Trabajo Práctico 1: Espía por error (numérico)

Universidad de Buenos Aires
acultad de Ciencias Exáctas y Naturales
(FCEyN)
Departamento de Computación

González Sergio (gonzalezsergio2003@yahoo.com.ar)
González Emiliano (XJesse_JamesX@hotmail.com)
Ferro Mariano (eltrencitomasverde@gmail.com)

10 de Septiembre, 2007

Resumen

En este trabajo presentamos un grupo de algoritmos para la aproximación de f(x)= eˆ(-x), basados en la serie de Taylor para la función correspondiente, alrededor del cero. Dichos algoritmos fueron implementados bajo un sistema de aritmética finita y arbitraria, con el estándar IEEE 754 como base y con un método de truncamiento simple para mantener la coherencia de la precisión. Como fruto de la utilización de una aritmetica de precisión finita, surgen distintos tipos de error. Distintas formas de implementar los algoritmos conducen a un mayor o menor error, en distintas circunstancias. De las aproximaciones obtenidas mediante los algoritmos implementados, calculamos el error relativo para la función eˆ(-x) evaluada en distintos puntos, precisiones y cantidades de términos y los comparamos, contrastando las hipótesis preliminares con los resultados obtenidos mediante la experimentación, en busca del algoritmo más idoneo para realizar el calculo.

Palábras clave:

- Error relativo.
- Aproximación.
- Aritmetica finita.
- eˆ(-x).

Introducción:

El análisis numérico básicamente se encarga de analizar, describir y crear algoritmos numéricos que permiten resolver problemas matemáticos. Estos algoritmos generalmente nos permiten obtener resultados aproximados, ya que contienen un número finito de pasos. El uso del análisis numérico toma gran importancia con el uso de las computadoras y el poder de cálculo que ellas tienen. Por este medio, es posible resolver problemas más complejos.

Pero el uso de computadoras para hacer cálculos complejos trae un problema consigo, y surge el concepto de error. Este concepto nace debido a que las computadoras trabajan con un rango finito de números, y además cada uno de estos está representado de una forma también finita.

Los errores están divididos en tres tipos: Errores en los datos de entrada, errores de redondeo y errores de truncamiento. Los errores en los datos de entrada no están causados por el algoritmo que resuelve el problema, sino por valores que inician el algoritmo, generalmente estos valores se refieren a mediciones o magnitudes físicas. Los errores de redondeo surgen cuando se utilizan operaciones que tienen una representación numérica finita, esto significa que tienen una precisión limitada con respecto al resultado que devuelven. Y los errores de truncamiento están relacionados con el algoritmo en si, esto quiere decir que dependen de la forma en que se resuelve el problema, en algunos casos el error de truncamiento se puede disminuir modificando o refinando el algoritmo, esto generalmente implica aumentar la cantidad de operaciones a hacer y por lo tanto aumentar el error de redondeo y el tiempo para resolver el problema.

El uso de computadoras en el cálculo numérico no solo conduce a errores de los antes mencionados, sino que, impulsado por alguno de los anteriores, aparece el error por resolver el problema no como se ha formulado, sino a través de alguna aproximación. Este error es causado por reemplazar un infinito(sumatoria o integral por ejemplo) por una cantidad finita de términos La precisión finita que introduce algunos de los errores numéricos, es la que opera bajo el estándar de la IEEE (Institute of Electrical and Electronics Engineers) nro. 754, que normaliza la notación en bits de los números de punto flotante, la norma tiene cuatro grados de precisión: simple $32bits$, simple extendida(43 bits, no se utiliza habitualmente), doble (64) y doble extendida (implementada en 80 bits o más). Todas las normas cuentan con tres campos: signo, mantisa y exponente, los dos últimos varían su longitud según de cual se trate. En el caso de la doble extendida, se asigna un bit al signo (la parte más alta de la cadena), 64 a la mantisa (que se almacena en la parte baja de la cadena en notación sin signo) y 15 al exponente(entre las otras dos, se encuentra desplazado 2ˆ(14)). El rango representado es [-10ˆ(4932), 10ˆ(4932)] aprox.

A su vez, existen varias maneras de cuantificar el error, de forma que su medición se torne algo más tangible y útil, entre ellas, el error relativo y el error absoluto son las que trataremos en este trabajo. Desde el punto de vista absoluto, el error surge de la diferencia que pudiera haber entre la magnitud real que se desea expresar, y la obtenida mediante cálculos computacionales. Por otra parte el error relativo tiene en cuenta no solo las magnitudes, sino el cociente entre la diferencia anterior y el valor absoluto de la magnitud medida. De esta forma se obtiene una real dimensión del error y no tan solo el valor absoluto del mismo. Por estos motivos, una de las tareas del análisis numérico se trata de buscar un algoritmo que lleve a la mejor solución posible de cada problema. De esta forma se pueden definir muchos algoritmos que lleguen a la solución, pero se seleccionará el que mejor aproxime a la solución del problema, es decir, el que mejor utilice las operaciones respecto de sus errores.

En este caso surge la noción de estabilidad numérica. La estabilidad numérica define cuan buena será la solución de nuestro problema usando métodos aproximados. Estos métodos pueden tener un resultado diferente al esperado, ya que tienen diferente estabilidad numérica, esto quiere decir que para ciertos valores de entrada, con sus respectivos errores, el método puede propagar el error por el algoritmo en mayor o menor medida. De esta forma, el algoritmo que mejor aproxime a la solución del problema, será aquel que tenga mejor estabilidad numérica.

