UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Arqueología informática: Ada Byron y el primer programa publicado de la historia

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autora: Sandra Adobes Soriano
Tutor: Xavier Molero Prieto

Curso 2015–2016

Resumen

Este proyecto plantea el estudio de un trabajo de ingeniería, cuyo objetivo es analizar y entender las aportaciones que realizó Ada Byron en el campo de la informática. El presente trabajo formará parte del proyecto de divulgación patrimonial del Museo de Informática de la Universidad Politécnica de Valencia con el fin de dar a conocer que, para llegar al nivel de desarrollo de las computadoras actuales, primero han tenido que existir genios como Charles Babbage o Ada Byron.

El desarrollo de este trabajo comprende diferentes fases, desde el estudio de la vida de Ada Byron hasta el análisis de su programa para la Máquina Analítica que calcula los números de Bernoulli y que fue publicado en sus famosas notas. De esta manera, observaremos como Ada Byron, a pesar de haber vivido en Londres victoriana del siglo XIX, tuvo unas ideas muy avanzadas en el campo de la informática, un área que tardaría todavía un siglo a hacer acto de presencia de la mano de la electrónica.

Palabras clave: Ada Byron, Máquina Diferencial, Máquina Analítica, Números de Bernoulli, Museo de Informática

Resum

Aquest projecte planteja l'estudi d'un treball d'enginyeria , l'objectiu del qual és analitzar i entendre les aportacions que va dur a terme Ada Byron en el camp de la informàtica. El present treball formarà part del projecte de divulgació patrimonial del Museu d'Informàtica de la Universitat Politècnica de València amb la finalitat de donar a conéixer que, per a arribar al nivell de desenvolupament de les computadores actuals, primer han hagut d'existir genis com ara Charles Babbage o Ada Byron.

El desenvolupament d'aquest treball comprén diferents fases, des de l'estudi de la vida d'Ada Byron fins a l'anàlisi del seu programa per a la Màquina Analítica que calcula els nombres de Bernoulli i que va ser publicat en les seues famoses Notes. D'aquesta manera, observarem com Ada Byron, malgrat haver viscut en la Londres victoriana del segle XIX, va tindre unes idees molt avançades en el camp de la informàtica, una àrea que tardaria encara un segle a fer acte de presència de la mà de l'electrònica.

Paraules clau: Ada Byron, Màquina Diferencial, Màquina Analítica, Números de Bernoulli, Museu d'Informàtica

Abstract

This project involves the study of engineering work, whose goal is analyze and understand the contributions made Ada Byron in the field of computing. This work will be part of the proposed asset disclosure Museum Informatics, Polytechnic University of Valencia in order to make known that, to reach the level of development of today's computers they first have had to be geniuses like Charles Babbage or Ada Byron.

The development of this work involves different phases, from the study of the life of Ada Byron to the analysis of its program for the Analytical Engine calculating Bernoulli Numbers and which was published in his famous notes. Thus, we see as Ada Byron, despite having lived in Victorian London of the nineteenth century, had very advanced ideas in the field of information technology, an area that would still take a century to make an appearance of the hand of electronics.

Key words: Ada Byron, Differential Engine, Analitical Engine, Bernoulli Numbers, Computer Museum

Índice general

Índice general v
Índice de figuras vii
Índice de tablas viii

Agradecimientos ix
1 Motivación y objetivos del trabajo 1
1.1 Motivación . . . 1
1.2 Objetivos . . . 2
1.3 Contexto histórico . . . 2
1.4 Estructura de la memoria . . . 5
1.5 Notas sobre la bibliografía . . . 5
2 Los protagonistas 7
2.1 Charles Babbage . . . 7
2.2 La Máquina Diferencial . . . 10
2.3 La Máquina Analítica . . . 12
2.4 Ada Byron . . . 15
3 Análisis de las Notas de Ada Byron 23
3.1 Nota A . . . 24
3.2 Nota B . . . 26
3.3 Nota C . . . 28
3.4 Nota D . . . 29
3.5 Nota E . . . 30
3.6 Nota F . . . 34
3.7 Nota G . . . 35
4 Cálculo de los números de Bernoulli 41
4.1 Definición . . . 42
4.2 Explicación Ada Byron . . . 42
4.3 Explicación de la autora del trabajo . . . 46
5 Conclusiones 65
5.1 Consideraciones finales . . . 65
5.2 Contratiempos y problemas sufridos . . . 66
5.2.1 Librería BigInteger . . . 66
5.2.2 Clase Comparable en C++ . . . 67
5.2.3 Números de Bernoulli . . . 67
5.2.4 Clase BigDecimal . . . 67
5.2.5 Función que calcula el Máximo Común Divisor . . . 68
5.2.6 BigRational . . . 69
Bibliografía 71

Apéndice
A Configuración del sistema de programación 73

A.1 Fase de inicialización: Visual Studio 73
A.2 Códigos 80

Índice de figuras

2.1 Charles Babbage. . . . 7
2.2 Engranajes de la Máquina Diferencial. . . . 11
2.3 Máquina Diferencial. . . . 12
2.4 Máquina Analítica. . . . 14
2.5 Mecanismo de funcionamiento de la Máquina Analítica. . . . 14
2.6 Padres de Ada 16
2.7 Ada Byron de niña. . . . 16
2.8 Ada Byron. . . . 18

3.1 Notación. . . . 26
3.2 Cantidades. . . . 27
3.3 Cálculo. . . . 27
3.4 Series de sustituciones. . . . 30
3.5 Almacenamiento diferentes variables. . . . 31
3.6 Sustituciones en la Fórmula (3.2). . . . 32
3.7 Introducción de valores. . . . 36
3.8 Cálculo. . . . 36
3.9 Números de Bernoulli. . . . 38
3.10 Aplicación de la notación. . . . 39

4.1 Jakob Bernoulli. . . . 41
4.2 Diagrama Números de Bernoulli. . . . 43
4.3 Primer bloque del diagrama. . . . 44
4.4 Segundo bloque del diagrama. . . . 44
4.5 Tercer bloque del diagrama. . . . 45

5.1 Error. . . . 69

A.1 Aplicación de consola. . . . 73
A.2 Inicio. . . . 74
A.3 Parámetros de entrada. . . . 74
A.4 Creación de ficheros 75
A.5 Añadir nuevo proyecto 75
A.6 Nuevo proyecto. . . . 76
A.7 Añadir libreria. . . . 76
A.8 Nuevo proyecto creado. . . . 77
A.9 Añadir librerías adicionales. . . . 77
A.10 Librerías adicionales. . . . 78
A.11 Compilación. . . . 78
A.12 Librerías. . . . 79
A.13 Compilación correcta. . . . 79
A.14 Ejecución de programa. . . . 80
A.15 Versión definitiva. . . . 80

Índice de tablas

3.1 Series de operaciones a resolver. 31
3.2 Cuarta serie de operaciones. 32
3.3 Última operación a realizar. 32
3.4 Ecuaciones a resolver. 34

4.1 Listado Números de Bernoulli. 42

Agradecimientos

Este trabajo de Final de Grado realizado en la Universitat Politècnica de València es un esfuerzo en el cual, directa o indirectamente, han participado distintas personas, ya sea opinando, escuchando, dando ánimo o acompañándome en los momentos de crisis. Este trabajo me ha permitido aprovechar la capacidad y experiencia de muchas personas que deseo agradecer en este apartado.

En primer lugar a mi tutor de Trabajo de Final de Grado, Dr. Xavier Molero Prieto, mi más sincero agradecimiento por haberme confiado este trabajo, por su paciencia, por su valioso tutelaje y apoyo para seguir hasta la finalización de el trabajo pese a mis problemas sufridos durante el presente curso académico.

A todos mis compañeros y amigos, en especial a Ramón y Álvaro, que sólo puedo dedicaros buenas palabras, ya que si no hubiera sido por esas horas de laboratorio, horas de trabajo y cervezas en la Vella no estaría donde estoy.

Todo esto nunca hubiera sido posible sin el apoyo y el cariño ilimitado de mis padres, que de esta forma incondicional, entendieron mis ausencias y mis ataques de crisis. Que a pesar de la distancia siempre han estado a mi lado.

A ti abuela, por todo. Las palabras nunca serán suficientes para describir mi cariño y complaciencia.

A todos, mi mayor agradecimiento y gratitud.

CAPÍTULO 1

Motivación y objetivos del trabajo

En esta sección vamos a tratar las razones por las que hemos decidido realizar este trabajo, así como los objetivos que pretendemos alcanzar, y una breve descripción de los capítulos que forman esta memoria bibliográfica.

1.1 Motivación

Cuando me preguntaron en qué universidad quería estudiar, sin pensarlo mucho siempre respondía en la Universitat Politècnica de València[1] y es que siempre había oído hablar a mi alrededor que era una buena Universidad, y aquí acabé.

Una vez dentro de la Escola Tècnica Superior d'Enginyeria Informàtica, fui conociendo y recorriendo poco a poco las diferentes secciones que la forman y me llamó mucho la atención el Museo de Informática[2], ya que al principio me parecieron un montón de trastos viejos, pero conforme vas superando los cursos académicos, te vas dando cuenta de dónde salen esos artilugios y que algunos de ellos pueden entrar en materia de examen y si no prestas atención, esa pregunta no estará correctamente contestada[3].

También debo añadir que el Museo ofrece otras muchas actividades a parte de la exposición de equipos informáticos, estas se pueden encontrar en su página web, citada anteriormente.

Para concluir con esta apartado, expondré una breve explicación sobre mi elección de el Trabajo de Final de Grado.

Siempre hemos podido ver a lo largo de la historia, que la mujer ha estado en segundo plano frente al hombre, y en muchas ocasiones cuando un descubrimiento era realizado por una mujer, tenía que disfrazar su nombre por miedo a el rechazo o a la censura; en cambio Ada Byron, trabajó sin ningún reparo con Charles Babbage, fiel a su pensamiento y a su trabajo, demostrando el importante papel de la mujer en la historia de la informática.

Este año se celebra el 201 aniversario del nacimiento de Ada Byron y que mejor honra que realizar mi Trabajo de Final de Grado por y para ella.

[1] Cuya página web es http://www.upv.es.

[2] http://museo.inf.upv.es.

[3] Ejemplo de las cintas magnéticas preguntadas en la asignatura de Diseño, Configuración y Evaluación de los Sistemas Informáticos, DCE.

1.2 Objetivos

Si tuviéramos que describir el origen de las máquinas de calcular, nos remontaríamos a la antigua Mesopotamia[4] con el uso del ábaco chino[5].

Otro de los hechos importantes en la evolución de la informática lo situamos en el siglo XVII, donde el científico francés Blas Pascal inventó una máquina calculadora[6].

En el siglo XIX, el matemático británico Babbage con la colaboración de Ada Byron desarrolló la Máquina Analítica, la cual podía realizar cualquier operación matemática a través de las operaciones elementales. Los historiadores consideran la Máquina Analítica como el antecedente de los ordenadores modernos.

En el primer tercio del siglo XX, con el desarrollo de la electrónica, se empiezan a solucionar los problemas técnicos que acarreaban estas máquinas, reemplazándose los sistemas de engranaje y varillas por impulsos eléctricos.[7]

Con el desarrollo de la segunda guerra mundial se construye el primer ordenador, Mark I y su funcionamiento se basaba en interruptores mecánicos. En 1944 se construyó el primer ordenador con fines prácticos que se denominó ENIAC. En 1951 son desarrollados el Univac I y el Univac II.

Conforme los años van trascurriendo, los avances tecnológicos son más importantes y los computadores avanzan con pasos agigantados, lo que hace unos años debía de estar en grandes salas con una amplia ventilación y con muy poca capacidad de almacenamiento, ahora lo podemos encontrar en pequeños cuartos y una cantidad impensable de memoria.

Por lo tanto este trabajo persigue tres grandes objetivos, estudiar y comprender la vida de Ada Byron, su importancia en la historia de la informática mediante su programa que calcula los números de Bernoulli y su contribución a la Máquina Analítica de Charles Babbage.

1.3 Contexto histórico

Nuestro trabajo se desarrolla en el siglo XIX, concretamente entre los años 1833 y 1845, por lo que haremos hincapié en estas fechas.

El verano de 1833, la alta sociedad londinense la formaban apenas cinco mil personas, emparentadas casi todas por matrimonio o por adulterio. Tenían un gran capital en el banco, disfrutaban de la mejor comida y bebida, la mayoría no necesitaba trabajar, y tenían a su cargo un ejército de criados.

Su vida era una continua sucesión de grandes almuerzos, tardes ociosas, veladas espléndidas y fogosos encuentros. El resto de la población, sobrevivía con una dieta no muy por encima del nivel del hambre, y aferrándose a las hebras de felicidad que encontraban entre la miseria y la mugre.

[4] Denominada con este nombre a la zona geográfica de Oriente Próximo ubicada entre el río Tigris y el río Éufrates

[5] Tabla de madera dividida en columnas formadas por bolas. A través de los movimientos de dichas bolas se podían realizar operaciones aritméticas sencillas (sumas, restas y multiplicaciones).

[6] Ésta sólo servía para hacer sumas y restas, pero este dispositivo sirvió como base para que el alemán Leibniz, en el siglo XVIII, desarrollara una máquina que, además de realizar operaciones de adición y sustracción, podía efectuar operaciones de producto y cociente.

[7] Los impulsos eléctricos se representaban con un 1 si había paso de corriente, y con un 0 si no lo había, como se hace actualmente.

La buena sociedad se concentraba en una zona relativamente pequeña de Londres con el río Támesis al sur. El matemático Charles Babbage, que desempeñará un papel decisivo en la vida de Ada, vivía en esta zona, concretamente en Dorset Street, cerca de Manchester Square.

Los aristócratas y la gente corriente parecían de mundos distintos. A veces, la riqueza o el poder político le franqueaban a un plebeyo el acceso a la nobleza; pero el matrimonio era entonces, el medio más fácil. Si bien los nobles se caracterizaban por la endogamia, de vez en cuando un plebeyo tenía suerte. Las fortunas de la nobleza procedían casi siempre de herencias.

En la tercera década del siglo XIX, las estructuras sociales parecían en la superficie más rígidas que nunca. Sin embargo, es cierto que Gran Bretaña estaba creciendo con rapidez. Entre las causas principales se hallaba la enorme repercusión de la maquinaria y la tecnología que tanto maravillaban a Ada.

Esta época fue denominada por Thomas Carlyle[8] en su ensayo *El signo de los tiempos* de la siguiente manera: «si hubiera que elegir un término para designar nuestra época, no la llamaríamos la era heroica, ni religiosa, ni filosófica, sino la era mecánica. En esta era, que enseña y practica con una fuerza avasalladora el gran arte de adeudar los medios a los fines, el artesano se le expulsa de su taller, que se va sustituido por artefactos más veloces; y la lanzadera pasa del tejedor a unas manos de hierro que la manejan más rápido.»

De esta idea fueron participes muchos, pero sobre todo Ada y su futuro amigo, Charles Babbage. Carlyle estimuló la imaginación de la gente sobre cómo "podían"hacerse las cosas, permitiendo a los creativos pensadores especular sobre la posibilidad de combinar unas máquinas con otras e imaginar nuevos usos que todavía no eran factibles tecnológicamente. En este aspecto, Ada tenía una manera de pensar fuera de lo común, que la convertiría en una de las más innovadoras de su tiempo.

En torno al año 1815 Gran Bretaña, después de veinte años de guerra con Francia, había pasado a ser la primera potencia económica y militar del mundo. Esta nación tan dinámica y segura de sí misma tenía por soberano al rey Guillermo IV, antiguo duque de Clarence y tercer hijo de Jorge III. Ascendió al trono en junio de 1830. En 1832 se aprobó la Ley de Reforma, creando nuevas jurisdicciones que incorporaran a la nueva clase media urbana y excluyera a los suburbios.

En las primeras décadas del siglo, el censo electoral era muy reducido; sólo el tres por ciento de los adultos varones tenía derecho al voto. Sin embargo, después de la aprobación de la Ley de Reforma, el sufragio estaba restringido a los hombres que acreditaran un nivel de renta determinado. A las mujeres se les denegaba el derecho a votar y a participar en la vida pública.

En una sociedad que no creía que las mujeres fueran aptas más allá de tener hijos y dar placer a los hombres, Ada jamás disfrutaría de las mismas oportunidades que Babbage.

Con la Ley de Reforma de 1832, la clase dirigente creyó la posibilidad de una revolución social, debido a los cambios acontecidos. La economía había crecido con rapidez en la última década. La hegemonía británica se debía a la amplia importación de los países conquistados. El crecimiento de la industria textil era ya notable.

A mediados del siglo XVIII, Gran Bretaña apenas había exportado tejidos, y en 1833 constituía la mitad del comercio exterior. La expansión del sector dependía del carbón y la siderurgia. El carbón era el combustible de los motores a vapor que propulsaban las máquinas de las fábricas textiles, y el hierro era utilizado para construir dichas máquinas.

