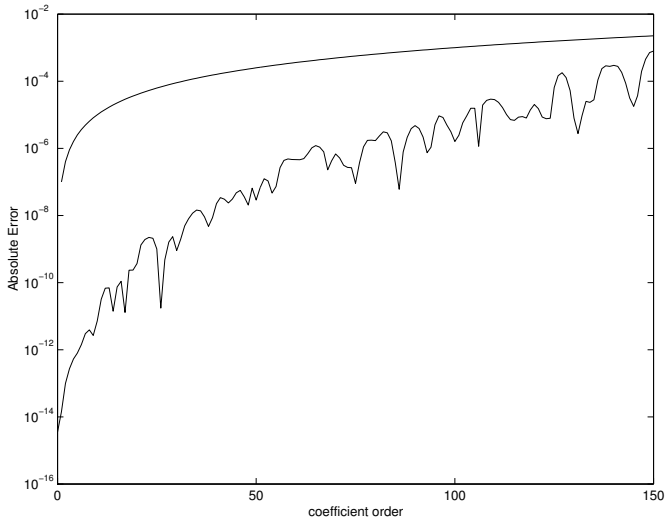

Figure 2: The error introduced by `TADIFF` on the k-th Taylor coefficient, versus the order k. The error is computed assuming as "reference" value the first 32 digits of the k-th Taylor coefficient computed by using the software `Mathematica`©, with multiprecision. The measured error is compared with the graph of the quadratic function $y = 10^{-7}k^2$.

References

[1] ANDERSSEN R.S., de HOOG F.R., 1984. Finite difference methods for the numerical differentiation of non-exact data, *Computing*, 33, pp. 259-267.

[2] BENDTSEN C., STAUNING O., 1997. TADIFF, a flexible C++ package for automatic differentiation, *Technical Report*, IMM - REP- 1997 -07.

[3] CULLUM J., 1971. Numerical differentiation and regularization, *SIAM J. Numer. Analysis* 8, 254-265.

[4] D'AMORE L., CAMPAGNA R., MELE V., MURLI A. 2012 ReLIADiff. A software package for Laplace transform Inversion, *ACM TOMS*, submitted.

[5] CUOMO S., D'AMORE L., RIZZARDI M., MURLI A. 2008. A Modification of Weeks' Method for Numerical Inversion of the Laplace Transform in the Real Case Based on Automatic Differentiation, *Advances in Automatic Differentiation,* Springer, 45-54.

[6] CUOMO S., D'AMORE L., MURLI A., RIZZARDI M. 2007. Computation of the inverse Laplace transform based on a collocation method which uses only real values, *Journal of Computational and Applied Mathematics*, n. 198, 98-115.

[7] DUFFY, D.G. 1993. On the numerical inversion of Laplace Transforms: Comparison of three new methods on characteristic problems from applications, *ACM Transactions on Mathematical Software*, Vol. 19, No. 3, 333-359.

[8] GRIEWANK, A. 2000. Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, *No. 19 in Frontiers in Appl. Math.*, SIAM, Philadelphia.

[9] GRIEWANK, 2012. On the numerical stability of algorithmic differentiation, *Computing*, Vol. 94, 125-149.

[10] HANKE M., SCHERZER O. 2001. Inverse Problems Light: Numerical Differentiation, *The American Mathematical Monthly*, Vol. 108, No. 6, 512-521.

[11] HIGHAM N. 2002. Accuracy and Stability of Numerical Algorithms, SIAM.

[12] KRONSJO L. 1979. Algorithms. Their complexity and efficiency. John Wiley and Sons.

[13] RAMM A. G. AND SMIRNOVA A. B., 2001. On stable numerical Differentiation, *Mathematics of Computation*, Vol. 70(235), 1131-1153