Desarrollo:

El problema puntual abordado por este trabajo es obtener una aproximación de la función eˆ(-x), mediante el polinomio de Taylor. El desarrollo de este algoritmo contempla dos variantes:

1. Desarrollar la función f(x) = e^(-x)en serie de Taylor alrededor del 0, y evaluar esta serie en el punto x = a.

2. Desarrollar la función f(x) = e^x en serie de Taylor alrededor del 0, evaluar esta serie en el punto x = a y responder la inversa del valor calculado.

En cualquier caso, se incurre en varios tipos de error, a saber:

1. Error debido a la implementación en aritmética finita

2. Error de truncamiento en cuanto a que el polinomio de Taylor es una serie, por tanto infinita, de términos, que se verá acotada a la cantidad que el usuario considere conveniente.

Es condición para el trabajo realizar la aritmética con precisión arbitraria (no por eso pierde su calidad de finita), determinada por el usuario. Entiéndase por precisión, el número de bits de la mantisa en notación normalizada, esta cantidad esta acotada (en este trabajo) superiormente por 64.

El resultado de analizar las magnitudes y los comportamientos de los errores introducidos por el polinomio de Taylor para el cálculo de la función serán abordados en este mismo trabajo, más adelante.

Detalles de implementación:

Como la condición del trabajo es obtener el valor de la función e^(-x) implementando una aritmética de precisión arbitraria, comenzamos pensando cómo es que realmente íbamos a implementarla. Luego de algunas reflexiones, llegamos a concebir dos formas diferentes. La primera, diseñar e implementar desde cero, una aritmética arbitraria adaptada al problema que se quería resolver. La segunda, utilizar la aritmética fija que nos provee la PC como base y modificarla de alguna manera para que tenga la propiedad de ser arbitraria.

Comenzamos planteando la primera opción, buscando una manera de representar los datos. No hubo divergencia en cuanto a este tema, ya que solo una forma de representación nos vino a la mente, la de utilizar un arreglo de bool del tamaño que el usuario indicara para la cantidad de dígitos en la mantisa, más los del exponente y el signo. Este arreglo contendría ceros y unos, y codificaría cada número a representar, ya sea por ingreso manual del dato o como resultado de alguna función del sistema a implementar.

El formato a utilizar para almacenar los datos dentro del arreglo iba a estar basado en el estándar IEEE numero 754. Es decir, que el arreglo tendría tres partes, una parte para el signo, otra para el exponente y otra para la mantisa. En principio, un problema que se nos planteo era el de saber o determinar cuantos "bits"tendría asignado el exponente, el signo y la mantisa no tendrían ningún inconveniente, ya que tendrían un bit y la cantidad ingresada por el usuario respectivamente. Como nuestra idea era conseguir una mejor precisión a la hora de resolver el problema, decidimos asignarle la misma cantidad de bits que usa el estándar IEEE numero 754 en su versión extendida, ya que consideramos esa cantidad ni muy pobre ni muy excesiva a la hora de hacer los cálculos.

Finalmente, dada la envergadura de la implementación que esta opción requeriría, la consideramos demasiado compleja, por lo que desistimos de utilizarla.

Planteamos la segunda opción, y de inmediato nos percatamos de que era relativamente mas fácil de implementar que la anterior, pero acarreaba el inconveniente de que la cantidad de bits en la mantisa estaba limitada superiormente. Esto nos permitió saber que íbamos a tener una cota con respecto a la precisión que podría requerir el usuario. De todas formas, a pesar del inconveniente mencionado, nos pareció mas seguro con respecto a los cálculos, trabajar con una representación que ya estaba asentada y por lo tanto funcionando.

Teniendo en cuenta estas consideraciones sobre la complejidad de implementación y confiabilidad de la aritmética nos decidimos finalmente por la segunda opción.

Utilizando el tipo de dato long double (que tiene un formato de 80 bits significativos, con las características del IEEE 754 doble extendido mencionados más arriba) de C++ como base,

creamos una clase que cuenta con dos atributos: el valor real, almacenado en un long double, y un entero que nos indica la precisión con la que se van a enmascarar los datos. El nombre de la clase es DLFloat (Defined Longitude Float).
La clase funciona de la siguiente manera:

1. Se crea un DLFloat con una precisión y un valor decididos por el usuario, que debe ser mayor a 0 y menor que 64. El valor, inicialmente long double, se trunca para acordarse con la precisión. Los valores por defecto, en el caso que el usuario no ingrese un parametro, son: cero para el valor (formato IEEE 754 doble extendido) y 63 de precisión. Nuestra clase define la precisión como los bits que comprenden la mantisa, incluido el bit explicito de la parte entera. Esto quiere decir que ni el exponente ni el signo pueden ser de longitud definida por el usuario, y la precisión no los afecta.

2. Las operaciones disponibles son: suma, resta, multiplicación, división y potenciación. En el caso particular de la división se incluyó por comodidad la división contra un long double, para todo el resto de las operaciones los operandos son siempre DLFloat.

3. Toda operación aritmetica se realiza con precisión doble extendida y finaliza con un llamado a la función "truncar", ésta se encarga de acotar la precisión del resultado obtenido mediante la manipulación en 80 bits de los operandos a la requerida por el usuario. El método que sigue esta función consiste en reservar los bits más significativos hasta completar la precisión requerida, y colocar ceros en los bits que sobrepasan la precision pedida. Tras efectuar el truncamiento, tanto el signo como el exponente conservan su valor.