[8]Thomas Carlyle (1795-1881) fue historiador, crítico social y ensayista escocés. De buena familia

Un siglo antes, en torno al año 1733, el invento John Kay había patentado la «lanzadera volante»[9]. Este invento facilitó el proceso textil y aumentó el consumo de hilo.

En 1832, casi todas las prendas fabricadas en Gran Bretaña se tejían a través de telares mecánicos. Se pensaba que el vapor iba a ser la fuerza motriz de todos los inventos, como la calculadora que Charles Babbage deseaba construir o la máquina de volar que imagina Ada. El motor a vapor, permitió a las máquinas funcionar con mayor rapidez y seguridad que las fuentes de energía tradicionales: la humana, la animal y la hidráulica.

Fue el inventor escocés James Watt quien convirtió el vapor en el factor decisivo del proceso conocido como Revolución Industrial[10]. En 1698, Thomas Savery había patentado una bomba a vapor, pero era extremadamente difícil de manejar. Watt corrigió los errores de este artilugio; su trabajo llamó la atención de un industrial de Birmingham, Matthew Boulton, que le encargó construir una máquina de vapor. En 1775 se asociaron y empezaron a construir máquinas juntos.

En 1833, en Gran Bretaña proliferaban las fábricas compuestas de máquinas de vapor rotativas, cuyas bielas se conectaban por un mecanismo de ruedas, correas y poleas a docenas de telares individuales y distintos tipos de hiladoras. Los aparatos británicos que se construían eran muy eficaces y complejos en su diseño, pero no tenían demasiadas aplicaciones. La madre de todas las máquinas se encontraba en el país vecino (y gran enemigo, durante siglos de Gran Bretaña), en Francia.

En esta época no había ningún armatoste textil tan avanzado tecnológicamente como el telar de seda patentado por Joseph-Marie Jacquard en 1804. El telar de Jacquard, era un extraordinario invento que servía para fabricar tejidos de lujo con patrones muy complejos (retratos, bodegones, paisajes,...); esta máquina fascinaba a Ada.

Antes de que se introdujera este telar, los dibujos se elaboraban muy despacio, y el proceso lo llevaban a cabo dos tejedores; uno de ellos manejaba la lanzadera, y el otro se colocaba encima del telar, en una plataforma desde la que tiraba de cientos de cuerdas, subiendo y bajando los hilos de la malla para elaborar la imagen deseada.

Este telar facilitó el proceso, permitiendo a un único operario reproducir automáticamente el dibujo con una serie de cartulinas perforadas que iban configurando la malla según un patrón determinado. En el año 1832, el mecanismo ya era famoso en Gran Bretaña. A Ada le maravillaba la posibilidad de idear y construir máquinas como la de Jacquard.

Charles Babbage no fue el primer inventor que se propuso construir una calculadora mecánica fidedigna, pero sí el que estuvo más cerca de conseguirlo en el siglo XIX. Babbage se basó en la idea de Blaise pascal que en el año 1642, ideó un aparato para sumar y restar y así poder ayudar a su padre con las cuentas. La denominada «pascalina» era una caja rectangular con ruedas que se accionaban con una manivela y que correspondían a las decenas, las centenas, etc. Cada rueda tenía diez dientes, para representar los dígitos del 0 al 9. Pese a esta idea tan novedosa, el invento no fue muy fiable ni contribuyó a la mecanización del cálculo.

En la época de Ada, la primera mitad del siglo XIX, no era común la participación de las mujeres en las ciencias y las artes. También estaban restringidos sus accesos a las universidades. Sin embargo, es oportuno notar que en la Gran Bretaña de esa época existía

[9] Kay aumentó la eficacia del telar con la invención de la lanzadera volante, provista de cuatro rodillos que se movía por medio de dos raquetas de madera y de un cordel que el tejedor sostenía en su mano. Esto posibilitó la fabricación de tejidos más anchos que antes en el campo de acción del brazo humano. La lanzadera volante permitía que la labor de tejido, en la que intervenían dos trabajadores, fuera realizada por uno solo, así como tejer piezas de algodón a mayor velocidad de lo que se podría lograr con la habilidad manual del trabajador.

[10] Finales del siglo XVIII, mediados del siglo XIX.

una publicación anual, *The Ladies Diary* o *Woman's Almanack*, dirigida a las mujeres, que incluía temas avanzados de matemáticas.

Junto a los progresos en el orden científico, tecnológico y económico, el papel de las mujeres en la sociedad también daba sus pasos de avance.

1.4 Estructura de la memoria

Nuestro memoria consta de cinco capítulos, y en este apartado, nos disponemos a realizar una breve descripción de cada uno de ellos; así como mencionar la presencia de varios anexos al final de la memoria, en los que se incluye la instalación y el código para poder realizar nuestro trabajo.

- Motivación y objetivos: en este capítulo se expone la motivación por el cuál se ha llevado a cabo el trabajo, así como los objetivos y el contexto histórico a tratar.

- Los protagonistas: en este capítulo hablamos sobre la influencia que tuvieron tanto Charles Babbage y sus máquinas como Ada Byron en la historia de la informática y las principales aportaciones a la misma.

- Análisis de las notas de Ada Byron: en esta parte hemos realizado una descripción detallada de cada una de las notas de Ada, destacando las aportaciones más significativas de cada una de sus notas.

- Cálculo de los números de Bernoulli: en este capítulo, explicamos ampliamente el funcionamiento para el cálculo de los números de Bernoulli, así como el cálculo que da Ada para dichos números.

- Conclusiones: en este capítulo se detallan los problemas que hemos sufrido a lo largo de el desarrollo de el trabajo así como las consideraciones personales a las que hemos llegado tras la finalización de este.

1.5 Notas sobre la bibliografía

En este apartado vamos a mencionar la bibliografía que hemos consultado para la realización de nuestro trabajo, centrándonos en cómo se ha aprovechado y cuáles de las opiniones que se nos proporcionan pueden ser útiles para la redacción de nuestra memoria.

Como se comenta al inicio del capítulo, la motivación principal del trabajo es demostrar que las mujeres (centrándonos en Ada Byron) son tan válidas como los hombres para todo tipo de actividades y en todos los campos.

Para poder argumentar y defender esta idea, el punto de partida de nuestra tarea ha sido analizar y comprender el contexto histórico en el que vivió la protagonista, para ello hemos usado las referencias bibliográficas [4, 8, 14, 19, 20, 22, 21].

El siguiente paso en la elaboración del trabajo, ha consistido en el amplio estudio de la vida de Ada Byron, desde que apenas tenía un mes de vida hasta el día de su muerte, cuyas referencias son [2, 3, 12, 10, 13, 14, 17, 19, 23].

Al mismo tiempo que hemos estudiado la vida de Lady Lovelace, hemos investigado sobre Charles Babbage (vida y obra) ya que tuvo un papel decisivo en la vida de la protagonista, cuyas referencias son [1, 6, 7, 14, 19].

Por último, hemos decidido incluir los inventos que Babbage intentó hacer realidad, como son la Máquina Diferencial y la Máquina Analítica con referecnias [1, 9, 11, 13, 14,

19, 6]; ya que gracias a estos artilugios, Ada creó sus *Notas* con referencias [3, 15, 10, 16, 18, 19] y se le ocurrió la brillante idea de crear el primer programa informático de las historia adaptado a la Máquina Analítica, los Números de Bernoulli con referencia [5].

CAPÍTULO 2
Los protagonistas

Este capítulo trata sobre la vida de los protagonistas de nuestro estudio. Podemos encontrar una sección dedicada a cada uno de ellos y otra para las máquinas, ya que sin ellas nada de esto sería posible. En las secciones que hacen referencia tanto a Charles Babbage como a Ada Byron aparece una explicación detallada de sus vidas así como el inicio y la posterior relación de amistad entre los protagonistas.

2.1 Charles Babbage

Charles Babbage, como se muestra en la Figura 2.1, nació el 26 de diciembre de 1791 en Walworth[1], en la orilla del Támesis.

Figura 2.1: Charles Babbage.

Su padre Benjamin Babbage, nacido en 1753, era orfebre y banquero. Las dos profesiones estaban íntimamente unidas: era muy práctico que los clientes que compraban joyas y oro guardaran sus bienes en las cajas fuertes del orfebre, y que el orfebre les prestara dinero sobre sus bienes cuando sus clientes lo necesitaran.

La suya era una familia bien arraigada desde finales del siglo XVII en Totnes, una aldea del condado de Devon, alrededor de unos cincuenta kilómetros de Londres. Ac-

[1] Actualmente este barrio se encuentra situado en el centro de Londres. Su nombre deriva del antiguo inglés "Wealhworth", cuyo significado es "granja".

tualmente es un pueblo que goza de mercado y gran popularidad entre los seguidores de la *New Age*[2] y las formas de vida alternativas.

Totnes debía su prosperidad a la lana obtenida de las ovejas que pastaban en las praderas de los alrededores y a su productos derivados; Benjamin Babbage fue poco a poco invirtiendo y creó un negocio boyante en Totnes que le abriría camino a las ciudades y los pueblos cercanos; cerraba acuerdos informales, actuando por cuenta propia y en representación de bancos de Londres. Era un comerciante sagaz.

Sin embargo, desde principios de la década de 1790, la industria local estaba en declive por la introducción de la máquina a vapor en el sector textil. El astuto Benjamin comprendió enseguida este cambio y traslado su negocio a Londres en el año 1791. Una vez allí fue nombrado consejero de Praeds Bank. Siempre había sido muy emprendedor así que no tuvo problema para prosperar. Benjamin tenía mal carácter, se impacientaba con Charles y le reprochaba la falta de ambiciones profesionales. En un primer momento se opuso a que se casara con Georgiana, argumentando que antes tenía que progresar en un oficio respetable.

Georgiana venía de una familia ilustre, poseía dinero y era una muchacha bondadosa y encantadora. Pero Benjamin consideraba que todo joven tenía que seguir su ejemplo y hacerse rico antes de casarse.

A la muerte de su padre, en 1827, Charles heredó todo su patrimonio, incluidas las cien mil libras que tenía en el banco. Babbage se consideraba, en efecto, filósofo más que matemático o científico[3]. Su padre, comerciante acaudalado, estaba en condiciones de proporcionarle una buena educación.

Cuando tenía ocho años, fue enviado a la escuela de Alphington, cerca de Exeter; más tarde, al colegio de King Edward VI, en Totnes y a otro en Enfield. Charles se educó casi siempre con preceptores.

En Enfield tuvo un excelente mentor, el reverendo Stephen Freeman, que le contagió su pasión por las matemáticas. Siempre fue un niño obediente, al contrario que su amiga Byron. Ya de niño tenía mucha curiosidad por saber cómo funcionaban las cosas. Deseaba «averiguar el mecanismo de los pequeños objetos y las causas de los pequeños fenómenos que asombraban a los niños».

Cuando a los cuarenta y dos conoció a la joven de diecisiete, Ada y a su madre, no había cambiado apenas. Sus padres alentaron esmeradamente su entusiasmo. Cuando vivían en Londres, Elizabeth, su madre, le llevó a varias exposiciones de maquinaria, entre ellas la organizada en Hannover Square, donde se mostraron dos figuras de plata que le habían costado al artista toda su vida y ni siquiera estaban terminadas; a Babbage le iba a suceder lo mismo con sus máquinas.

Charles ingresó en la Universidad de Cambridge cuatro años más tarde que el padre de su amiga, Lord Byron; pero no llevó una vida disoluta ni se endeudó. Le apasionaba en ajedrez y fundó una sociedad con sus amigos llamada el Club de los Fantasmas, en la que se dedicaban a reunir pruebas para demostrar que los fantasmas existen.

Lo cierto es que Babbage, que siempre se había visto respaldado en sus entusiasmos, era un poco diletante. Se apasionaba fácilmente, pero no solía terminar lo que emprendía. Al contrario que su amigo Charles Dickens, se podía permitir el lujo de ser inconstante, pues contaba con la herencia paterna. Por lo demás, le repugnaba la idea de ganar dinero con sus inventos. Quería trabajar a su aire. Cuando ingresó en el Trinity Collage de Cambridge, Gran Bretaña ya estaba en medio de una revolución tecnológica sin precedentes.

[2] Se trata de un movimiento cultural basado en la Era de Acuario, cuya creencia tiene como argumento que cuando el Sol «pasa» de un signo del zodíaco al siguiente, se producirían cambios en la humanidad.

[3] Término que no se hizo común hasta la década de 1890.

Babbage aspiraba a contribuir de algún modo a esa revolución, así que se apartó de los examenos, para estudiar lo que le interesaba en matemáticas y en ciencias.

En Cambridge tenía dos amigos, George Peacock y John Herschel[4], con los que fundó la llamada Sociedad Analítica, que tenía como objetivo principal reformar la enseñanza del cálculo en la universidad sustituyendo la notación de Newton por la de Leibniz, que consideraban mucho más eficaz. La campaña acabó triunfando, pero para entonces Babbage ya se había licenciado. Terminó sus estudios en 1814 y más tarde paso a ser titular de la prestigiosa Cátedra Lucasiana de Matemáticas. La brillantez de los argumentos esgrimidos a favor de la reforma hizo que el mundo matemático empezara a fijarse en los fundadores del grupo, Babbage y Herschel.

La relación entre Babbage y Herschel fue, para cada uno de ellos, la primera amistad intelectual importante. Se tenían tanto aprecio que Babbage bautizó a su primer hijo como Benjamin Herschel en homenaje a su padre y a su amigo. Para Babbage, que sufrió tantos reversos en la vida, la amistad con Herschel fue en muchas ocasiones una vía de escape y consuelo.

Estando vivo su padre, Charles estuvo muy interesado en encontrar un trabajo con el que ganarse la vida, pero no encontró ninguno que le satisficiera ni le gustara. A mediados de septiembre de 1815, el matrimonio se instaló en el número 5 de Devonshire Street, en Londres. Babbage pronto alcanzó renombre en los círculos científicos.

En 1815 (el año en el que nació Ada) pronunció una serie de conferencias sobre astronomía en la Royal Institution[5], y en la primavera de 1816 fue elegido miembro de la Royal Society[6], que reunía a los científicos más eminentes.

En los años siguientes el trabajo de Babbage fue principalmente matemático: publicó más de una docena de artículos. Babbage, como Byron, adoraba a sus hijos, pero después de morir Georgiana en 1827, los dejó al cuidado de su madre, Elizabeth. Así, la única niña vivía con su abuela, y el hijo mayor, Herschel, pasaba las vacaciones escolares con su abuela.

Aunque le gustaba alternar en sociedad, Babbage podía ser peculiarmente excéntrico, y tendía a mostrarse hosco, malhumorado y a veces engreído y sabelotodo. Tenía la manía de tratar las cosas triviales como fenómenos dignos de análisis matemático, lo que lo convertía en una atracción en las fiestas elegantes.

Viajó numerosas ocasiones a Francia con su amigo Herschel, la primera vez fue en 1819, pero se trataba de viajes intelectuales, para intercambiar ideas e información con matemáticos y científicos franceses: el prestigio de sir William, el padre de su amigo, les permitió conocer a varios muy notables.

Fue en uno de estos viajes cuando vio por primera vez el telar de Jacquard y su método de tarjetas perforadas, que posteriormente incluiría en su diseño de la máquina analítica.

Fue seguramente en su primer viaje a París cuando oyó hablar de un ambicioso proyecto que se había llevado a cabo en Francia a finales del siglo anterior. Se trataba de elaborar tablas matemáticas fidedignas para el Servicio Nacional de Cartografía. El proyecto fue dirigido por el barón Gaspard de Prony, un ilustre ingeniero civil perteneciente al período del Terror de la Revolución francesa.

[4]Hijo del famoso astrónomo sir William Herschel, que en 1781 descubrió Urano, el primer planeta hallado con un telescopio

[5]Se trata de una organización dedicada a la educación y la investigación científicas situada en Londres. Fue fundada en 1799 por los principales científicos británicos de la época.

[6]Se trata de la sociedad científica más antigua de Reino Unido, fundada en 1660. En esta sociedad, trataban temas como asuntos de estado o actualidad, medicina, anatomía, geometría, navegación, entre otros, y los experimentos que desarrollaban.

El Servicio de Cartografía tenía que hacer infinidad de multiplicaciones con números grandes, y para simplificar y agilizar el trabajo necesitaba tablas logarítmicas[7] precisas. De Prony se propuso calcular los logaritmos de los números comprendidos entre el 1 y el 200.000. La Revolución seguía cortando cabezas, y a De Prony, le aterraba que su proyecto fracasase.

De Prony se basó en el principio de la división del trabajo para construir las tablas con la máxima precisión y en un tiempo razonable, así pues dividió a sus colaboradores en varios equipos. Las tablas obtenidas con el novedoso procedimiento de Prony ocupaban diecisiete volúmenes enormes y se consideraban tan fiables que el ejército francés las utilizó hasta 1940 para cálculos topográficos. A la inteligencia analítica de Babbage le sedujo el método que basándose en la producción en serie, había ideado de Prony para organizar un gigantesco proceso de cálculo.

A la hora de construir la primera calculadora con ruedas dentadas, el científico inglés adoptó un procedimiento similar, el método de las diferencias, que reducía el cálculo de las tablas a operaciones sencillas. Este artefacto entrañaba grandes dificultades conceptuales y técnicas, que Babbage resolvió con un ingenio fuera de lo común.

Sin embargo, no llegó a construir más que la séptima parte de la máquina, y fue este embrión lo que Ada vio por primera vez en 1833, cuando se conocieron en la fiesta celebrada en casa de Mary Somerville, amiga de ambos, como veremos más adelante en el apartado siguiente.

Babbage empezó con su proyecto de la Máquina Diferencial, y acabó diseñando, ya que nunca la llegó a construirla, una calculadora controlada por tarjetas perforadas, la mundialmente conocida Máquina Analítica.

Tras gastar casi todo su patrimonio en dar vida a su ambicioso invento, Charles Babbage falleció en Londres el 20 de octubre de 1871.

2.2 La Máquina Diferencial

La Máquina Diferencial fue diseñada por Babbage en el año 1822; en la Figura 2.3 podemos observar una parte de la misma. El gobierno británico financió inicialmente el proyecto proporcionándole al autor una subvención de mil quinientas libras, permitiéndole así acelerar la construcción del prototipo.

A partir de 1827, Babbage aumentó su trabajo. El duque de Wellingtton, se interesó por el proyecto, ayudando a Babbage con la cantidad de diecisiete mil quinientas libras. Para justificar esta extraordinaria cantidad, bastaba con que Babbage enseñara el prototipo. Lo cierto es que gastó demasiado dinero en perfeccionar los componentes que la formaban. La máquina poseía varios millares de ruedas dentadas, que debían seguir unas especificaciones muy concretas.

En el verano de 1833 ya había construido parte de su amada máquina y era bien conocida en Londres. El proyecto, sin embargo, estaba estancado: ya que debido a no haber una industria de precisión se tenían que construir a mano cada uno de los componentes. Proceso extraordinariamente duro, lento y costoso.

Babbage poseía dinero, pero sus fondos no eran ilimitados. Para estas fechas Ada y Babbage ya se habían conocido. Madre e hija acudieron a su casa y como agradecimiento

[7] Un logaritmo es la potencia a la que es necesario elevar una cantidad positiva para que resulte un número determinado. Este concepto fue introducido por John Napier en el siglo XVII. Un logaritmo decimal tiene como base el número diez.

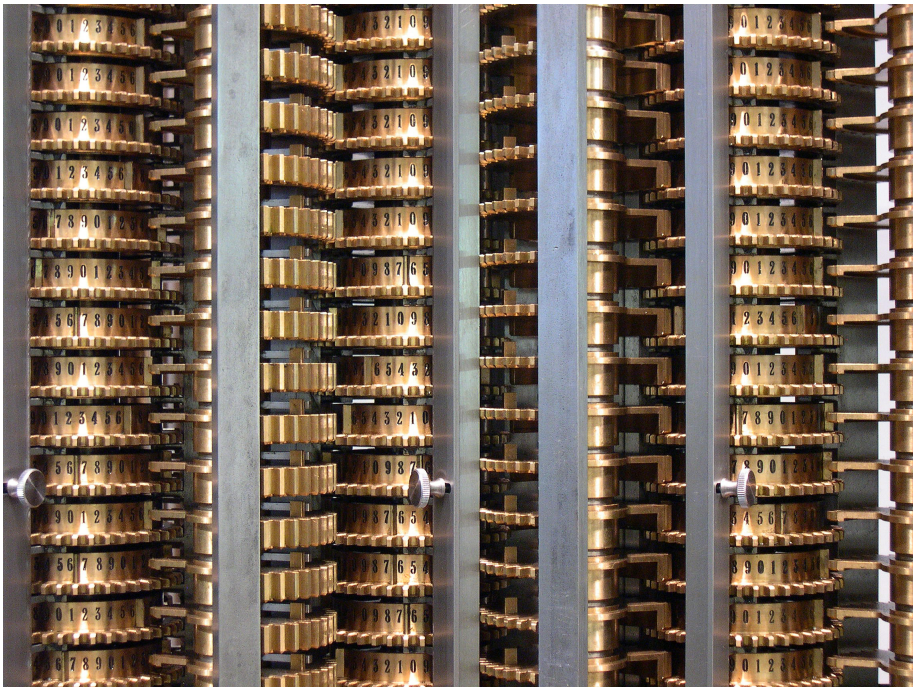

Figura 2.2: Engranajes de la Máquina Diferencial.

por la visita, Babbage les enseñó lo que tanto dinero y fatigas había costado construir, la séptima parte de la Máquina Diferencial.

Ada describió la máquina de la siguiente manera: «la máquina pensante»[8] es capaz de elevar al cuadrado y al cubo, y calcula la raíz de una ecuación de segundo grado. La máquina cuenta 1,2,3,4... hasta llegar a 10.000 y luego sigue otra progresión. Las excepciones que se producen al operar la máquina, sigue un número de experiencias uniformes mayor que el número de días y noches que ha conocido el universo.

Ada supo ver más allá del maravilloso movimiento de las ruedas, que hacía de las matemáticas una experiencia real y estimulante. Al contrario que su madre, la joven comprendió cómo la extraordinaria inteligencia de Babbage había conseguido enlazar el mundo abstracto de las matemáticas con un objeto mecánico, siendo la clave el uso de las ruedas dentadas (única tecnología disponible en la época).

El fundamento de la máquina se basa en la unión entre los dientes de las ruedas y los diez dígitos que las forman como vemos en la Figura 2.3. El funcionamiento consistiría en hacer girar las ruedas de manera independiente, dando lugar así a un cálculo aritmético.

Babbage adaptó el sistema de numeración decimal para las ruedas[9]. De tal manera que, en cada columna, la primera rueda empezando por abajo representara las unidades; la segunda, las decenas; la tercera, las centenas, y así sucesivamente.

Esta máquina fue denominada con el nombre de «Máquina Diferencial» ya que se basa en el "método de las diferencias"[10] para realizar los cálculos. La belleza de las ruedas numéricas de Babbage radica en la capacidad de efectuar las sumas. La máquina, por tan-

[8] Ada describe así a la Máquina Diferencial, muy original para la época y muy razonable para la actualidad debido a la manera de trabajar que poseía la máquina.

[9] El sistema de numeración decimal es un sistema de numeración en el que las cantidades se representan utilizando como base aritmética las potencias del número diez.

[10] Este método permite calcular tablas matemáticas sumando las diferencias entre los términos de una serie. Una serie es un conjunto ordenado de números que se derivan unos de otros según una ley determinada. Éste método nos permite simplificar el cálculo en series muy largas y complejas. El sistema permite sustituir multiplicaciones muy complejas por sumas sencillas.

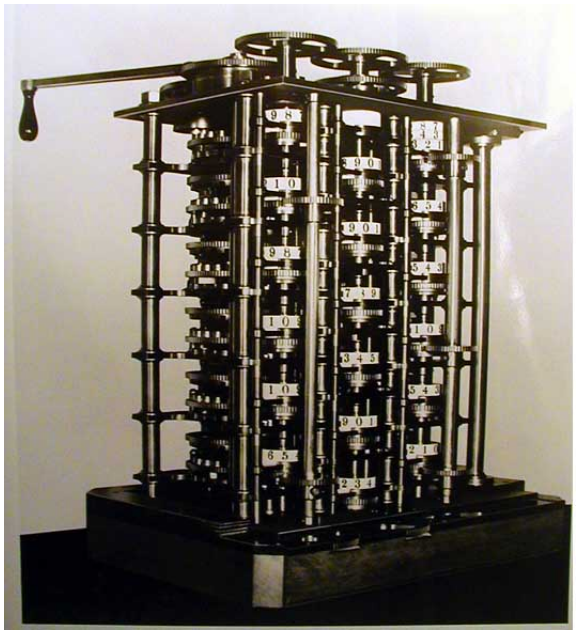

Figura 2.3: Máquina Diferencial.

to, poseía todos los elementos que entusiasmaban a Ada: para ella, el invento de Babbage demostraba que un día «las matemáticas harían posible poder volar» como ya intentó de cría.

A Ada y Babbage, pese a sus diferencias, les interesaban los mismos problemas matemáticos. Al matemático, le interesaba la idea de la predestinación. También le intrigaba la dificultad de conciliar la creciente importancia de las máquinas, que estaban al servicio del hombre.

Le asombraban la regularidad y fiabilidad de las máquinas, y le complacía enormemente observar el funcionamiento de un artefacto bien calibrado, donde el mecanismo volvía invariablemente al mismo punto del ciclo después de un intervalo determinado. Su inteligencia, sin embargo, era más prosaica que la de Ada: para él, la tecnología era fundamentalmente un instrumento que satisfacía las necesidades humanas, y no un estímulo para la especulación teórica.

2.3 La Máquina Analítica

Para poder entender bien el trabajo que Ada realizó sobre la Máquina Analítica, primero daremos una explicación detallada de la misma.

La Máquina Analítica surgió a raíz del intento de solventar un importante problema mecánico que planteaba la Máquina Diferencial, así pues ésta quedó sustituida por la primera.

Esta máquina puede considerarse la más brillante invención teórica del siglo XIX, ya que no ha sido construida por completo. La Máquina Analítica tiene su origen en el segundo artículo que Babbage dedicó a la Máquina Diferencial, y que leyó en la Royal Astronomical Society[11] el 13 de diciembre de 1822, cuando Ada apenas acababa de cumplir los siete años.

[11] Sociedad fundada en 1820 en Londres bajo el nombre de .Astronomical Society of London". En 1831 pasó a llamrse Royal Astronomical Society". La Sociedad organiza una reunión mensual, generalmente el segundo viernes de cada mes en Londres.

En el artículo reconocía que a pesar de ser útil, la Máquina Diferencial tenía un grave inconveniente: había que reajustarla por cada nueva serie de cálculos que se quisieran realizar, introduciendo a mano los números en las ruedas dentadas. Una vez accionada la manivela, la máquina realizaba los cálculos automáticamente, sin que uno tuviese que intervenir de nuevo. Sin embargo en ciertos casos, los resultados iban perdiendo precisión a medida que se elaboraban las tablas. Hacía falta una maquina que evitara esta continua pérdida de exactitud y no se limitase únicamente a calcular tablas.

El nuevo armatoste, pronto superó a la máquina diferencial. En los meses que siguieron a su encuentro con Ada y su madre, Babbage se dedicó sin descanso a llevar a cabo su idea. La nueva máquina sería enorme, un aparato del tamaño aproximado de una locomotora a vapor como vemos en la Figura 2.4. Poseería hasta veinte mil ruedas dentadas, dispuestas en columnas y accionadas por palancas, levas y varillas, que emitirían los cálculos de unas partes a otras de la máquina.

Babbage realizó diversos bocetos de las piezas e incluso llegó a construir algunas para las ruedas. La principal característica de esta máquina fue el uso de tarjetas perforadas con las que se controlaría el funcionamiento. Ésta no fue la única idea que barajó para hacerla funcionar. Antes de optar por las tarjetas, consideró la posibilidad de programar la máquina con un tambor giratorio provisto de pequeñas varillas. Este tipo de tambor fue la base del sistema de control que el inventor francés Jacques de Vaucanson había empleado en su telar, precedente del de Jacquard.

Babbage acertó usando este sistema, ya que era mucho más fácil y barato elaborar las tarjetas que fabricar tambores mecánicos. A parte que por medio de estas tarjetas, la programación podría llegar a ser ilimitada, mientras que un tambor giratorio por definición, no tardaba en repetirse.

El código de nuestros ordenadores procede directamente del telar de Jacquard y su sistema de programación: podemos considerar a la tarjeta perforada el antecedente del dígito binario o bit[12]. Esta máquina estaría compuesta de dos componentes: el almacén, donde se encontrarían todas las variables que intervienen en el cálculo y la fabrica, donde se realizarían las operaciones. Decidió denominarlos así pensando en la industria textil que se encontraba en auge en Gran Bretaña en esta época.

La Máquina Analítica sería capaz de realizar operaciones algebraicas a partir de valores numéricos asignados a ciertas variables. Babbage y Ada hacen distinción entre tres tipos de tarjetas, como veremos con más detalle en el capítulo destinado a las Notas, las tarjetas de operaciones que se encargarían de controlar las operaciones que realizaría la máquina, las tarjetas de variables que indicarían en que parte del almacén están los números con los que se va a operar y las tarjetas numéricas en las que figurarían los valores numéricos que se debían introducir en la máquina.

Así pues, en la Máquina Analítica podíamos encontrar, asombrosamente, los elementos básicos de un ordenador moderno (como vemos en la Figura 2.5):

- **Mecanismos de entrada,** formado por tarjetas perforadas. La máquina era capaz de distinguir los diferentes tipos de tarjetas que se introducían por las ranuras.

- **Memoria,** capaz de almacenar mil números de cincuenta dígitos cada uno.

- **Almacén,** se trataba de un mecanismo que controla que las operaciones se realizasen en el orden adecuado, según las instrucciones del programa contenido en las tarjetas.

- **Molino,** que realizaba las operaciones aritméticas y las discriminaciones lógicas, lo que hoy en día conocemos como la Unidad Aritmético-Lógica.

[12]Un bit es la unidad de información más pequeña de la informática. Se formula como una elección entre dos posibilidades o 0 o 1. El 0 representa estado “apagado” y el 1 estado “encendido”.

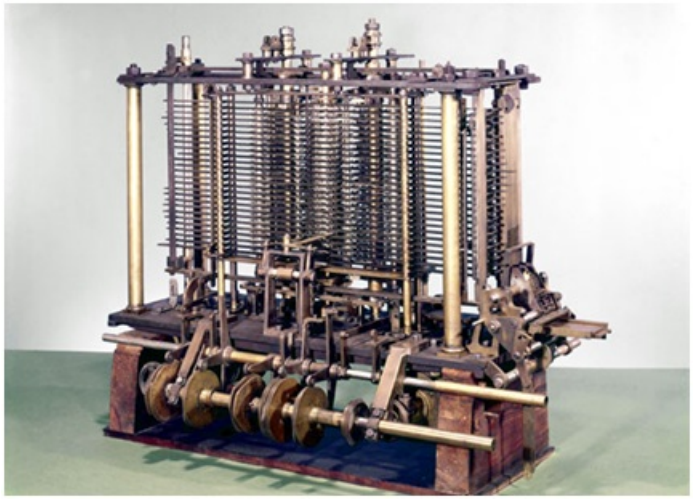

Figura 2.4: Máquina Analítica.

- **Mecanismos de salida,** se trataba también de tarjetas perforadas con una campana que sonaba cada vez que una de las diferentes tarjetas salía.

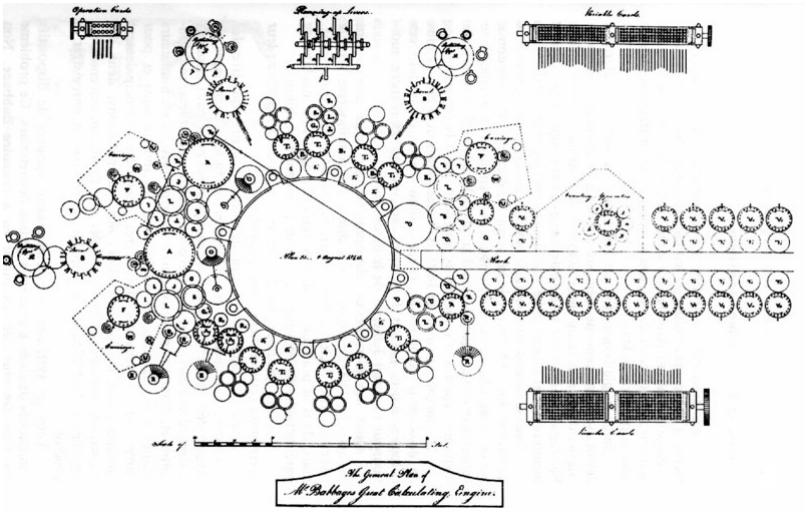

Figura 2.5: Mecanismo de funcionamiento de la Máquina Analítica.

Pese a que la máquina no pudiera construirse mientras Ada y Babbage vivieron, debido a la mala calidad de los materiales de la época y a la poca financiación por parte del Gobierno, años más tarde, concretamente en 1910, su hijo construyó una parte parcial de la máquina, y un siglo y medio después (entre los años 1989 y 1991) del diseño de Babbage, el Museo de Ciencias de Londres decidió construir un modelo funcional de la máquina, compuesta por unas ocho mil piezas pesando aproximadamente unas dos toneladas y media, de tal manera que es capaz de de imprimir automáticamente los resultados de un cálculo y un usuario puede cambiar algunos parámetros como son el espacio entre líneas, el número de columnas o elegir entre dos tipos de tipografía.