En cuanto a la función 'truncar', cabe aclarar que durante gran parte del desarrollo se mantuvo una función llamada 'redondear', que seguía el mismo método que la anterior, pero sumaba uno en el bit más significativo de los que serían descartados. Esto acarreó varios problemas, en primer lugar, por la forma de la implementación la suma no se propagaba a través de los bytes. Cuando solucionamos ese inconveniente surgió otro, si la mantisa del número consistía solo en unos, el redondeo se propagaba hasta el exponente (comportamiento matemáticamente correcto) y la mantisa quedaba en cero. Este formato es indefinido para los compiladores utilizados, y si bien en el caso de las potencias inversas de e, obtenerlo resulta imposible, se opto por el truncamiento en pos de una implementación más firme y reutilizable.

Habiéndonos decidido en cuanto a la representación de los valores que debíamos manejar, pasamos a trasladar el algoritmo de la serie de Taylor a la PC en las condiciones antes mencionadas.

Esta implementación no revistió mayores dificultades, razón por la cual decidimos agregar dos versiones más a las ya requeridas, teniendo en total cuatro versiones del algoritmo que implementa el polinomio de Taylor. Las dos primeras son las mencionadas en el enunciado y reproducen fielmente la serie de Taylor, definida matemáticamente, hasta una cantidad de términos decidida por el usuario. Las otras dos, alteran la sucesión de los términos de manera que la suma de los mismos se efectúa de manera inversa, yendo desde los términos más pequeños hacia los más grandes.

Establecidas las cuatro formas de aproximar e^(-x), las hipótesis preliminares acerca del error situaban a la serie de Taylor de e^(-x) calculada de menor a mayor como la de mejor rendimiento. En segundo lugar, pero sin unanimidad, se encontraban la serie de Taylor de e^(x) invertida de mayor a menor, y la variante e^(x) de menor a mayor. En último lugar, todos estuvimos de acuerdo en colocar a la serie para e^(x) invertida y calculada de mayor a menor.

Los criterios para establecer estas hipótesis son básicamente tres: si se suman primero los números más pequeños, estos se vuelven los suficientemente significativos como para poder modificar a un número más grande en una suma. De otro modo puede suceder que sumar un número grande a uno pequeño, nos dé por resultado el grande sin modificación.

El segundo criterio es muy sencillo, consideramos que la serie de Taylor para e^(-x), aproxima mejor a ese número que la de e^(x) invertida, ya que la primera es la serie 'exacta', mientras que la segunda no lo es, y para transformarla se requiere una operación extra, que incrementa el error.

El tercer criterio es que se esperaba que las restas de la serie 'exacta' de eˆ(-x) compensaran el valor y se pierdan menos datos por culpa del truncamiento.

Resultados:

Se hicieron diversas pruebas, en las cuales se probo la calidad del resultado a calcular. Estas consisten en: Mostrar el error relativo para cada método en función de la cantidad de iteraciones de la serie de Taylor; Mostrar el error relativo en cada método en función de la precisión requerida por el usuario; Y comparar el error relativo de los métodos en considerando el valor en el que se instanciaría la función
Orden vs Precision:

En los dos primero gráficos se muestran lo errores relativos calculando eˆ(-x) y 1/eˆ(x), los dos en función de la precision ingresada, para un punto de evaluación x=1 y para un orden de la serie de Taylor de 50 términos.

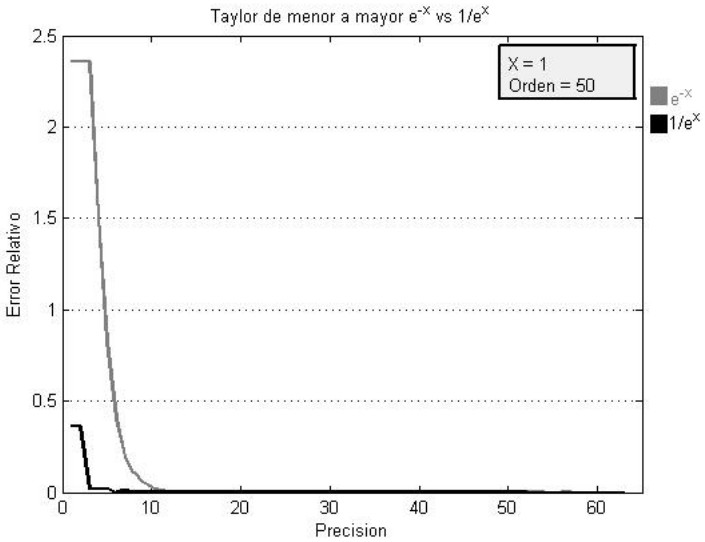

Grafico1

En el gráfico 1 se ve reflejado el error relativo usando la versión de la serie de Taylor que calcula desde los términos más chicos en valor a los más grandes. En este caso se ve una clara diferencia entre eˆ(-x) y 1/eˆx al comienzo del gráfico, luego a medida que aumenta la precisión, el error relativo de ambos métodos se va acercando a cero rápidamente. De todas formas, según el gráfico la función que menos error relativo tiene a medida que aumenta la precisión es 1/eˆx.