2.4 Ada Byron

Augusta Ada Byron nació el 10 de diciembre de 1815, hija del famoso poeta Lord Byron y la baronesa Anna Isabella Milbanke, como podemos ver en la Figura 2.6. La historia de sus padres, fue una historia complicada. Ada fue la única hija legítima del poeta, cuya reputación no era la oportuna debido a sus amoríos con mujeres y hombres, su pasión por su hermanastra Augusta y su calamitoso matrimonio con la madre de Ada, la joven aristócrata Anna Isabella Milbanke.

Se casaron el 2 de enero de 1815; para entonces Byron ya era bien conocido en Gran Bretaña y Europa, así como en otros continentes tanto por su agitada vida sentimental como por sus poemas. Pero Anabella lo soportó por poco tiempo. El matrimonio pasó por graves apuros económicos, algo a lo que Byron estaba acostumbrado. Así como estallidos de ira contra su mujer. Con un solo mes de vida, Annabella escapó con su hija Ada, lejos de su marido. Al cabo de un mes de su huida, las desventuras de la pareja ya eran objeto de murmuraciones en todo el país, rumoreando que Byron había sido infiel a Annabella con Augusta. Agobiado por las deudas y el escándalo de su matrimonio Byron decidió abandonar Inglaterra dedicando un poema a Ada; se trasladó a Ostende[13], donde permaneció hasta su muerte.

De niña, como podemos ver en la Figura 2.7, Ada ya despertó el morboso interés de una sociedad en la que los escándalos eran protagonistas. Su madre puso mucho empeño en protegerla, pero no pudo controlarlo todo. Lady Byron (la llamaremos a partir de ahora así), cometió el error de casarse con un hombre excéntrico, caprichoso y egoísta, y ahora, después de abandonarlo, se vio en una situación desoladora. Toda su educación a lo largo de su vida había sido dirigida para ser esposa y madre. Se instalaron en la mansión junto con sus padres, en la pequeña aldea de Kirkby Mallory.

Tras haber pasado dos semanas de su separación el matrimonio se escribió cartas en tono cariñoso con la esperanza de que su mujer y su hija volvieran; y de no haber sido por los padres de Annabella, que después de contarles el trato que había recibido por parte de Byron, se opusieron rotundamente. Unos días más tarde, lady Byron presentó una demanda de separación.

Una vez que Byron abandonó Inglaterra, constantemente escribía a su hermanastra Augusta para que le preguntara a lady Byron sobre Ada, y poder saber por ejemplo de que color tenía el pelo su hija, como se le estaba educando..., pero jamás tuvo contacto con Ada.

Lady Byron estaba decidida a darle una educación prudente. En aquella época, a los niños se les consideraba adultos imperfectos, salvajes y desagradecidos y se les vestía como adultos diminutos. De esta visión era muy participe lady Byron: Ada no podría jugar con otros niños sin la aprobación de su madre; así la mayor parte de la infancia de Ada la pasó sola.

Su educación, que comenzó con apenas cuatro años, fue todo lo amplia que podía ser para la época en la que se encontraban. Lady Byron era muy exigente con los profesores y no tenía ningún reparo en despedirlos si, consideraba que Ada no estaba aprendiendo lo suficiente. Lady Byron le impuso una disciplina estricta a través de recompensas y castigos: cuando Ada se comportaba bien y atendía el estudio, le recompensaba con unos «billetes» de papel y cuando no cumplía sus expectativas, se los quitaba. Su madre, que fue denominada por lord Byron como la «princesa de los paralelogramos» puso especial empeño en que Ada aprendiese matemáticas. Intentó reprimir su imaginación, ya que al

[13]Se trata de una ciudad belga perteneciente a la provincia de Flandes Occidental, situada en el centro de la costa belga

(a) Lord Byron **(b)** Lady Byron

Figura 2.6: Padres de Ada

Figura 2.7: Ada Byron de niña.

proceder de los Byron consideraba peligrosa y dañina, enseñándola a ser lo más cerebral posible. Ada le recordaba constantemente a su fracasado matrimonio. El de Ada no fue un genio indómito como el de su padre, ya que lady Byron se encargó de contenerlo, dirigiendo a su hija a la vida de virtud.

Desde bien pequeña Ada, sufrió casi todas las afecciones infantiles y sufría dolor de cabeza con frecuencia. A los siete años contrajo una enfermedad tan grave, que le produjo cefaleas[14] fortísimas que le afectaron a la vista y tuvo que interrumpir su educación durante un periodo. Lord Byron estuvo al corriente de las enfermedades que sufrió su hija gracias a las cartas que le envía su hermanastra Augusta. El poeta decidió viajar a Grecia para defender la lucha por la liberación del pueblo griego, dónde murió el 19 de abril de 1824 dedicándole unas palabras a su hija en el lecho de muerte: «*¡Oh, mi pobre*

[14]La cefalea o dolor de cabeza representa una de las formas más comunes de dolor en la raza humana. Hace referencia a los dolores y molestias localizadas en cualquier parte de la cabeza, en los diferentes tejidos de la cavidad craneana, en las estructuras que lo unen a la base del cráneo, los músculos y vasos sanguíneos que rodean el cuero cabelludo, cara y cuello.

niña, mi querida Ada! Dios mio, ¡ojalá te hubiera conocido! Decidle que la bendigo!». Al entierro no asistieron ni su mujer ni su hija, pero Ada fue consciente de su muerte.

En los primeros años de la educación de Ada, lady Byron recibió ayuda económica por parte de sus padres. Y tras la muerte de la abuela de Ada, su madre heredó una generosa suma, por la que ya no tuvo que volver a pasar penurias, como cuando estuvo con Byron. Criar a Ada fue un gran reto para lady Byron. La niña era famosa en todo el país por su padre. Al entrar Ada en la adolescencia, el afán educador de su madre no decayó.

A principios del siglo XIX, las oportunidades educativas para una niña eran casi nulas. Incluso las niñas de clase media y las de familias pudientes se limitaban a aprender lo justo para manejar la casa que algún día podían aspirar a dirigir. El celo de lady Byron como educadora se debía a que sus padres le habían proporcionado una educación muy buena; había aprendido historia, literatura, italiano, latín, griego, dibujo y danza.

Esto no significa que, a Ada le fuese más fácil que a su madre seguir desarrollando su energía intelectual una vez terminada su instrucción. Incluso para una niña de su condición social y con inquietudes intelectuales, las posibilidades de ejercer una profesión era casi imposibles. No había por lo general otro camino que no fuera casarse, tener hijos y vivir para el hombre con el que una se casaba.

La idea de que Ada hiciera otra cosa que casarse ni se le había pasado por la cabeza a su madre. Pero antes ésta, creía necesario llenarle la cabeza de datos, ya que así sería más difícil que cayera en los dos vicios que se le atribuían a su padre: la falta de disciplina y el exceso de imaginación. Tenía muy claro que no la animaría a dedicarse al mundo de las matemáticas, lady Byron quería que su hija se casará con un aristócrata que le asegurara una cómoda y tranquila vida.

Es cierto que algunas mujeres llegaron a dedicarse a la escritura o a las matemáticas, como Mary Somerville, que como veremos después, fue íntima amiga de Charles Babbage y profesora de Ada.

Ada casi siempre estaba rodeada de adultos, quitando a su primo por parte paterna, George, con el que jugaba siempre que iban a visitarles. Debido a esta ausencia de amigos, Ada disponía de tiempo suficiente como para inventar un aparato con el que volar. Así fue.

A principios del año 1829, Ada contrajo una grave enfermedad que le produjo una seria parálisis y la dejó postrada en cama hasta mitad de 1832. Durante este largo período, Ada quedó muy marcada, pero no perdió las ganas de estudiar; superó su tendencia a la ensoñación y se volvió una mujer más sistemática. Debido a su delicada salud, lady Byron intentaba que su hija no realizara grandes esfuerzos, pero Ada era muy exigente consigo misma.

Su recuperación fue muy lenta, casi no podía ni escribir ni moverse; perdió las ganas de montar a caballo e incluso de volar. Al pasar tanto tiempo en cama Ada había engordado bastante y ya tenía dieciséis años, por lo que faltaba muy poco para que fuera presentada en la alta sociedad londinense; con la esperanza de encontrar un marido aristócrata. Debía de salir del caparazón y enfrentarse al mundo real. En una de las veladas a las que la Ada asistió, conoció a su abogado y autor de su biografía, Woronzow Greig. Diez años mayor que Ada, nació en 1805 y era uno de los hijos varones de Marie Somervi-

lle[15]. Ada y su madre la conocieron por William Frend, un matemático de la Universidad de Cambridge, que impartió clases a madre e hija.

El 5 de junio de 1833, Ada y su madre acudieron a una fiesta de postín en Londres y fue allí dónde Ada conoció a la única persona en Inglaterra que compartía su fascinación por las cuestiones de mecánica que habían acabado cansando a su madre. Se llamaba Charles Babbage.

A Ada, que entonces tenía diecisiete años, la amistad con Babbage la estimuló mucho intelectualmente, permitiéndole avanzar es sus especulaciones sobre el futuro del cálculo hasta concebir una idea extraordinariamente audaz: la de un telar de Jacquard, pero aplicado a los números; es decir, una computadora.

Debido a la insistente educación por parte de su madre Ada, estaba constantemente cambiando de preceptores. Su último tutor fue William King.[16] A Ada le faltaba poco para cumplir dieciocho años, y su madre podía estar satisfecha de cómo iba madurando. El viernes 10 de mayo de 1833, menos de un mes antes de conocer a Babbage, se presentó a Ada en la corte; un ritual que cumplían todas las hijas de familias pudientes al hacerse mujeres. Desde ahora se le podía invitar a las fiestas distinguidas y considerar aspirante a un marido de alta cuna. Lady Byron estaba decidida a casar a su hija con un noble cuyo título tuviese como mínimo un siglo de antigüedad.

La ceremonia le hizo mucha ilusión a Ada. A la joven que acababa de cumplir dieciocho años, como vemos en la Figura 2.8, que hasta entonces había vivido resguarda bajo las faldas de su madre, no le deslumbró ni el apuesto príncipe ni el lujo que rodeaban al monarca; le impresionaron los científicos que conoció días después. Ada se alegró sobre todo de conocer a Charles Babbage, que entonces tenía cuarenta y cuatro años.

Figura 2.8: Ada Byron.

[15]Nacida el 26 de diciembre de 1780 en Escocia, Mary fue otra mujer que brilló con luz propia; científica que llegó a dominar las matemáticas, la astronomía y otras ciencias, pero cuando tuvo cierta edad fue autodidacta y su familia (muy típico de la época) no aprobaba que fuera tan estudiosa. En 1804 se casó con el coronel Samuel Greig, que murió tres años más tarde. En 1812 se volvió a casar; esta vez con un inspector de hospitales, William Somerville. Éste estimulaba su interés por la ciencia y la animaba a que emprendiera nuevos proyectos. Cuando Ada y lady Byron la conocieron estaba apunto de convertirse en una de las figuras matemáticas más importantes del mundo. Su hijo Woronzow fue más adelante el abogado de Ada y su íntimo amigo.

[16]Nacido en 1786 y oriundo de Brighton, ciudad de la costa sur de Inglaterra. Fue médico, filántropo, director de manicomios y evangélico devoto. Hoy en día se le conoce como uno de los primeros defensores del cooperativismo, movimiento que fomentaba la creación de sociedades que suministraban a sus miembros artículos y servicios en condiciones beneficiosas.

Babbage, que se apasionaba con tanta facilidad como Ada, les describió a ella y a lady Byron la famosa «máquina diferencial». El trabajo de Babbage no interesaba únicamente a madre e hija. En 1832 ya era imperiosa la necesidad de una calculadora fiable: con el desarrollo y la difusión de la tecnología se hacían cada vez más comunes los errores en las tablas matemáticas. Si uno manejaba tablas logarítmicas inexactas para hacer un cálculo importante, era inevitable obtener un resultado incorrecto, y se hacía muy complicado encontrar el error.

Si a Annabella y a Ada les entusiasmó tanto conocer a Babbage fue porque, en diciembre de 1832, el científico, su principal colaborador técnico, Joseph Clement, y un equipo de operarios habían terminado de construir el único dispositivo equiparable en complejidad al telar de Jacquard: el prototipo de la calculadora que Babbage llamaba máquina diferencial, y a la que había dedicado veinte años de trabajo. El invento se convirtió en el principal tema de conversación en los círculos científicos de Londres, y Babbage disfrutaba hablando de él a las visitas y haciendo demostraciones públicas.

El lunes 17 de junio de 1833, apenas doce días después de haber conocido a Babbage, Ada y Lady Byron le visitaron en su casa, dónde les mostró la séptima parte de la máquina diferencial, aquello que había costado tanto trabajo y dinero construir. Ada supo ver más allá del maravilloso movimiento de las ruedas, y comprendió cómo Babbage había hecho posible enlazar el mundo abstracto de las matemáticas con un objeto mecánico. Fue entonces cuando Ada descubrió en Babbage a una persona afín, con la que compartía ante todo una enorme curiosidad intelectual.

Ada y Babbage no trabaron amistad enseguida, debido a la diferencia de edad entre ambos. El científico la cautivó. Madre e hija siguieron en contacto con él y la joven le vio varias veces mientras su madre le buscaba marido.

A finales de 1834, Ada no andaba pensando únicamente en la Máquina Analítica. El 10 de diciembre cumplió diecinueve años y lady Byron consideraba que ya era hora de que encontrara marido. Babbage era un candidato admisible. Había descubierto en Ada la pasión por las matemáticas, pero para su madre no era apto. Lady Byron estaba decidida a que su hija se casase con un noble, y lo ideal que poseyera un título como mínimo con un siglo de antigüedad. Así pues en la primavera de 1835, Ada conoció a William, lord King, que para entonces tenía treinta años. El aristócrata era un buen partido.

Su familia era, una de las familias influyentes desde el punto de vista político, social, intelectual y religioso; y el título de lord King se había creado en 1725, así que poseía la antigüedad requerida; aparte de tener gozar de varias propiedades importantes. Ada no era insensible al encanto de William, y el 28 de junio de 1835 le escribió una carta en la que le expresaba sus sentimientos. La pareja se casó el 8 de julio de 1835 y tuvieron tres hijos: el mayor y heredero, Byron nació el 12 de mayo de 1836; la niña, Annabella, el 22 de septiembre de 1837; y el segundo varón, Ralph, el 2 de julio de 1839. En 1837, cuando la princesa Victoria ascendió al trono, William pasó de barón a vizconde (gesto político a uno de sus ministros) y recibió el título de conde de Lovelace. Así, Ada pasó a firmar como Ada Lovelace.

La falta de ambición por parte su marido acabó por irritarla. Ada ansiaba un marido que hiciese grandes cosas y alcanzase renombre, y comprendiera, la acuciante necesidad que ella sentía de desarrollar su inteligencia. Pero William no era así. Ada dejó de lado sus deberes conyugales y se entregó de nuevo a las matemáticas. Estaba decidida a encontrar un mentor, alguien que le giara en su trabajo intelectual y que candidato mejor que Charles Babbage.

A raíz de esto, empezó una gran amistad y envíos de cartas entre Ada y Babbage. En esta correspondencia trataban temas puramente científicos, aunque es cierto que con el paso del tiempo, ambos se trataban de una manera más cariñosa. En una de las cartas que

Babbage le mandó a Ada, le anima a traducir al inglés el artículo que escribió Luigi Federico Menabrea sobre la Máquina Analítica, para una revista suiza *Bibliothèque Universelle de Genève*, en octubre de 1842.

Traducir este escrito tenía dos objetivos muy importantes para la joven Ada; el primero era dar a conocer el increíble trabajo que había realizado su amigo Babbage, y el segundo, ayudarla a cumplir su sueño de una vida intelectual que la elevase por encima de las exigencias de la maternidad, del cuidado de las casas y de un marido tan rico como inútil.

Ada no ignoraba la dificultad de la tarea, pero se consideraba perfectamente capaz, y enseguida se puso a trabajar con la energía que la caracterizaba. Su francés era excelente, y sus escritos tienen una fluidez, una claridad expositiva y una riqueza metafórica que recuerdan la prosa de su padre.

Con el tiempo fue ganando seguridad en sí misma y se acostumbró a imponer su voluntad, pero en el trabajo siempre fue muy humilde: se conformaba con estar en segundo plano con respecto a los investigadores varones.

A Babbage le sorprendió que la joven quisiera traducir el artículo que creó Menabrea, ya que poseía conocimientos suficientes como para escribir uno propio. De nada sirvió que él le dejara claro lo mucho que confiaba en su talento. Ada emprendió la traducción con la ayuda del científico. Lady Lovelace trabajó con sus propios materiales sobre la Máquina Analítica y pedía consejo a Babbage cuando lo creía necesario. Éste quedaba asombrado con las brillantes ideas que tenía la joven sobre la máquina. Durante la primavera de 1843, Babbage y Ada se vieron varias veces para hablar sobre la traducción del artículo, aunque Ada no comenzó a redactar sus notas hasta finales de mayo.