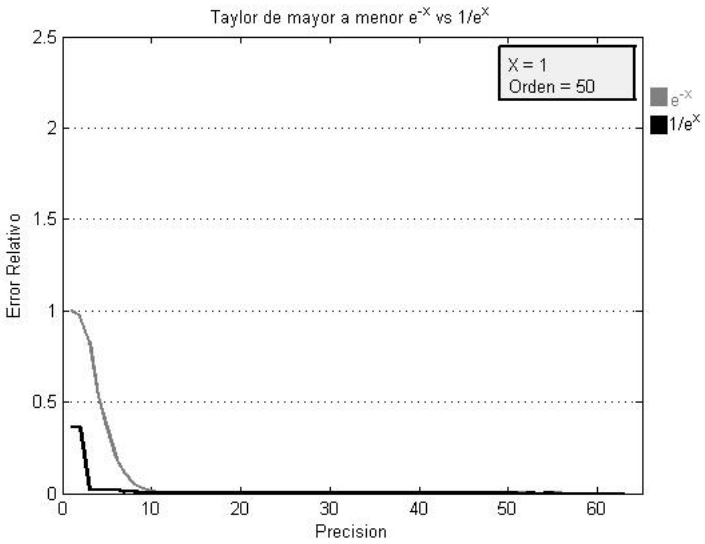

Grafico2

En este otro gráfico se refleja la misma situación antes comentada, con la diferencia que se usa la variante de Taylor que comienza por los términos más grandes en tamaño hacia los más chicos. En este caso se percibe al comienzo del gráfico, que ambos métodos tienen un error relativo mucho menor con respecto a la otra serie. Al igual que en el gráfico anterior, el método que menor error relativo tiene es 1/eˆx.

En los dos siguientes gráficos se mostrará el error relativo en función del orden de la serie de Taylor(es decir la cantidad de términos a desarrollar).

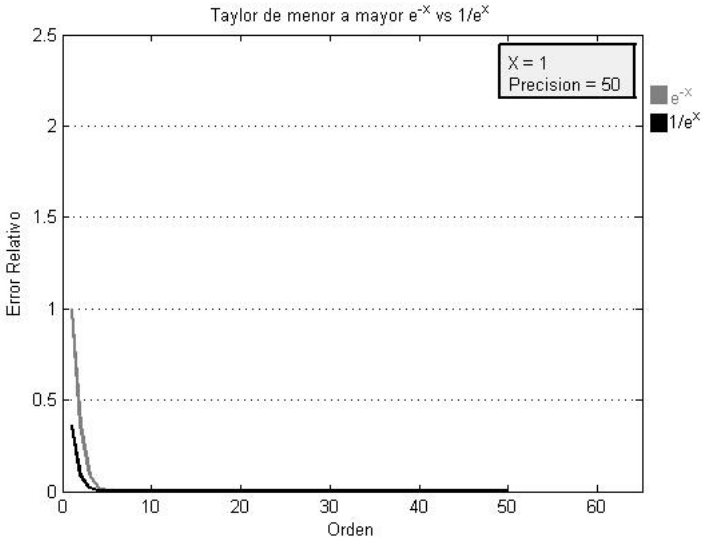

Grafico3

En el grafico 3 se ve claramente como a partir del quinto orden, el error relativo se hace muy chico, para ambas series de Taylor. En este juego de gráficos la serie de menor error relativo es nuevamente 1/eˆx.

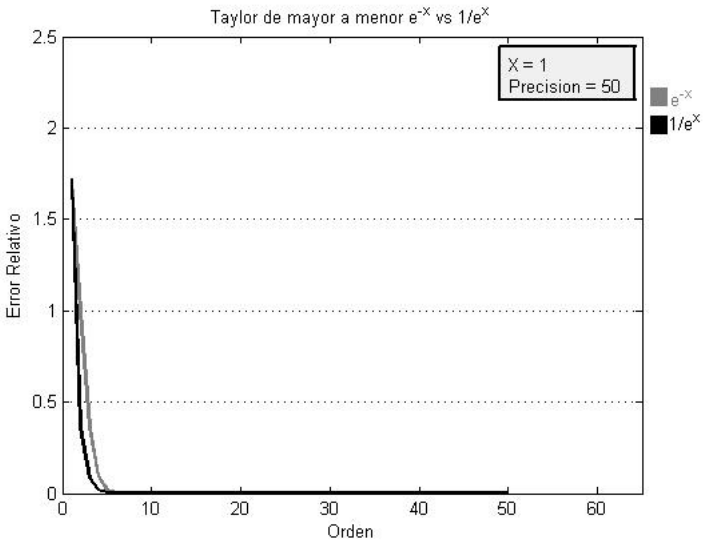

Grafico4

Este caso es claramente peor que el anterior ya que para una cantidad pequeña de términos de la serie, el error que se comente es mucho mayor. En ambos gráficos se nota un mejor desempeño para las series calculadas de menor a mayor, esto se debe, creemos, al criterio expresado anteriormente, según el cual este método de sumar los términos, tiende a brindarle significatividad a cada uno de los operandos.

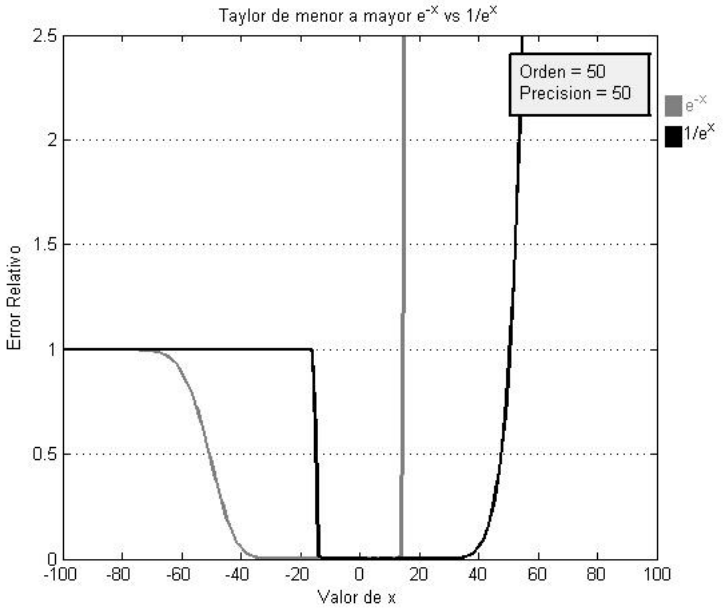

Grafico5

En los últimos 2 gráficos, en donde varia el punto de evaluación y permanecen fijos el orden de la serie y la precisión, se ve como para ciertos valores un método aproxima mejor que otro, esto quiere decir que el error relativo para ciertos valores es mucho menor en un método que en otro. Por ejemplo, para los valores positivos se observa que ambos tienen un error relativo pequeño hasta el valor 15 aproximadamente, luego de ese valor los 2 métodos se comportan de manera similar cuando el valor a evaluar es muy grande, salvo entre los valores 15 y 35 aproximadamente donde se observa claramente que 1/eˆx tiene una mejor aproximación con respecto a eˆ(-x).