El artículo de Menabrea comienza explicando los antecedentes (máquina de Pascal, anteriormente mencionada en el contexto histórico) del nuevo invento de Babbage, y continua describiendo la importancia de la máquina: materializa «una idea extraordinariamente ambiciosa», ya que es capaz de realizar operaciones aritméticas y de análisis.

La traducción que realizó Ada consta de unas ocho mil palabras, no es un texto muy largo pero si denso debido a las fórmulas matemáticas que encontramos en él, cómo veremos en el siguiente capítulo.

Cuando las notas de Ada estaban casi listas para su publicación (septiembre de 1843), las cosas se complicaron. Convencido de que el gobierno le tenía enemistad y animadversión y de que por eso le habían negado varios puestos académicos a los que consideraba que tenía derecho, Babbage quería incluir, en el mismo número de la revista *Scientific Memoirs*, y con el título de «Sketch of the Analytical Engine invented by Charles Babbage, Esq.» donde se iban a publicar las notas, un escrito denunciando enérgicamente el trato injusto que recibía. Ada se negó rotundamente y le pidió al impresor que no incluyera la sátira de su amigo.

Después de 1843, el año más apasionante como científica, Ada y Babbage siguieron frecuentando y tratando con la élite intelectual de la Inglaterra victoriana: Charles Dikens[17], Michel Faraday[18] y muchos otros científicos y escritores de suma importancia.

En la década de 1840, Ada se volvió adicta a las carreras de caballos, perdió mucho dinero apostando a principios de la siguiente. En esta última época de su vida pasó apuros económicos continuos y su marido apenas le ayudó.

[17] Fue un destacado escritor y novelista inglés de la era victoriana.

[18] Fue un físico y químico británico que estudió el electromagnetismo y la electroquímica. Sus principales descubrimientos son la inducción electromagnética, el diamagnetismo y la electrólisis.

En el verano de 1851, la salud de lady Lovelace empeoró bruscamente. Siempre fue bastante frágil, pero desde mediados de la década anterior, había enfermado con frecuencia y padecido continuamente de agotamiento nervioso y debilidad general. Fue entonces cuando aparecieron los primeros síntomas del cáncer de útero, sangrando a menudo, aunque al principio sin dolor.

El cáncer la fue devorando lenta pero inexorablemente, y los dolores, al principio intermitentes, eran cada vez más intensos. A finales de 1851, nada invitaba ya al optimismo, pasaba casi todo el tiempo echada en el sofá.

A principios de 1852 entró en fase terminal, y pronto moriría en medio de dolores insoportables. Babbage estaba muy preocupado por su amiga, fue amigo suyo desde que la conoció en 1833 y no la abandonó en ningún momento. La visitó por última vez el 12 de agosto de 1852; aquel día tuvo una fuerte discusión con lady Byron y le prohibió volver a ver a su hija nunca más. La última voluntad de Ada fue ser sepultada al lado de su padre, algo no grato para lady Byron.

El día 30, Ada se quedó sin pulso unos minutos; el dolor se agudizó y empezó a delirar pidiéndole a lady Byron y William que rezaran por ella.

Así pues, murió el 27 de noviembre de 1852. A su lado se encontraban lady Byron y su esposo. Una semana después fue sepultada junto a su padre en la parroquia del pueblo de Hucknall, cerca de la abadía de Newstead.

Lady Byron no fue al entierro de su hija. Tampoco asistió Babbage, que seguramente juzgó inoportuna su presencia por haberse indispuesto con su madre. Así, la Ada que había conocido y quizá amado ya no existía.

CAPÍTULO 3
Análisis de las Notas de Ada Byron

Las siete notas, designadas con las letas de la A a la G, aparecen justo despúes del artículo que tradujo la joven y bajo el título de *Notas de la traductora* con referencias [15, 19]. Estas notas constan de unas veinte mil palabras, más del doble de la traducción. Ada dedica gran parte de su estudio a describir con lenguaje muy técnico cómo funcionaría la Máquina Analítica, pero también ofrece una serie de observaciones generales que dan una clara idea de su aportación teórica.

En estas notas Ada distingue entre datos y procesamiento. Esta distinción, que hoy en día damos por sentada, era sin embargo revolucionaria en aquel tiempo.

Ada aspiró a crear la informática y separarla de las matemáticas, que al contrario que Babbage, la joven reparó en las aplicaciones prácticas de la máquina.

Cómo ya mencionamos anteriormente, Ada creó sus propias notas sobre la Máquina Analítica de Babbage al traducir el artículo creado por Menabrea a raíz de la conferencia que impartió el matemático en Italia; Menabrea se centra principalmente en los fundamentos matemáticos de la Máquina Analítica y no en sus subyacentes operacines mecánicas, concepto que Ada si aborda en sus notas; en este capítulo trataremos cada una de ellas y destacaremos los conceptos más importantes mediante una detallada descripción.

Las famosas Notas de Ada, consisten en siete notas escritas de su puño y letra en las cuales, nos relata en qué consisten y nos introducen términos que hoy en día son bien conocidos en el mundo de la informática y en concreto en programación.

También hemos de tener en cuenta que Ada hace una constante alusión tanto a la Máquina Diferencial como a la Máquina Analítica e incluso en alguna de sus notas hace una comparación entre ellas, para poder entender mejor el funcionamiento y alcance de la última.

Teniendo en cuenta estas consideraciones, ya podemos empezar a estudiar sus notas.

3.1 Nota A

En esta nota se describen los fundamentos y principios de la Máquina Analítica, se menciona por primera vez el telar de Jacquard y su relación con la misma y nos introduce el concepto de tarjetas perforadas. Empecemos.

La función integral para la que fue concebida la Máquina Diferencial es la siguiente:

$$\Delta^7 ux = 0$$

El objetivo en el que la Máquina Diferencial ha sido especialmente intencionada y adaptada es para llevar a cabo cálculos de coputación náutica y tablas astronómicas. La integral desarrollada es:

$$u_z = a + bx + cx^2 + dx^3 + ex^4 + fx^5 + gx^6$$

cuyas constamtes son a, b, c, ... y son representadas en las siete columnas de discos que la forman la Máquina Diferencial. Esta máquina es capaz de tabular de manera precisa e ilimitada, todas las series en las que el término general esté integrado en la fórmula. El resto de series son resueltas por el Método de las Diferencias Finitas[1].

La Máquina Analítica, por el contrario, no se limita únicamente a aproximar y tabular los resultados de una función particular, sino que es apta para desarrolar y tabular cualquier función. Así pues, la máquina puede ser descrita como un ser material que calcula funciones indefinidad de cualquier grado de generalidad y complejidad, cómo serían estos tipos:

$$F(x, y, z, \log x, \sin y, x^p)$$

La Máquina Analítica consta de un estado neutro o cero[2] en el que la máquina se encuentra lista para recibir la información a través de las tarjetas perforadas[3], disminuyendo el número de errores, el tiempo computacional y la mano de obra[4]; es en este momento cuando Ada relata su famosa frase *La Máquina Analítica teje patrones algebraicos tal como el telar de Jacquard es capaz de tejer flores y hojas*.

De este modo, la protagonista hace una clara distinción entre la Máquina Diferencial y la Analítica; la primera se basa en la síntesis y la segunda es capaz tanto de analizar como de sintetizar.

En esta nota Ada nos habla también sobre los datos de entrada y de salida; diciéndonos que cabe la posibilidad que en alguna operación a través de los datos de entrada obtengamos la misma salida escribiéndolos en una misma columna o conjunto de columnas para poder volver a utilizarlos (recordemos que la Máquina Analítica consta de

[1] El método consiste en una aproximación de las derivadas parciales por expresiones algebraicas con los valores de la variable dependiente en un limitado número de puntos seleccionados. Como resultado de la aproximación, la ecuación diferencial parcial que describe el problema es reemplazada por un número finito de ecuaciones algebraicas, en términos de los valores de la variable dependiente en puntos seleccionados. El valor de los puntos seleccionados se convierten en las incógnitas. El sistema de ecuaciones algebraicas debe ser resuelto y puede llevar un número largo de operaciones aritméticas.

[2] Primer término informático introducido por Ada, lo que hoy en día conocemos en informática como estado inicial.

[3] Característica distintiva de la Máquina Analítica y basada en el telar de Jacquard como ya vimos anteriormente, y que funcionan siguiendo un cierto orden como veremos en la Nota C y G.

[4] Hemos de tener en cuenta que pese a encontrarnos en pleno auge revolucionario diversos trabajos aún eran manuales.

columnas puestas una detrás de la otra de manera perpendicular dentro de un prisma rectangular).[5]

También nos describe a la máqunia como la "*personificación de la ciencia de las operaciones*" ya que ésta realiza los cálculos mediante las operaciones de aritmética básica a diferencia de la Máquina Diferencial que sólo operaba mediante sumas.

Ada definió dos principios sobre la Máquina:

- Las magnitudes numéricas son los resultados de las operaciones realizadas en los datos numéricos, es decir, los resultados que obtenemos son fruto de operar en la máquina con los datos directamente.

- Los resultados simbólicos[6] que se adjuntan a esos resultados numéricos, son resultados simbólicos de las consecuencias necesarias y lógicas de las operaciones realizadas en los datos simbólicos, que son resultados numéricos cuando los datos son numéricos, es decir, todas las operaciones que se realizan sobre los datos nos dan como consecuencia, los resultados.

Lo que Ada trata de decirnos con estos dos principios no es más que el modo de calcular que tiene la máquina; modo que veremos más adelante tanto en el apartado que corresponde a la **Nota G** como en el capítulo que corresponde a los **Números de Bernoulli**.

[5] Reutilización de código, otro término introducido por Ada, conceptos muy avanzados para la época en la que se encontraba, de hecho esto pudo llegar a ocasionarle algún problema, tratándose de una mujer con ideas tan avanzadas para la época.

[6] Debemos entender por resultados símbolicos a los signos y símbolos que encontramos en las operaciones como pueden ser $+, -, \times, \div, =$ entre otros.

3.2 Nota B

En ella se describe a la Máquina Analítica como tal, es decir, las partes que la forman y el funcionamiento de cada una de ellas.

Como ya sabemos la Máquina Analítica está formada por un prisma rectangular y dentro de éste se encuentran las columnas. A esta parte Ada la denomina «almacén»[7]; estas columnas están compuestas por discos inscritos del cero al nueve.

Además, las columnas son giratorias, es decir, pueden girar horizontalmente de manera que podríamos colocar el dígito que deseemos ante nuestros ojos. El disco que se encuentra más abajo pertenece a las unidades, el siguiente de arriba a las decenas, el siguiente a las centenas y así sucesivamente hasta llegar al último disco (de tal manera que podemos formar números considerables, ya que si se hubier llegado a construir la máquina podría poseer aproximadamente unas doscientas columnas).

Si tuviéramos que hacer una representación en un papel de las columnas, probablemente las visualizaríamos como se muestran en la Figura 3.1:

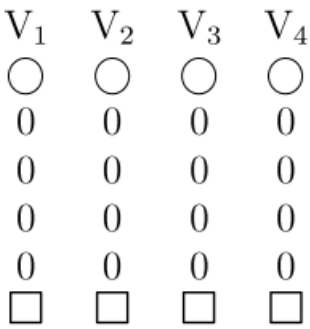

Figura 3.1: Notación.

dónde V hace referencia a las variables, ya que los valores de dichas columnas van a cambiar constantemente. Cada círculo contendrá o bien el signo + o el signo -, de manera que podrán ser sustituidos el uno por el otro indistintamente según si el número que queremos representar es positivo o negativo. Los ceros que se encuentran debajo de cada círculo, representan a los diferentes discos.

Dado que cada disco puede representar cualquier dígito y cada círculo cualquier signo, los discos de cada columna pueden ajustarse de modo que cualquier número positivo o negativo puede representarse dentro de los límites de la Máquina.

Cada uno de los cuadrados por debajo de los ceros está destinado a la inscripción de cualquier símbolo general o combinación de símbolos que deseemos, de tal manera que el número representado en la columna inmediatamente anterior es el valor númerico de ese símbolo, o combinación de símbolos.

Ada hace referencia a esta explicación con el siguiente ejemplo:

Tenemos tres cantidades, a, n, x, suponiendo que cada una de ellas tiene el siguiente valor: $a = 5$, $n = 7$, $x = 98$, quedándo de tal manera como se muestra en la Figura 3.2:

[7] Ya nombrado así por Menabrea en su artículo[19].

V_1	V_2	V_3	V_4	&c.
+	+	+	+	
0	0	0	0	
0	0	0	0	&c.
0	0	9	0	
5	7	8	0	&c.
a	n	x		

Figura 3.2: Cantidades.

Como ya hemos introducido las cantidades con sus correspondientes símbolos, ahora debemos combinarlos y escribir, en la siguiente columna a la derecha, el resultado de nuestra función. Podemos introducir todo tipo de funciones, como por ejemplo:

ax^n, $x^a n$, $a \cdot n \cdot x$, $\frac{a}{n}x$, $a + n + x$

Para exponer la explicación Ada nos proporciona, usaremos la primera como se muestra en la Figura 3.3, pero podíamos usar cualquier otra indistintamente.

V_1	V_2	V_3	V_4
+	+	+	+
0	0	0	0
0	0	0	0
0	0	9	0
5	7	8	0
a	n	x	ax^n

ax^n

Figura 3.3: Cálculo.

El primer paso ha de ser el cálculo de las seis multiplicaciones para calcular x^n, en nuestro caso 98^7. Una vez esté calculado esto, llevaremos a cabo la multiplicación de $a \times x^n$, en nuestro caso 5×98^7, teniéndo así un conjunto de siete multiplicaciones para llegar al resultado final

$$(\times, \times, \times, \times, \times, \times, \times)$$

.

Para determinar el valor de ax^n se lleva a cabo a través de operaciones homogéneas, es decir, se van calculando en pequeñas operaciones (como ya hemos visto, por un lado seis operaciones y por el otro la séptima y última operación) en las fases sucesivas a la anterior.

Este proceso lo podemos llevar a cabo gracias a las tarjetas perforadas, en las que cada operación está distribuida de forma que se adapte a cada función en particular según en la fase en la que se encuentre.

Aquí concluye la Nota B, veámos ahora la siguiente nota.

3.3 Nota C

Tenemos dos conceptos protagonistas en esta nota; el primero es la reutilización de código que tanto usamos hoy en día y que nos es de gran ayuda en ciertas ocasiones, y el segundo es el famoso telar de Jacquard y sus tarjetas perforadas.

El funcionamiento clave de la Máquina Analítica son las tarjetas perforadas, ya que gracias a ellas y a las columnas por las que está constituida, hacen posible que podamos operar y manipular los datos.

Ada va más allá de este concepto y estudia el modo de aplicación de las tarjetas, y nos enuncia que el modo de empleo que se está llevando a cabo en la técnica de tejer no es lo suficientemente efectiva para todas las simplificaciones que eran necesarias para alcanzar los procesos que ha de desarrollar; ya que ha diferencia de la Máquina Analítica, el telar de Jacquard tenia un funcionamiento muy básico.

El objetivo que Ada pretende alcanzar es poder usar cualquier tarjeta en particular o conjunto de tarjetas un número sucesivo de veces en la solución de un problema, es decir, poder reutilizar código, teniendo en cuenta que el código al que hacemos alusión en este contexto se encuentra contenido en las tarjetas.

Para llevar a cabo lo arriba mencionado, el prisma donde se introducen las tarjetas ha de girar hacia atrás en lugar de hacia delante, hasta que ninguna de las tarjetas en particular o conjunto de tarjetas, pueda traerse de nuevo hasta la posición que ocupaba justo antes de que se utilizara, es decir, las columnas han de girar tanto hacia delante como hacia atrás, de tal forma que logremos posicionar la tarjeta en su estado inicial. Una vez hecho esto, el prisma girará hacia delante y traerá la tarjeta o conjunto de tarjetas una segunda vez. Este proceso puede obviamente ser repetido cualquier número de veces.

3.4 Nota D

En estos momentos nos encontramos en el ecuador de sus notas. La Nota D, es el precedente de la nota más importante de Ada, la **Nota G**.

En esta nota se describen los tipos de variables que usamos para operar en la Máquina Analítica así como las series de sustituciones que debemos realizar.

Para poder llevar a cabo una operación dentro de la Máquina Analítica tenemos que tener en cuenta que existen tres tipos de variables, empleándose cada una de ellas para una acción en concreto.

Los diferentes tipos de variables que se utilizan son los siguientes:

- Variables de Datos (VD), estas variables se emplean para escribir los datos con los que operaremos en la máquina.

- Variables de Resultados (VR), se trata de aquellas variables en las que guardaremos el resultado una vez que los cálculos ya estén finalizados.

- Variables Temporales (VT), son las variables donde realizaremos los cálculos y que una vez que hayamos obtenido el resultado pasaremos a las Variables Resultado, careciendo así de símbolos. También podrían ser denominadas Variables de Datos Secundarios (debido a su naturaleza), aunque Ada prefiere llamarlas Variables Temporales.