Para los valores negativos, ocurre algo diferente. Hasta el valor -15 aproximadamente ambos

métodos tiene un error pequeño. Pero entre los valores -15 y -35 vemos que pasa lo inverso a lo que pasaba en los valores positivos, en este caso el método para e^(-x) mantiene un error relativo pequeño respecto del de 1/e^x, por lo tanto se obtiene un mejor resultado. Para los valores menores que -35, se observa que ambos métodos convergen en un error igual a 1, lo que indica que no se obtiene una buena aproximación para esos valores.

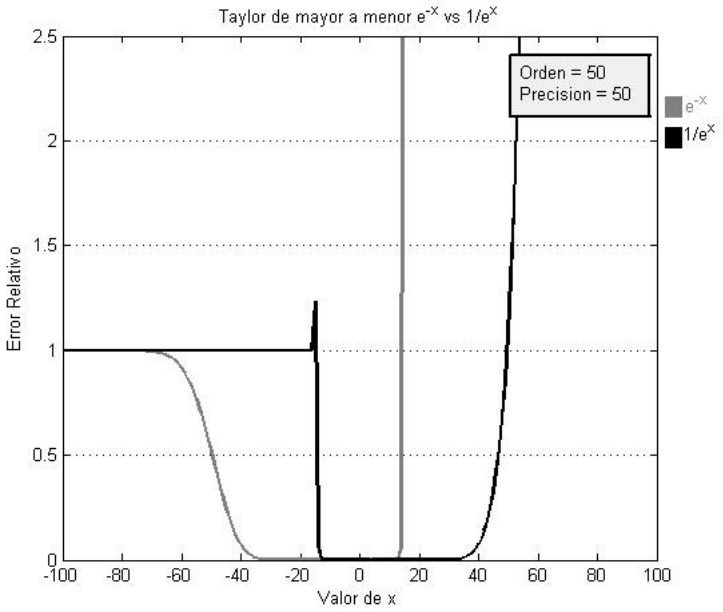

Grafico6

Este último gráfico es muy similar al anterior, con la diferencia que con la versión de la serie de Taylor utilizada en este gráfico, se observa una leve perdida en la precisión, además del pico que se observa entre los valores -15 y -20 aproximadamente para 1/e^x. No es despreciable notar en los gráficos 5 y 6 que para valores altos de x el error es no acotado, tiende a infinito. Este fenómeno es una consecuencia de la utilización de la aritmética de precisión finita. Con esta forma de representación para los valores, aquellos x por sobre un valor determinado se salen del rango aceptado, haciendo que la PC busque representaciones alternativas y que el error relativo sea muy grande.

Como conclusión general resultante de todos los gráficos, los métodos que emplean la inversa de e^x son con mucho, más precisos que los que se basan directamente en e^(-x). Esto da por tierra con una de nuestras hipótesis preliminares, sin embargo, tiene un asidero teórico. La segunda variante es una serie de sumas y restas, esta última operación esta mal condicionada, lo que quiere decir que para valores similares incurre en grandes errores. Justamente la serie de Taylor guarda cierta similaridad entre sus valors contiguos, con lo que podríamos considerarla un caso patologico para el mal condicionamiento de la resta, con lo que esta variante pierde confiabilidad.

Discusion:

Como conclusión general resultante de todos los gráficos, los métodos que emplean la inversa de e^x para x positivo son más precisos que los que se basan directamente en e^(-x). Esto da por tierra con una de nuestras hipótesis preliminares, sin embargo, tiene un asidero teórico. La segunda variante es una serie de sumas y restas, esta última operación esta mal condicionada, lo que quiere decir que para valores similares incurre en grandes errores. Justamente la serie de Taylor guarda cierta similaridad entre sus valores contiguos, con lo que podríamos considerarla un caso patologico para el mal condicionamiento de la resta, con lo que esta variante pierde confiabilidad.

Sin embargo, los métodos que emplean e^(-x) para x negativo son más precisos que los que se basan directamente en la inversa de e^x. Esto se debe a que la serie de Taylor de e^(-x) tiene sus restas en los términos impares, lo que quiere decir que con x negativo, el termino se hara positivo,

y los términos pares serán también positivos pero gracias a la potencia par de x. Ademas la inversa de e^x debe hacer una o operación de mas (la inversa) lo cual suma error relativo, para x positivo perdia importancia gracias al mal condicionamiento de la resta de e^(-x). Entonces la serie e^(-x) para x negativo cede sus términos negativos a la la inversa de e^x, y sumando se evita el mal condicionamiento de la resta que sufría para x positivo.