Las columnas que reciben este último tipo se denominan "de trabajo de variables" ya que en ellas se unen los datos para formar el resultado. Las VR en alguna ocasión pueden participar en las columnas mencionadas anteriormente y se reutilicen una y otra vez para escribir los resultados de las operaciones. Las Variables Datos también pueden pasar en alguna ocasión a Variables Resultado si es necesario.

A la hora de operar hemos de tener en cuenta las siguientes consideraciones descritas por Ada, no mencionadas hasta el momento: Cuando el valor de cualquier variable se pone en uso puede volver a su estado inical después de haberse usado, puede hacerse cero o puede llegar a ser una Variable Resultado.

Llegaremos a cero en una variable cuando no necesitemos más su uso, es decir, no apliquemos más la recurrencia[8]para esta variable; por lo tanto al final de cada cálculo las columnas deben de ser cero excepto donde se encuentren las VR, que contendrán, como su propio nombre indica, los resultados de las operaciones.

Las diferentes variables se organizan dentro de las tarjetas perforadas según dos variedades, una de ellas es que las variables se pueden reutilizar de nuevo volviendo a su estado inicial y la otra es cuando sustituyendo dicha variable se hace cero. Podemos distinguir estas variedades cuando vamos a usar las tarjetas en la Máquina, ya que están conectadas por medio de cables a la columna con la que se pretende operar.

En esta nota Ada también nos explica como van cambiando las variables conforme se van usando; una variable cuyo índice es cero significa que el resultado es cero, si el índice es igual a uno, esto nos indica que posee un valor, cuando es igual a dos significa que el valor anterior ha cambiado de tal manera que conforme vayan cambiando los valores el índice será de $n + 1^V$ y volverá a cero cuándo el valor sea nulo. Por lo tanto, los índices altos nos indican alteración en el valor de una variable y el índice nulo se trata de índice local, como podemos ver en la imagen.

Estas son las tres sustituciones sucesivas para cada una de las ecuaciones. Podemos observar que cada

[8]Proceso que se basa en sí mismo.

$$
\begin{array}{lllll}
\text{(1.)} & \text{(2.)} & \text{(3.)} & \text{(4.)} & \\
{}^1\mathrm{V}_{16} = \frac{{}^1\mathrm{V}_{14}}{{}^1\mathrm{V}_{12}} = \frac{{}^1\mathrm{V}_{10} - {}^1\mathrm{V}_{11}}{{}^1\mathrm{V}_6 - {}^1\mathrm{V}_7} = \frac{{}^1\mathrm{V}_0 \cdot {}^1\mathrm{V}_5 - {}^1\mathrm{V}_2 \cdot {}^1\mathrm{V}_3}{{}^1\mathrm{V}_0 \cdot {}^1\mathrm{V}_4 - {}^1\mathrm{V}_3 \cdot {}^1\mathrm{V}_1} = \frac{d'm - dm'}{mn' - m'n} \\
\text{(1.)} & \text{(2.)} & \text{(3.)} & \text{(4.)} & \\
{}^1\mathrm{V}_{15} = \frac{{}^1\mathrm{V}_{13}}{{}^1\mathrm{V}_{12}} = \frac{{}^1\mathrm{V}_8 - {}^1\mathrm{V}_9}{{}^1\mathrm{V}_6 - {}^1\mathrm{V}_7} = \frac{{}^1\mathrm{V}_2 \cdot {}^1\mathrm{V}_4 - {}^1\mathrm{V}_5 \cdot {}^1\mathrm{V}_1}{{}^1\mathrm{V}_0 \cdot {}^1\mathrm{V}_4 - {}^1\mathrm{V}_3 \cdot {}^1\mathrm{V}_1} = \frac{dn' - d'n}{mn' - m'n}
\end{array}
$$

Figura 3.4: Series de sustituciones.

3.5 Nota E

En esta nota Ada nos explica la manera que tiene la Máquina Analítica de calcular funciones trigonométricas y nos introduce también el término de *ciclo*[9].

El ejemplo que usaremos para explicar esta nota es una función trigonométrica que contiene variables, tal y como sigue:

$$A + A_1 \cos\theta + A_2 \cos 2\theta + A_3 \cos 3\theta + \cdots$$

$$B + B_1 \cos\theta$$

Multiplicando las dos fórmulas anteriores obtenemos la fórmula general, representándola de la siguiente manera:

$$C_0 + C_1 \cos\theta + C_2 \cos 2\theta + C_3 \cos 3\theta + \cdots \tag{3.1}$$

Se ha elegido esta serie trigonómetrica porque es apropiada para ilustar los procesos que realiza la Máquina Analítica. Poseemos tres maneras para deducir los valores de cualquier fórmula analítica dentro de la máquina.

El primero, encontrar el valor numérico de toda la fórmula,es decir, resolver la función sin tener en cuenta los procesos que se han llevado a cabo para su resolución. Estos valores son de naturaleza estrictamente aritmética.

El segundo, calculamos el valor numérico de cada término de la fórmula, teniéndo así en cada columna los diferentes términos a calcular.

El tercero y último, se puede calcular el valor numérico de varias subdivisiones de cada término y mantener los resultados separados. Para llevar a cabo estas operaciones hemos de tener en la máquina dos columnas para cada término, es decir, una columna para la variable y otra para los coeficientes.

En ocasiones puede ser deseable tener separados y distribuir los valores numéricos de las diferentes partes de una fórmula; y el poder efectuar tales ditribuciones a cualquier medida es esencial para el carácter algebraico sobre el que se basa la Máquina Analítica. Ada nos detalla que la máquina puede organizar y combinar sus cantidades numéricas exactamente como si fueran letras o cualquier otro símbolo matemático; y podría desarrollar tres conjuntos de resultados de forma simultánea: *resultados simbólicos* (vistos en las notas A y B), *resultados numéricos* y *resultados algebraicos en notación literal*. El objetivo de la máquina es dar la máxima eficiencia práctica posible para los recursos de interpretaciones numéricas, al tiempo que utiliza los procesos y combinaciones de este último.

[9]En lenguaje de programación lo denominamos Buble, se trata de una sentencia que se realiza un determinado número de veces a un trozo de código en concreto, hasta que la condición asignada a dicho ciclo deje de cumplirse. Generalmente, el bucle se utiliza para ahorrar tiempo y procesos, ya que no hace falta escribir el mismo código varias veces

Para resolver la serie trigonométrica sólo utilizarmos los cuatro primeros términos ($A + A_1 \cos\theta + \cdots$), ya que con esto será suficiente para mostrar el método. Vamos a obtener por separado el valor numérico de cada coeficiente $C_0, C_1, \cdots$ de la fórmula general que mencionamos anteriormente. Así, la multiplicacion directa de los dos factores da como resultado:

$$BA + BA_1 \cos\theta + BA_2 \cos 2\theta + BA_3 \cos 3\theta + \cdots$$

$$B_1 A \cos\theta + B_1 A_1 \cos\theta \cdot \cos\theta + B_1 A_2 \cos 2\theta \cdot \cos\theta + B_1 A \cos 3\theta \cdot \cos\theta \tag{3.2}$$

En la Máquina Analítica quedaría de la siguiente manera como vemos en la Figura 3.5:

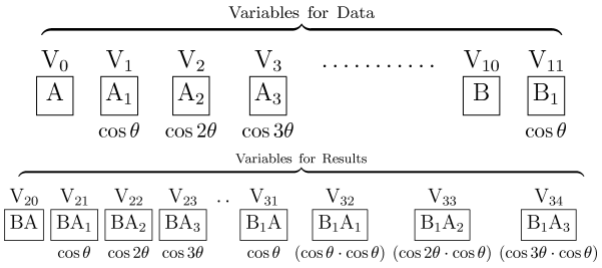

Figura 3.5: Almacenamiento diferentes variables.

La variable que pertenece a cada coeficiente está escrito debajo de el, como podemos observar el la imágen anterior. La única reducción que podríamos hacer sería la sumas de V_{21} a V_{31} (en cuyo caso $B_1 A$ debe ser borrado de V_{31}) teniendo que calcular entonces:

Primeras Operaciones	Segundas Operaciones	Última Operación
$V_{10} \times V_0 = V_{20}$	$V_{11} \times V_0 = V_{31}$	$V_{21} \times V_{31} = V_{21}^2$
$V_{10} \times V_1 = V_{21}$	$V_{11} \times V_1 = V_{32}$	V_{31} pasa a ser $= 0$
$V_{10} \times V_2 = V_{22}$	$V_{11} \times V_2 = V_{33}$	
$V_{10} \times V_3 = V_{23}$	$V_{11} \times V_3 = V_{34}$	

Tabla 3.1: Series de operaciones a resolver.

No vamos a entrar en más detalle de cada paso de los procesos como en los ejemplos que hemos dado en la Nota D o como daremos en la Nota G, ya que no es necesario hacerlo, porque el lector recuerda el significado y el uso de los índices superior e inferior, explicados anteriormente.

Para poder continuar con nuestro ejemplo debemos saber:

$$\cos n\theta \times \cos\theta = \frac{1}{2}\cos\overline{n+1}\theta + \frac{1}{2}\overline{n-1}\theta \tag{3.3}$$

En consecuencia, si hacemos las sustituciones apropiadas en la fórmula (3.2) nos dará como resultado:

BA	$+\mathrm{BA_1} \cdot \cos\theta$ $+\mathrm{B_1A} \cdot \cos\theta$	$+\mathrm{BA_2} \cdot \cos 2\theta$	$+\mathrm{BA_3} \cdot \cos 3\theta$	
$+\frac{1}{2}\mathrm{B_1A_1}$	$+\frac{1}{2}\mathrm{B_1A_2} \cdot \cos\theta$	$+\frac{1}{2}\mathrm{B_1A_1} \cdot \cos 2\theta$	$+\frac{1}{2}\mathrm{B_1A_2} \cdot \cos 3\theta$	$+\frac{1}{2}\mathrm{B_1A_3} \cdot \cos 4\theta$
$\mathrm{C_0}$	$\mathrm{C_1}$	$\mathrm{C_2}$	$\mathrm{C_3}$	$\mathrm{C_4}$

Figura 3.6: Sustituciones en la Fórmula (3.2).

Representando cada uno de los coeficientes en V_{20}, V_{21}, V_{22}, V_{23} y V_{24} respectivamente.

En la cuarta serie de operaciones introducimos los coeficientes en las variables V_{32}, V_{33}, $\cdots$ (siempre comenzando con V_{32}) en las tarjetas adecuadas diviendo en dos nuestra fórmula, cómo podemos observar en la Tabla 3.2:

Cuarta Operación

$$V_{32}^1 \div 2 + V_{22}^1 = V_{22}^2 = BA_2 + \tfrac{1}{2}B_1A_1$$
$$V_{32}^1 \div 2 + V_{20}^1 = V_{20}^2 = BA + \tfrac{1}{2}B_1A_1 = C_0$$
$$V_{33}^1 \div 2 + V_{23}^1 = V_{23}^2 = BA_3 + \tfrac{1}{2}B_1A_2 = C_3$$
$$V_{33}^1 \div 2 + V_{21}^2 = V_{21}^3 = BA_1 + B_1A + \tfrac{1}{2}B_1A_2 = C_1$$
$$V_{34}^1 \div 2 + V_{24}^0 = V_{24}^1 = \tfrac{1}{2}B_1A_3 = C_4$$
$$V_{34}^1 \div 2 + V_{22}^2 = V_{22}^3 = BA_2 + \tfrac{1}{2}B_1A_1 + \tfrac{1}{2}B_1A_3$$

Tabla 3.2: Cuarta serie de operaciones.

El cálculo de los coeficientes C_0, C_1, C_2 $\cdots$ estaría ahora completamente guardado en V_{32}, V_{33}, $\cdots$ teniéndo así el resultado del cálculo en las Variables Resultado.

Vamos a profundizar más en el ejemplo y vamos a calcular ahora el valor de cada término completo de la fórmula (3.1), es decir, para coeficiente y variable; debemos tener en cuanta que para cada $(n+1)$ -ésimo término terndríamos $C_n \cdot \cos\theta$

Para llevar a cabo este cálculo, debemos de introducir las variables en otro conjunto de columnas (V_{40}, V_{41}, V_{42}, $\cdots$.), y multiplicarlas por V_{21}, V_{22}, $\cdots$, como visualizamos en la Tabla 3.3:

Quinta Operación

$$V_{20}^3 \times V_{40} = V_{40}$$
$$V_{21}^3 \times V_{41} = V_{41}$$
$$V_{22}^3 \times V_{42} = V_{42}$$
$$V_{23}^3 \times V_{43} = V_{43}$$
$$V_{24}^1 \times V_{44} = V_{44}$$

Tabla 3.3: Última operación a realizar.

V_{40} se multiplicará por el coeficiente de V_{20}, el cual no contiene nada, y su valor será entonces de $\cos 0\theta = 1$. En el momento que acaba la quinta y última serie de operaciones las variables V_{40}, V_{41}, $\cdots$ se escribirán en Variables de Trabajo para más adelante ser destinatarias de los resultados finales.

Ya hemos visto que la Máquina calcula (1) de manera directa si conocemos la fórmula del término general. Los dos primeros términos son:

$$(BA + \frac{1}{2}B_1A_1) + \overline{(BA_1 + B_1A + \frac{1}{2}B_1A_2 \cdot \cos\theta)} \tag{3.4}$$

y el término general:

$$BA_n + \frac{1}{2}B_1 \cdot \overline{A_{n-1} + A_{n+2}}) \cos n\theta \tag{3.5}$$

dónde $(n+1)$ -ésimo término. La Máquina calculará los dos primeros términos a través de un conjunto separado de Tarjetas de Operaciones y necesitará otro conjunto de tarjetas para el tercer término, donde se calculara en términos infinitos (requeriendo de ciertas variables y nuevas tarjetas para cada término y poder colocar las operaciones a realizar en las columnas adecuadas).

Por tanto donde haya un término general, habrá un grupo recurrente de operaciones como ocurre en el ejemplo anterior. Ada a este grupo recurrente de operaciones lo denomina ciclo; y concluye así *"es igualmente un ciclo aunque sólo se repita dos veces o un número indefinido de veces"*. Puede ocurrir que en algun caso hayemos un grupo recurrente de varios ciclos, es decir, un ciclo de ciclos.

Veámoslo con un ejemplo:

Supongamos que queremos dividir una serie de una serie:

$$\frac{a + bx + cx^2 + \cdots}{a' + b'x + c'x^2 + \cdots'}$$

en la que se requiere desarrollar el dividendo y el divisor en sucesivas potencias de x para obtener la solución. Si ponemos el primer cociente en función de p, se completará con las siguientes operaciones:

$$((\div), p(\times, -))$$

o su equivalencia en números

$$((1), p(2, 3))$$

el segundo cociente se completará con un conjunto de operaciones similares que actuarán sobre el resto obtenido en el primer set, en lugar de aplicarlo en el dividendo inicial. De manera que el número total de procesos que se ha aplicado es:

$$2((\div), p(\times, -))$$

o en valores numéricos

$$2(1), p(2, 3)$$

es decir, un ciclo que incluye otro ciclo, o *ciclo de segundo orden*. Las operaciones para la división completa, suponiendo que proponemos para obtener n términos de la serie que constituye el cociente, serán:

$$n((\div), p(\times, -))$$

o su equivalencia en números

$$n(1), p(2, 3)$$

El número n de la fórmula anterior es siempre el de la serie de términos que pretendemos obtener y el coeficiente n-ésimo es el coeficiente de la $(n-1)$-ésima potencia de x. Es posible que en algún momento nos encontremos ante una serie que implique ciclos de ciclos de ciclos a una extensión indefinida. El desarrollo algebraico en una serie de la función de n-ésimo de las sigueintes funciones es de esta naturaleza:

$$\Phi(a, b, c, \cdots, x)$$

siéndo x la variable.

Por lo tanto en esta nota hemos visto tres formas de deducir valores:

- Valores estrictamente aritméticos.

- Valores numéricos colectivos de un fórmula o serie.

- Valores numéricos de varias subdivisiones de cada término,es decir, calcular los coeficientes.

3.6 Nota F

Nos encontramos en la recta final, su penúltima nota. En ésta Ada nos habla sobre el telar de Jacquard y sus tarjetas perforadas así como la reutilización de código a través de las mismas.

Gracias al uso repetitivo de las tarjetas aludido por Menabrea y explicado ampliamente en la **Nota C** se reduce ampliamente el uso necesario de estas tarjetas; es obvio que esta mejora es especialmente aplicable donde se producen ciclos en las operaciones matemáticas y en la preparación de los cálculos que efectuará la Máquina, es deseable para organizar el orden y la combinación de procesos con el fin de obtenerlos y aprobechar las ventajas mecánicas al máximo.

Como un mero ejemplo de la medida en que los sitemas combinados de ciclos y de respaldo pueden disminuir el número de tarjetas requeridas; supongamos que se requiere eliminar nueve variables de diez ecuaciones simples de la forma:

Ecuaciones
$ax_0 + bx_1 + cx_2 + dx_3 + \cdots = p$
$a^1x_0 + b^1x_1 + c^1x_2 + d^1x_3 + \cdots = p'$
$a^2x_0 + b^2x_1 + c^2x_2 + d^2x_3 + \cdots = p''$
$\cdots + \cdots + \cdots = \cdots$

Tabla 3.4: Ecuaciones a resolver.

El primer paso sería la eliminación de la primera x_0 (cantidad desconocida entre las dos primeras ecuaciones); esto se obtiene por la forma:

$$(a^1a - aa^1)x_0 + (a^1b - ab^1)x_1 + (a^1c - ac^1)x_2 + (a^1d - ad^1)x_3 + \cdots = a^1p - ap^1$$

con un total de $10(\times, \times, -)$ necesarios para su resolución. El segundo paso sería la eliminación de x_0 entre la segunda y la tercera ecuación, con el mismo número de operaciones requeridas; teniéndo un total de $10(\times, \times, -), 10(\times, \times, -) = 20(\times, \times, -)$ operaciones requeridas. Continuando de la misma manera, el número total de operaciones para la eliminación completa de X_0 entre todos los pares sucesivos de ecuaciones sería:

$$9 \cdot 10(\times, \times, -) = 90(\times, \times, -)$$

Nos quedaremos así con nueve ecuaciones simples de nueve variables, eliminado ahora la variable x_1 requiriendo un número de operaciones de

$$8 \cdot 9(\times, \times, -) = 72(\times, \times, -)$$

Ahora tendríamos ocho ecuaciones con ocho variables en las que eliminaremos x_2 con un total de operaciones de:

$$7 \cdot 8(\times, \times, -) = 56(\times, \times, -)$$

y así sucesivamente hasta llegar a eliminar todas las variables. El total de operaciones necesarias para eliminar todas las variables será:

$$9\cdot 10 + 8\cdot 9 + 7\cdot 8 + \cdot 7 + 5\cdot 6 + 4\cdot 5 + 3\cdot 4 + 2\cdot 3 + 1\cdot 2 = 330$$

que se llevarán a cabo con tres tarjetas de operación en vez de una tarjeta por operación.

Si tomamos n ecuaciones que contengan $n-1$ variables, siendo n un número ilimitado en magnitud, el caso se vuelve más evidente, ya que las mismas tres tarjetas podrán usarse para un gran número (miles o millones por ejemplo), en vez de una tarjeta por operación.

3.7 Nota G

Todas las notas de Ada, son importantes, pero ésta es la principal, ya que en ella nos relata las operaciones necesarias que debemos realizar para calcular los famosos números de Bernulli, a través un programa para la Máquina Analítica, creando así, el primer programa informático de la historia.

Antes de estudiar en profundidad esta nota debemos recordad que Ada, en múltiples ocasiones, deja muy claro que la Máquina Analítica no crea de la nada los datos, sino que nosotros somos los que introducimos y pedimos que realice ciertas operaciones, no tiene el poder de anticipar las relaciones analíticas o verdades.

Ahora vamos a describir los principales elementos con los que trabaja la máquina:

1.- Realiza las cuatro operaciones aritméticas simples sobre cualquier número.

2.- No existe un límite con los números ni variables que se pueden emplear.

3.- Se pueden combinar estos números y cantidades de manera aritmética, en las relaciones ilimitadas en cuanto a variedad, extensión o complejidad.

4.- Utiliza signos algebraicos de acuerdo con sus leyes propias y desarrolla las consecuencias lógicas de estas leyes.

5.- Se puede sustituir arbitrareamente cualquier fórmula para cualquier otra; borrando la primera de las columnas en las que está presente, y hacer que el segundo aparezca en su lugar.

6.- Se puede prever valores singulares. Su poder de hacer esto se menciona ya en la traducción que realizó Menabrea, donde se pasan valores a través de cero e infinito.

La máquina puede efectuar estos procesos de dos formas distintas:

- Podemos pedir que, por medio de la operación y de las tarjetas de variables, ir a través de las diferentes etapas por las que el límite requerido puede ser resuelto por cualquier función que esté bajo esta consideración.

- Se puede efectuar la integración por sustitución directa, como ya vimos en la **Nota B** que cualquier conjunto de columnas en las que se inscriben los números, representa meramente una función general de las diversas cantidades de términos que forman la integral o es el propio coeficiente.

Por ejemplo, para ax^n introduciríamos los valores y quedaría como se muestra en la Figura 3.7,

Teniendo en V_3 la combinación de cada uno de los términos, aunque podríamos ordenarlos en función de anx^{n-1}, visualizados en la Figura 3.8.

Del mismo modo podriamos tener $\frac{a}{n+1}x^{n+1}$, la integral de ax^n.

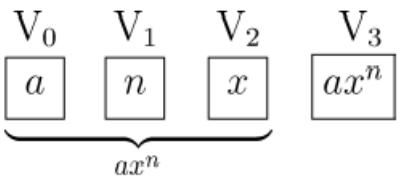

Figura 3.7: Introducción de valores.

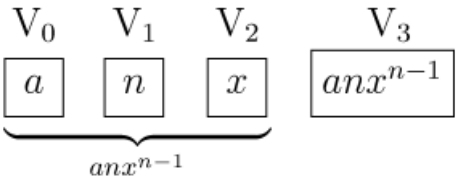

Figura 3.8: Cálculo.

Un ejemplo interesante sería de la forma de

$$\int \frac{x^n dx}{\sqrt{a^2 - x^2}}$$

o cualquier otro caso de integración por reducciones sucesivas, dónde una integral que contiene una operación repite n veces puede hacerse depender de otro que contiene el mismo $n-1$ o $n-2$ veces, y así sucesivamente hasta la reducción continua hasta que llegemos a la forma general.

Vamos a terminar estas notas mediante el seguimiento en detalle de los pasos a través de los cuales la Máquina puede calcular los números de Bernoulli. La manera más sencilla de calcular estos números sería a partir de la expresión directa de:

$$\frac{x}{e^x - 1} = \frac{1}{1 + \frac{x}{2} + \frac{x^2}{2\cdot 3} + \frac{x^3}{2\cdot 3\cdot 4} + \cdots} \tag{3.6}$$

que es de hecho, un caso particular de

$$\frac{a + bx + cx^2 + \cdots}{a' + b'x + c'x^2 + \cdots}$$

mencionado ya en la **Nota E** y podríamos calcularlas de forma conocida:

$$B_{2n-1} = 2\cdot \frac{1\cdot 2\cdot 3\cdots 2n}{(2\pi)^{2n}} \cdot \{1 + \frac{1}{2^{2n}} + \frac{1}{3^{2n}} \cdots\} \tag{3.7}$$

Ya hemos visto que hay diferentes formas de llevar a cabo el cálculo, pero para nuestro estuio vamos a elguir esta:

$$\frac{x}{e^x - 1} = 1 - \frac{x}{2} + B_1 \frac{x^2}{2} + B_3 \frac{x^4}{2\cdot 3\cdot 4} + B_5 \frac{x^6}{2\cdot 3\cdot 4\cdot 5\cdot 6} + \cdots \tag{3.8}$$

donde B_1, B_3, B_5 son los Números de Bernoulli. Si ampliamos el denominador de las primeras potencias de x, y luego dividimos el numerador y denominador por x, y derivamos, obtenemos

$$1 = (1 - \frac{x}{2} + B_1 \frac{x^2}{2} + B_3 \frac{x^4}{2\cdot 3\cdot 4} + \cdots)(1 + \frac{x}{2} + B_1 \frac{x^2}{2} + B_3 \frac{x^4}{2\cdot 3\cdot 4} + \cdots) \tag{3.9}$$

Si se lleva a cabo esta multiplicación tendremos una serie de la forma general:

$$1 + D_1x + D_2x^2 + D_3x^3 + \cdots \tag{3.10}$$

en la que vemos, en primer lugar, que todos los coeficientes de las potencias de x son igual a cero y en segundo lugar, que la forma general de D_{2n} es el coeficiente $2n+1$-ésimo término, para cualquier x^{2n} incluso potencia de x, es el siguiente:

$$\frac{1}{2\cdot 3\cdots 2n+1} - \frac{1}{2}\cdot\frac{1}{2\cdot 3\cdots 2n} + \frac{B_1}{2}\cdot\frac{1}{2\cdot 3\cdots 2n-1} + \frac{B_3}{2\cdot 3\cdot 4}\cdot\frac{1}{2\cdot 3\cdots 2n-3} + \frac{B_3}{2\cdot 3\cdot 4\cdot 5\cdot 6} + \\ + \cdots + \frac{B_{2n-1}}{2\cdot 3\cdots 2n}\cdot 1 = 0 \tag{3.11}$$

multiplicando cada término por $2\cdot 3\cdots 2n$ tenemos

$$0 = -\frac{1}{2}\cdot\frac{2n-1}{2n+1} + B_1(\frac{2n}{2}) + B_3(\frac{2n\cdot(2n-1)(2n-2)}{2\cdot 3\cdot 4}) + \\ + B_5(\frac{2n\cdot(2n-1)(2n-2)\cdots(2n-4)}{2\cdot 3\cdot 4\cdot 5\cdot 6} + \cdots + B_{2n-1} \tag{3.12}$$

cuyo término general es:

$$0 = A_0 + A_1B_1 + A_3B_3 + A_5B_5 + \cdots + B_{2n-1} \tag{3.13}$$

A_1,A_3,$\cdots$ siendo las funciones de n que respectivamente pertenecen a B_1, B_3, $\cdots$.

Al examinar las fórmulas (3.11) y (3.12) anteriores percibimos que estas fórmulas están contenidas en la anterior a ellas, fórmula (3.10), de donde se derivan y considerandolas en sí mismas por separado y de forma independiente, n puede ser cualquier número entero; aunque cuando (3.11) se produce como uno de los D's en (3.10), es obvio que entonces n no es arbitraria, pero es siempre una función contenida en la distancia de esa D desde el principio. Si esa distancia fuera $= d$, entonces

$$2n+1 = d, \quad y \quad n = \frac{d-1}{2} \quad para \quad los \quad pares$$

$$2n = d, \quad y \quad n = \frac{d}{2} \quad para \quad los \quad impares$$

Operaremos con la fórmula (3.12); por lo tanto se debe recordar que las condiciones para el valor de n ahora están modificados, y que n es un número entero arbitrario. Esta circustancia, combinada con el hecho (que podemos percibir con facilidad) que todo lo qu n es, cada término de (8) después $(n+1)$ es $= 0$ y el $(n+1)$-ésimo término en sí mismo es siempre $= B_{2n-1}\cdot\frac{1}{1} = B_{2n-1}$, nos permite encontrar el valor (tanto de manera numérica como ariméticamente) de cualquier enésimo *Número de Bernoulli* B_{2n-1}, en términos de todas las anteriores. Si nos centramos en la fórmula (3.12) tendremos en consideración los sigueintes cálculos:

- Para $n = 1$, obtenemos B_1.

- Para $n = 2$, sustituyendo el valor de B_1 calculado anteriormente, obtenemos B_3.

- Para $n = 3$, sustituyendo los dos valores anteriores obtenemos B_5.

- Y así sucesivamente para cualquier valor de n.

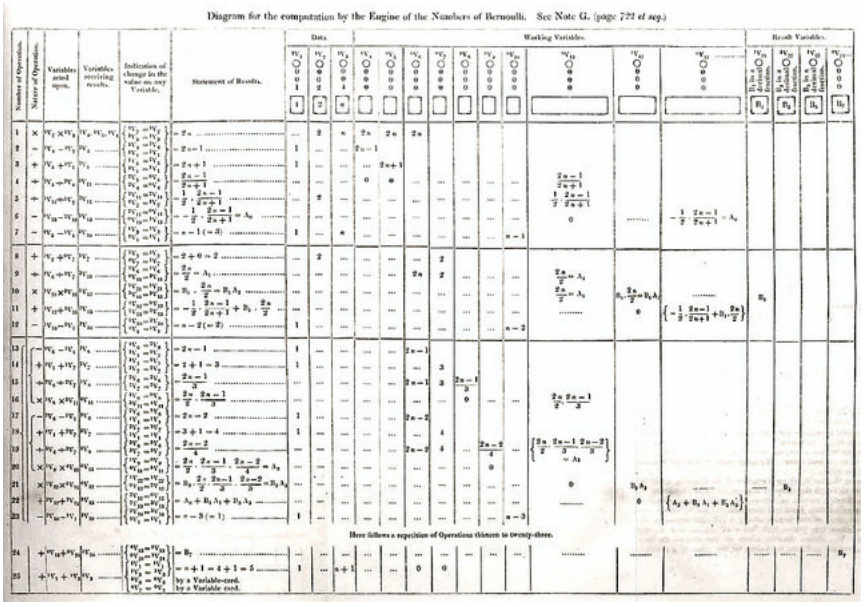

Figura 3.9: Números de Bernoulli.

Ahora mostraremos un diagrma en el que podemos encontar cada uno de los números.

En el diagrama se representan las columnas de la Máquina dispuestas a efectuar el cálculo de B_{2n-1} (Ir a la **Nota D** para recordar la notación de las tablas). Seis datos son requeridos para realizar estas combinaciones. Estos datos son $n = 1$, $n = 2$, $n = 3$ y $n = 4$ para calcular B_1, B_3 y B_5 (números impares como veremos en el siguiente capítulo). Para $n = 5$, obtendríamos B_7; para $n = 6$ obtendríamos B_9 y así secesivamente. Por tanto, el número real de datosnecesarios siempre será $n + 2$ para $n = n$, siendo $\overline{(n+2+3)}$ los sucesivos **Números de Bernoulli**, colocándolos en las distintas columnas. En la tabla no hemos tenido en cuenta los signos de los números, se ha dejado un espacio en blanco para los mismos. Ada también nos explica la función que desempeña cada una de las diferentes tarjetas. Veámoslo.

Las tarjetas de operación $1, 2, 3, 4, 5$ y 6 preparan la operación $(-\frac{1}{2}) \cdot \frac{2n-1}{2n+1}$. Por lo tanto, la tarjeta 1 multiplica por dos n, y las tres siguientes tarjetas pertenecientes a V_4, V_5 y V_6 permiten que el resultado $2n$ sea colocado en cada una de estas últimas columnas.

Para la tarjeta 6, la cantidad positiva pasa a ser negativa, simplemente restando la cantidad de una columna que contiene un cero (el signo de V_8 se convertiría en dicho proceso). La tarjeta 7 será initeligible a menos que tengamos en cuenta que si estábamos calculando $n = 1$ en lugar de $n = 4$. La tarjeta 6 habrá completado el cómputo de B_1, y la Máquina guardará en V_{21} dicho valor, y comenzará de nuevo con las operaciones $1, 2, 3, \cdots$ para el valor de $n = 2$[10] con el fin de introducir el cálculo de B_3. La operación 7 debe llevar a cabo un resultado igual a cero (si $n = 1$); o un resultado mayor que cero, como el presente caso y la Máquina efectúa uno u otro.

Las operaciones 8, 9 y 10 llevan a cabo las operaciones de

$$-\frac{1}{2} \cdot \frac{2n-1}{2n+1} + B_1 \frac{2n}{2}$$

En la operación 9 vemos un ejemplo de un índice superior que a su vez se convierte en un valor después de haber pasado de los valores anteriores a cero. V_{11} ha sido sucesivamente

[10] Tratándose de un Buble, como ya explicamos en la **Nota E**.

V_{11}^0, V_{11}^1, V_{11}^2, V_{11}^0 y V_{11}^3, y seguirá cambiando hasta que haya valores que calcular y esta variable sea necesaria para el cómputo.

La operación 12 tiene que realizar la misma función que la operación 7; ya que si $n = 2$ la operación 11 ha completado ya el cálculo encontrándo B_3.

Las operaciones de la 13 a la 20 calculan A_3. Siempre A_{2n-1} se compone de $2n - 1$ factores y A_3 tiene tres fractores. Las operaciones 13, 14, 15 y 16 hacen que el segundo de estos factores se multiplique por el primero y las operaciones 17, 18, 19 y 20 hacen que el tercer factor se multiplique por el producto de los dos factores anteriores.

La operación 23 desempeña la misma función que las operaciones 7 y 11, ya que si $n = 3$ las operaciones 21 y 22 completan el cálculo de B_5. Para llegar a B_7 se ha de repetir el grupo de operaciones desde la 13 a la 20 y luego repetir las operaciones 21 y 22 para obtener el resultado.

Nos damos cuenta que cada unidad añade a n en B_{2n-1} e implica una repetición adicional de operaciones $(13 \cdots 23)$ para el cálculo de B_{2n-1}. No todas las operaciones son las mismas, pero sin embargo, para cada una de estas repeticiones se requieren suministrar los mismos pares de columnas a excepción de la operación 21 que necesita B_5 de V_{23} en vez de B_3 de V_{22}. Esta identidad en las columnas que suministran los números necesario no se ha de confundir con la identidad de los valores de tienen esas columnas. La mayor parte de esos valores sufren alteraciones durante una actuación de las operaciones $(13 \cdots 23)$, y em consecuencia las columnas presentan un nuevo conjunto de valores para la próxima actuación de $13 \cdots 23$. Cuando termina la repeticion de $13 \cdots 23$ con el cálculo de B_7 la alteración de los valores de las variables son:

$V_6 = 2n - 4$ en vez de $2n - 2$.