Un dato interesante que se obtiene de los gráficos del 1 al 4 es que se observa que con una buena precisión, no se requiere desarrollar demasiados términos en la serie de Taylor para obtener un error relativo pequeño(en los gráficos 3 y 4 se puede ver como con orden mayor a 5 el error relativo tiende a cero), en cambio, cuando se usa una precisión baja (por baja se entiende menor a 10), el incrementar el orden de la serie, no implica disminuir demasiado el margen del error relativo. Resulta evidente que si se mantiene fija la cantidad de términos y se incrementa la precisión, se esta actuando unicamente para acotar el error, mientras que mantener quieta la precisión y alimentar la serie con más términos, incluye además de una ganancia en exactitud, una perdida, ya que un orden mayor involucra más operaciones que aumentan el error.

Este resultado ademas de teorico es bastante intuitivo, por una simple razón: cuando se usa una precisión alta y se calculan varios términos, estos actúan generando una ganancia en la precisión al margen de que se producen errores en las operaciones, sin embargo en el otro caso, al haber poca precisión la incidencia del error es mucho mayor.

Conclusiones Generales:

Comparando las hipótesis preliminares con respecto a los resultados obtenidos, podemos concluir que, las mismas no se ven respaldadas en su totalidad. Los fundamentos teóricos que rebaten las suposiciones formuladas sobre el comportamiento del algoritmo que implementa e^(-x) contra 1/e^(x), ya se han comentado en secciones anteriores. Como conclusión de este hecho, podemos decir que para obtener la mejor aproximación posible se debe utilizar una conjunción de ambos métodos, empleando e^(-x) cuando se trabaja con x negativo, y 1/e^(x) en caso contrario. Cada uno tiene un rendimiento notoriamente superior a su contraparte en los casos en los que se desempeña mejor. Además, siempre basándonos en los gráficos 5 y 6 en los algoritmos que suman desde los términos más pequeños de la serie hacia los más grandes parece haber una magnitud de error menor. Para lograr una aún mejor aproximación, recomendamos incrementar la precisión y mantener la cantidad de términos de la serie de Taylor en la menor posible, por razones también ya comentadas en secciones anteriores, ya que existe un limite donde agregar terminos no mejora la aproximacion.

Otra observación con respecto a las hipótesis planteadas al principio del artículo, es que la variante del polinomio de Taylor que trabaja con los términos más pequeños hacia los más grandes, no resultó tan efectiva como la imaginábamos. Observando los gráficos del uno al cuatro, primero se evidencia que en los primeros dos, Taylor de menor a mayor tiene un menor desempeño con respecto a su contraparte original, si bien la diferencia es casi nula para el método 1/e^x, calculando e^(-x) se observa mejor.

No se encontraron fundamentos teóricos que respalden lo observado en los gráficos del 1 al 4, en cuanto a que no siempre es mejor el algoritmo que calcula la serie desde los menores hacia los mayores. Sin embargo no se descarta ninguna hipótesis, entre las cuales se encuentra un posible error de implementación, tanto en el algoritmo como en el experimento, o un caso de prueba en particular poco favorable.

El método general para la manipulación aritmética en precisión finita fue el truncamiento, sin embargo, extender esto al redondeo mejoraría notoriamente la aproximación, reduciendo el error absoluto prácticamente a la mitad, con la consecuente reducción del error relativo, aunque en menor magnitud. Ya se ha comentado acerca de este tema en los Detalles de Implementación.

Otra posible estrategia para mejorar la aproximación es implementar un sistema de aritmética arbitraria sin la cota superior de 64 bits, mediante un arreglo arbitrariamente grande de valores bool, que emule un numero normalizado. Esta opción de implementación se descarto por los tiempos a los que estaba sujeto el proyecto. Por esta misma razón, no se realizaron algunos experimentos más que podrían haber arrojado alguna luz sobre el comportamiento errático de los

algoritmos que implementan la serie de menor a mayor.

Apendice A:

Laboratorio de Métodos Numéricos - Segundo cuatrimestre 2007
Trabajo Práctico Número 1: Espía por error (numérico)

El objetivo del trabajo práctico es realizar un análisis empírico del comportamiento numérico de la serie de Taylor para calcular el valor de e^{-a}, con $a \in \mathbf{R}$. Consideremos los siguientes métodos para calcular el valor de esta expresión:

1. Desarrollar la función $f(x) = e^{-x}$ en serie de Taylor alrededor del 0, y evaluar esta serie en el punto $x = a$.

2. Desarrollar la función $f(x) = e^{x}$ en serie de Taylor alrededor del 0, evaluar esta serie en el punto $x = a$ y responder la inversa del valor calculado.

Implementar ambos métodos con aritmética binaria de punto flotante con t dígitos de precisión y comparar los errores relativos de los resultados obtenidos en ambos casos. ¿Por qué se observan estos resultados? ¿Puede dar una explicación intuitiva de los errores obtenidos? Para profundizar en el análisis, realizar los siguientes experimentos numéricos:

1. Reportar el error relativo de cada método en función de la cantidad de iteraciones de la serie de Taylor. ¿Se puede afirmar que algún método tiene una convergencia superior?

2. Reportar el error relativo de cada método en función de la cantidad t de dígitos binarios considerados para implementar los cálculos numéricos. Discutir estos resultados, explicando las razones que motivan este comportamiento y agregando todos los experimentos numéricos que sean necesarios.

3. Comparar el error relativo de ambos métodos en función del valor de a, dentro de un intervalo razonable.

Se deben presentar los resultados de estas pruebas en un formato conveniente para su análisis. Sobre la base de los resultados obtenidos, ¿se pueden extraer conclusiones sobre la conveniencia de utilizar uno u otro método?

El informe debe contener una descripción detallada de las distintas alternativas que el grupo haya considerado para la implementación de la aritmética de punto flotante de t dígitos binarios de precisión, junto con una discusión de estas alternativas que justifique la opción implementada. Por otra parte, se debe incluir en la sección correspondiente el código que implementa esta aritmética, junto con todos los comentarios y decisiones relevantes acerca de esta implementación.

Fecha de entrega: Lunes 10 de Septiembre

Apendice B:

1. **DLFloat.h**

```
#ifndef _DLFloat_H
#define _DLFloat_H

#include <stdio.h>
#include <iostream>

using namespace std;
```

```
class DLFloat{
        friend ostream& operator << (ostream &o, const DLFloat &a);
        friend void truncar(DLFloat *c);

    public:

        DLFloat(int = 63, long double = 0); //precision predefinida: 63 bits
                                            //valor predefinido: 0

        int  precision (void);
        long double valor    (void);
        DLFloat  operator+ (const DLFloat &a);
        void      operator++ (int);
        DLFloat  operator-(const DLFloat &a);
        DLFloat  operator* (const DLFloat &a);
        DLFloat  operator/ (const DLFloat &a);
        DLFloat  operator/ (long double a);
        DLFloat  operator^ (int a);
        void  operator= (const DLFloat &a);
        void  operator= (long double a);

        ~DLFloat(){};
    private:
        int prec;
        long double numero;
};

#endif /*_DLFloat_H*/
```

2. DLFloat.cpp

```
#include "DLFloat.h"

#define MAX(a, b) (a > b) ? a : b

DLFloat :: DLFloat(int precision, long double valor){
    assert((precision < 64) && (precision > 0));
    prec = precision;
    numero = valor;
    truncar(this);
}

int DLFloat :: precision(void){
    return prec;
}

long double DLFloat :: valor(void){
    return numero;
}

DLFloat DLFloat :: operator+(const DLFloat &a){
   DLFloat res(MAX(a.prec, prec));

   res.numero = numero + a.numero;
   truncar(&res);

   return res;
}
```

```
void DLFloat :: operator++(int a){
    numero++;
}

DLFloat DLFloat :: operator-(const DLFloat &a){
   DLFloat res(MAX(a.prec, prec));

   res.numero = numero - a.numero;
   truncar(&res);

   return res;
}

DLFloat DLFloat :: operator*(const DLFloat &a){
   DLFloat res(MAX(a.prec, prec));

   res.numero = numero * a.numero;
   truncar(&res);

   return res;
}

DLFloat DLFloat :: operator/(const DLFloat &a){
   DLFloat res(MAX(a.prec, prec));

   res.numero = numero / a.numero;
   truncar(&res);

   return res;
}

DLFloat DLFloat :: operator/(long double a){
   DLFloat res(prec);

   res.numero = numero / a;
   truncar(&res);

   return res;
}

DLFloat DLFloat :: operator^(int a){
   DLFloat res(prec);
   bool neg = 0;

   if ( a < 0 ){
      a = -a;
      neg = 1;
   }

   res = numero;
   a--;

   while (a > 0){
      res.numero *= numero;
      truncar(&res);
      a--;
   }
```

```
   if (neg){
      res.numero = 1/(res.numero);
      truncar(&res);
   }

   return res;
}

void DLFloat :: operator=(const DLFloat &a){
    numero = a.numero;
    truncar(this);
}

void DLFloat :: operator=(long double a){
    numero = a;
    truncar(this);
}

void truncar(DLFloat* c)
{
   //La variable bytes representara la cantidad de bytes que tenemos de mantisa
   int bytes = 8;    //"ignoramos" los 2 bytes de exponente + signo al truncar
                     //que estan en la posicion de memoria mas alta

   bytes -= (int)((c->prec)/8);
   //esto representa la cantidad de bytes enteros que voy a eliminar
   //desde el menos significativo

   unsigned char* ch = ((unsigned char*)&(c->numero) + bytes - 1);


   /* PARTE DE REDONDEO CANCELADA */
   /* *ch += ( 1 << (7 - (c->prec % 8)) ); */
   /* con esto le sumamos 1 al primer bit que voy a truncar */

   *ch &= (255 << (8 - (c->prec % 8)) ); //255 = 1111 1111
   //ahora borramos los bytes menos significativos
   memset(&(c->numero), 0, bytes - 1);
}

ostream& operator<<(ostream &os, const DLFloat &a){
   os << a.numero;
   return os;
}
```

3. main.cpp

```
#include <iostream>
#include <fstream>
#include <math.h>
#include <iomanip.h>
#include "DLFloat.h"

using namespace std;

DLFloat factorial(unsigned int n, unsigned int p)
{
   DLFloat res(p, 1);
```

```
   DLFloat i(p, 2);
   //i lo hice DLFloat por la linea: res = res*i; que viene luego

   if (n == 0)
      return res;

   while (i.valor() <= n){
      res = res*i;
      i++;
   }
   return res;
}

////////////////////////////////////////////////////////////////////////////
//Aproximacion de 1/(e^x) con taylor, sumando desde el mayor termino al menor
////////////////////////////////////////////////////////////////////////////
DLFloat taylorMenorAMayorInv(long double valor, unsigned int n, unsigned int precision){
   DLFloat res(precision, 1); //lo inicializo en 1 para no calcular el 1er termino de la serie
   DLFloat x(precision);

   x = valor;

    while (n > 0){
      res = res + (x^n)/factorial(n,precision);
      n--;
   }

   return res^(-1);
}

////////////////////////////////////////////////////////////////////////////
//Aproximacion de 1/(e^x) con taylor, sumando desde el menor termino al mayor
////////////////////////////////////////////////////////////////////////////
DLFloat taylorMayorAMenorInv(long double valor, unsigned int n, unsigned int precision){
   DLFloat res(precision, 1); //lo inicializo en 1 para no calcular el 1er termino de la serie
   DLFloat x(precision);

   x = valor;

   unsigned int i = 1;

   while (i < n){
      res = res + (x^i)/factorial(i, precision);
      i++;
   }

   return res^(-1);
}


////////////////////////////////////////////////////////////////////////////
//Aproximacion de e^(-x) con taylor, sumando desde el mayor termino al menor
////////////////////////////////////////////////////////////////////////////
DLFloat taylorMenorAMayor(long double valor, unsigned int n, unsigned int precision){
   DLFloat res(precision, 1); //lo inicializo en 1 para no calcular el 1er termino de la serie
   DLFloat x(precision);
   DLFloat neg(1, 1);
```

```
   x = valor;

   while (n > 0){
      neg = pow(-1,n%2);
      res = res + neg*((x^n)/factorial(n,precision));
      n--;
   }

   return res;
}

////////////////////////////////////////////////////////////////////////////
//Aproximacion de e^(-x) con taylor, sumando desde el menor termino al mayor
////////////////////////////////////////////////////////////////////////////
DLFloat taylorMayorAMenor(long double valor, unsigned int n, unsigned int precision){
   DLFloat res(precision, 1); //lo inicializo en 1 para no calcular el 1er termino de la serie
   DLFloat x(precision);
   DLFloat neg(1, 1);

   x = valor;

   unsigned int i = 1;

   while (i < n){
      neg = pow(-1,i%2);
      res = res + neg*((x^i)/factorial(i, precision));
      i++;
  }

    return res;
}

int main()
{
unsigned int p, orden;
long double valor, e = 2.71828182845904523536028747135266249775724709369995957496696762772 4;
char hacerPrueba = 's';

while(hacerPrueba == 's')
   {
       cout << "Aproximacion de e^(-x) con presicion y orden variable" << endl;
       cout << "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n\n" << endl;

       cout << "Ingresar precision con el cual quiere trabajar el punto flotante: ";
       cin >> p;

       cout << "\n\n";

       cout << "Ingresar orden con el cual quiere aproximar a e^(-x) (1~170): ";
       cin >> orden;

       cout << "\n\n";

       cout << "Ingresar x: ";
       cin >> valor;

       cout << "\n\n";
```

```
        DLFloat resultado1(p);
        DLFloat resultado2(p);

        DLFloat resultado1Inv(p);
        DLFloat resultado2Inv(p);

        resultado1 = taylorMenorAMayorInv(valor, orden, p);
        resultado2 = taylorMayorAMenorInv(valor, orden, p);

        resultado1Inv = taylorMenorAMayor(valor, orden, p);
        resultado2Inv = taylorMayorAMenor(valor, orden, p);

        cout.precision(40);
        cout << "Resultado verdadero de e^(-x): " << pow(e,-valor) << endl;
        cout << "Resultado con taylor de mayor a menor 1/(e^x): " << resultado1 << endl;
        cout << "Resultado con taylor de menor a mayor 1/(e^x): " << resultado2 << endl;
        cout << "Resultado con taylor de mayor a menor e^(-x) : " << resultado1Inv << endl;
        cout << "Resultado con taylor de menor a mayor e^(-x) : " << resultado2Inv;
        cout << endl << endl;

        cout << "Error Absoluto mayor a menor 1/(e^x): ";
        cout << (fabs(pow(e,-valor) - resultado1.valor())) << endl;

        cout << "Error Relativo mayor a menor 1/(e^x): ";
        cout << (fabs(pow(e,-valor) - resultado1.valor()) / pow(e,-valor));
        cout << endl << endl;

        cout << "Error Absoluto menor a mayor 1/(e^x): ";
        cout << (fabs(pow(e,-valor) - resultado2.valor())) << endl;

        cout << "Error Relativo menor a mayor 1/(e^x): ";
        cout << (fabs(pow(e,-valor) - resultado2.valor()) / pow(e,-valor));
        cout << endl << endl << endl;

        cout << "Error Absoluto mayor a menor e^(-x): ";
        cout << (fabs(pow(e,-valor) - resultado1Inv.valor())) << endl;

        cout << "Error Relativo mayor a menor e^(-x): ";
        cout << (fabs(pow(e,-valor) - resultado1Inv.valor()) / pow(e,-valor));
        cout << endl << endl;

        cout << "Error Absoluto menor a mayor e^(-x): ";
        cout << (fabs(pow(e,-valor) - resultado2Inv.valor())) << endl;

        cout << "Error Relativo menor a mayor e^(-x): ";
        cout << (fabs(pow(e,-valor) - resultado2Inv.valor()) / pow(e,-valor));
        cout << endl << endl << endl;

        cout << endl << endl;
        cout << "Desea realizar otra prueba? (s/n)" << endl;
        cin >> hacerPrueba;
    }

return EXIT_SUCCESS;
}
```

Referencias

[1] http://www.cplusplus.com/ . Referencia y consulta para C++

[2] http://www.oss-watch.ac.uk/resources/googlecode.xml . Como crear un proyecto usando el Repositorio de google code.

[3] http://svnbook.red-bean.com/ . Manual de SVN.

[4] pcmap.unizar.es/ pilar/latex.pdf . Introduccion a latex.

[5] http://es.wikipedia.org/wiki/Análisis_numérico . descripción del Analisis numerico, errores relativos y metodos de redondeo.

[6] http://babbage.cs.qc.edu/courses/cs341/IEEE-754references.html . Referencia del estandar IEEE 754.

[7] webs.uvigo.es/mat.avanzadas/PracME_1.pdf . Introducción a MatLab.