$V_7 = 4$ en vez de 6.

$V_{10} = 0$ en vez de 1.

$V_{13} = A_0 + A_1B_1 + A_3B_3 + A_5B_5$ en lugar de $A_0 + A_1B_1 + A_3B_3$.

Las operaciones 24 y 25 son las que hacen posible este propósito. Hemos de tener en cuenta que cuando se producen las repoeticiones $13 \cdots 23$ se producen cambios en algunos de los índices superiores durante el transcurso de la repetición, por ejemplo, V_6^3 pasará a V_6^4 y V_6^5. La operación 21 siempre requiere de uno de sus factores y la operacion 24 siempre pone su resultado en una nueva columna. Teniedo essto en cuenta, podemos concluir que para $n = 1$ necesitamos nueve operaciones, para $n = 2$ se necesitan catorce operaciones y cuando $n > 2$ se necesitan veinticinco operaciones.

Si ahora aplicamos la notación para los ciclos vista en la **Nota E** podemos expresar las operaciones para el cálculo de los *Números de Bernoulli* como vemos en la Figura 3.10:

(1...7), (24, 25) .... gives	B_1	=1st number;	(n being = 1).
(1...7), (8...12), (24, 25)	B_3	=2nd ......;	(n ..... = 2).
(1...7), (8...12), (13...23), (24, 25)	B_5	=3rd ....;	(n ..... = 3).
(1...7), (8...12), 2(13...23), (24, 25)	B_7	=4th ....;	(n ..... = 4).
....			
....			
(1...7), (8...12), $\Sigma(+1)^{n-2}$(13...23), (24, 25)...	B_{2n-1}	=nth ......;	(n ..... = n).

$$(1\ldots7), (24,25), \underset{\text{limits 1 to } n}{\Sigma(+1)^n} \left\{ \underset{\text{limits 0 to } (n+2)}{(1\ldots7), (8\ldots12), \Sigma(n+2)(13\ldots23), (24,25)} \right\}$$

Figura 3.10: Aplicación de la notación.

que representa las operaciones totales para el cálculo de cada número en la sucesión de B_1 a B_{2n-1} imclusive.

En esta fórmula vemos un ciclo variable de primer orden, y un ciclo ordinario de segundo orden. Este último ciclo en este caso incluye un ciclo variable. Observando el diagrama percibimos que V_6 siempre trabaja los numeradores, V_7 los denominadores, V_8 siempre recibe el $2n-3$-avo factor de A_{2n-1}, V_9 se encarga del $2n-1$-ésimo y V_10 decide siempre cuál de los dos procesos subsiguientes va a continuar, etc.

CAPÍTULO 4
Cálculo de los números de Bernoulli

Para poder entender el trabajo que Ada realizó en su tiempo, primero vamos a estudiar y entender qué son y cómo funcionan los Números de Bernoulli.

Comenzaremos hablando de Jakob Bernoulli que lo encontramos en la Imágen 4.1, fundador de los famosos Números de Bernoulli[1]; famoso científico y matemático suizo, hijo mayor de la familia Bernoulli[2]. Se graduó en Teología en el año 1676. Comenzó a ejercer como profesor de mecánica en la Universidad de Basilea en 1683.

Figura 4.1: Jakob Bernoulli.

A él se le atribuye el término *integral* usado por primera vez en 1690 por el mismo. Utilizó tempranamente las coordenadas polares y descubrió el isócrono[3].También trabajó en la Teoría de la Probabilidad[4].

Jakob Bernoulli falleció el 16 de Agosto de 1705 en Basilea, Suiza.

Una vez sabido quién estudió los Números de Bernoulli, es el turno de ver como funcionan, cómo veremos en el apartado siguiente.

[1]Esta denominación fue dada por Abraham de Moivre, matemático francés famoso por la fórmula de De Moivre, cuya fórmula nos dice que para cualquier número real o complejo x y para cualquier entero n se verifica que:

$$(\cos x + i \sin x)^n = \cos(nx) + i \sin(nx).$$

[2]Se trata de una familia de matemáticos y físicos suizos que destacó en el mundo científico a finales del siglo XVII.

[3]Curva que se forma al caer verticalmente un cuerpo con velocidad uniforme.

[4]Rama de las matemáticas que estudia los fenómenos aleatorios estocásticos, es decir, sucesos cuyo comportamiento no es determinista.

4.1 Definición

Los números de Bernoulli, denotados por B_n forman una sucesión de números racionales.

Se pueden definir de diversas formas equivalentes: Como los términos independientes de los polinomios de Bernoulli $B_p(x)$ correspondientes, es decir, $B_p = B_p(0)$, Mediante una función generatriz G(x), en este caso:

$$G(x) = \frac{x}{e^x - 1} = \sum_{n=0}^{\infty} B_n \frac{x^n}{n!} : \quad \text{si} \quad |x| < 2\pi$$

dónde cada coeficiente B_n de la Serie de Taylor[5]es el n-ésimo número de Bernoulli.

Ahora vamos a moestrar una tabla en la que se ven los resultados según el Número de Bernoulli que estemos calculando.

n	B_n
0	1
1	-1/2
2	1/6
3	0
4	-1/30
5	0
6	1/42
7	0
8	-1/30
9	0
10	5/66

Tabla 4.1: Listado Números de Bernoulli.

4.2 Explicación Ada Byron

La explicación detallada que proporciona Ada de los Números de Bernoulli para la Máquina Analítica, lo expone en la **Nota G**.

En este apartado introduciremos una imagen (Figura 4.1) y veremos operación por operación como se van calculando dichos números.

Para poder entender el cálculo, vamos a describir las diferentes columnas que forman la tabla sobre la que se calculan los números.

La primera columna muestra el número de operación en el que nos encontramos. La segunda, indica el tipo de operación que se va a realizar. En la tercera columna encontramos las variables con las que se va a operar. A continuación, en la cuarta columna

[5]Es una aproximación de funciones mediante una serie de potencias o suma de potencias enteras de polinomios como $(x-a)^n$ llamados términos de la serie, dicha suma se calcula a partir de las derivadas de la función para un determinado valor o punto a suficientemente derivable sobre la función y un entorno sobre el cual converja la serie, cuya expresión general es:

$$\sum_{n=0}^{\infty} \frac{f^(n)(a)}{n!}(x-a)^n$$

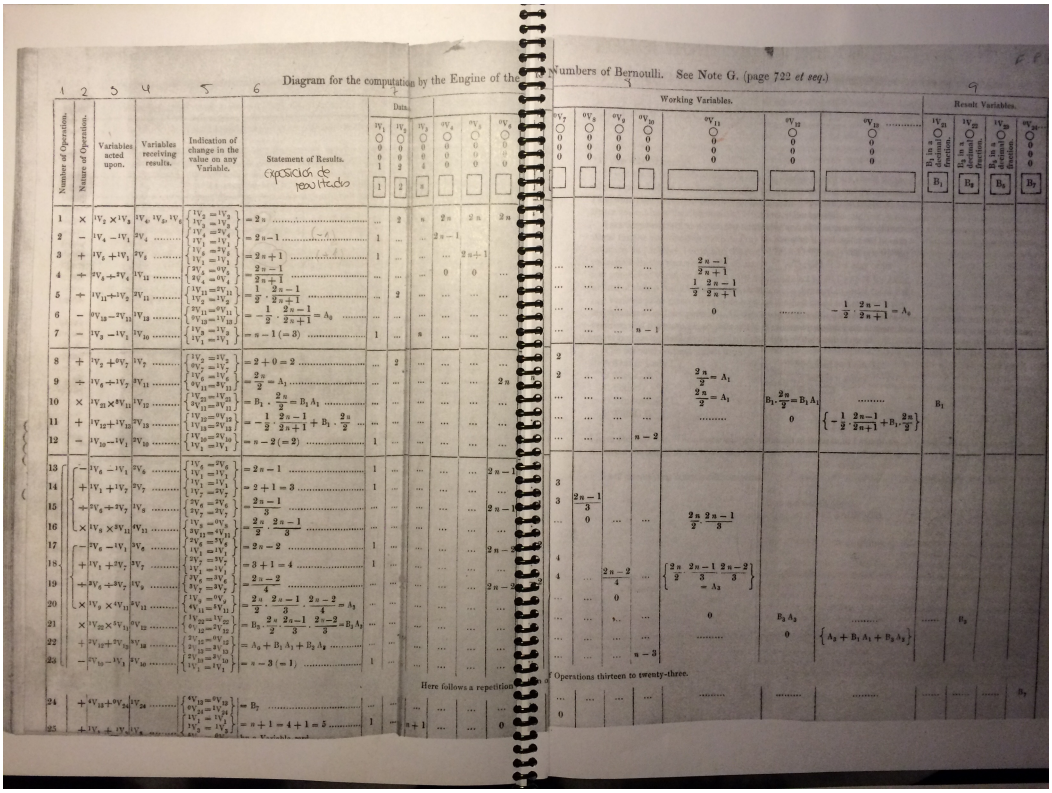

Figura 4.2: Diagrama Números de Bernoulli.

hallamos las variables en las que se van a almacenar los resultados, teniéndo en cuenta que los subíndices cambiarán dependiendo del número de veces que dichas variables sean empleadas. La quinta columna nos describe el valor que contendrán las diferentes variables. En la sexta columna se nos expone los reultados de las operaciones efectadas. Las sigueintes columnas pertenecen a los diferentes tipos de variables, perteneciéndo las tres primeras columnas a las Variables de Datos, las diez siguientes columnas a las Variables Temporales y las cuatro últimas columnas a las Variables de Resultados.

Una vez entendido ésto, comenzaremos a explicar el diagrama operación por operación. El diagrama está dividido en tres bloques. El primero de ellos contiene las siete primeras operaciones.

Lo primero que tenemos que tener en cuenta son los valores que están almacenados en cada variable para poder efectuar las diferentes operaciones. Así, en la primera operación vamos a multiplicar el valor que hay almacenado en V_2 por el valor de V_3 siendo respectivamente 2 y n, almacenadolas en las variables $V_4, V_5 y V_6$. En la segunda operación vamos a restar el valor de V_4 menos el valor de $V1$ dando como resultado $2n-1$ y se va a almacenar en V_4^2[6]. En la tercera operación vamos a sumar el valor que poseen las variables V_5 y V_1 y se almacena en la variable V_5. En la cuarta operación vamos a dividir los valores que tienen las variables V_5 y V_4 y será guardado en V_{11}; hemos de tener en cuenta que para el cálculos de los Números de Bernoulli se opera con derivadas, por eso en esta operación y en alguna de las siguientes se trata de la inversa $\frac{2n-1}{2n+1}$ en vez de tener $\frac{2n+1}{2n-1}$. En la sigueinte operación hemos de dividir V_{11} entre V_2 y se almacena en V_{11}^2 dando como resultado $\frac{1}{2}\cdot\frac{2n-1}{2n+1}$ teniéndo en cuenta la consideración que ya mencionamos anteriormente. La sexta operación resta V_{13} menos V_{11} y se almacena en V_{13} dando lugar a A_0 que como

[6]Fijese que el subíndice ha cambiado a dos ya que se ha empleado dos veces, la primera vez en la operación uno y la segunda en la operación dos; y así sucesivamente con todos y cada uno de los subíndices que sufran alguna modificación.

ya vimos en el apartado correspondiente a la **Nota G** la forma general está compuesta de dos términos A_n y B_n correspondiéndo a los Números, es decir, $-\frac{1}{2}\cdot\frac{2n-1}{2n+1} = A_0$. La séptima operación se trata de restar el valor de V_3 menos el valor que contenga la variable V_1 guardándo el resultado en V_{10} dándo lugar a $n-1=3$. Con estas siete operaciones finaliza el primer bloque, como podemos ver en la Figura 4.3.

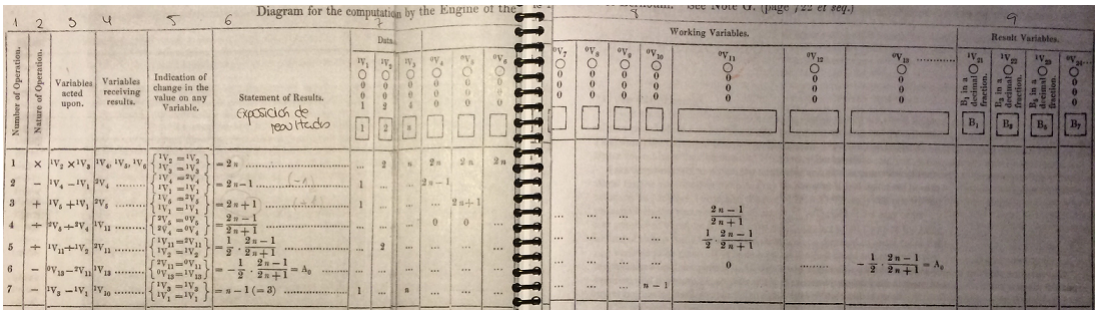

Figura 4.3: Primer bloque del diagrama.

A continuación describimos el segundo bloque, que podemos visualizar en la Figura 4.4, formado por las cinco siguientes instrucciones. La octava operación consiste en la adicción de la variable V_2 con la variable V_7 y almacenándose el resultado en la variable V_7, así pues obtenemos $2+0=2$. La novena operación trata de dividir V_6 entre V_7 almacenándose en V_{11} generando como resultado $A_1 = \frac{2n}{2}$ La décima operación es una multiplicación de V_{21} por V_{11} y se guarda en V_{12} dando lugar al primer Número de Bernoulli, $B_1\cdot\frac{2n}{2} = B_1A_1$. La siguiente operación es una adicción entre V_{12} y V_{13} guardando el resultado en V_{13}, lo que nos genera $-\frac{1}{2}\cdot\frac{2n-1}{2n+1} + B_1\cdot\frac{2n}{2}$. Para acabar el bloque tenemos una diferencia entre V_{10} y V_1 dando lugar a $n-2=2$.

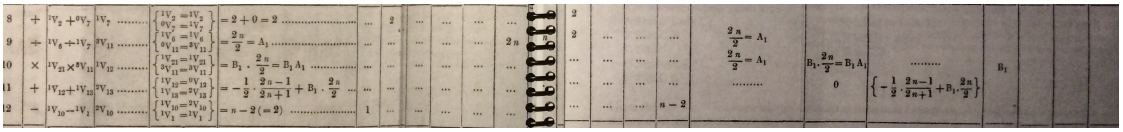

Figura 4.4: Segundo bloque del diagrama.

Ahora comenzamos a explicar el último bloque del diagrama, mostrado en la Figura 4.5. En este bloque nos encontramos un bucle desde la decimotercera hasta la vigésimo tercera operación. La primera operación del tercer bloque es una diferencia entre V_6 y V_1 quedando almacenado el resultado en V_6, $2n-1$. La siguiente operación es una adicción entre V_1 y V_7 y se almacena el resultado en V_7, $2+1=3$. A continuación tenemos una división entre V_6 y V_7, guardando el resultado en V_8, $\frac{2n-1}{3}$. La siguiente operación se trata de una multiplicación entre V_8 y $V_1$1 quedándo almacenado el resultado en V_{11}, así pues tenemos $\frac{2n}{2}\cdot\frac{2n-1}{3}$. La decimoséptima operación es una diferencia entre V_6 y V_1, generando como resultado $2n-2$. LA siguiente operación es una adicción entre los valores de V_1 y los valores de V_7, $3+1=4$. A continuación tenemos una división entre V_6 y V_7 cuyo almacenamiento es en V_9, $\frac{2n-2}{4}$. La vigésima operación es una multiplicación de los valores de V_9 con los valores de V_{11} dando como resultado $A_3 = \frac{2n}{2}\cdot\frac{2n-1}{3}\cdot\frac{2n-2}{4}$[7]. La sigueinte operación consiste en la multiplicación de V_{22} y V_{11} generando así el tercer Número de Bernoulli, $B_3\cdot\frac{2n}{2}\cdot\frac{2n-1}{3}\cdot\frac{2n-2}{3} = B_3A_3$. La penúltima operación es una adicción entre V_{12} y V_{13} cuyo resultado es almacenado en V_{13}, es decir, $A_0 + B_1A_1 + B_3A_3$. La última operación del bucle consiste en una diferencia entre V_{10} y V_1, generando como resultado $n-3=1$. Con este bloque habríamos calculado el tercer Número de Bernoulli. Para calcular los

[7] Fíjese que conforme va incrementando el valor del númerador el valor del denominador también aumenta en $n+1$.

siguientes números tendríamos que repetir el bucle tantas veces como números estemos dispuestos a calcular.

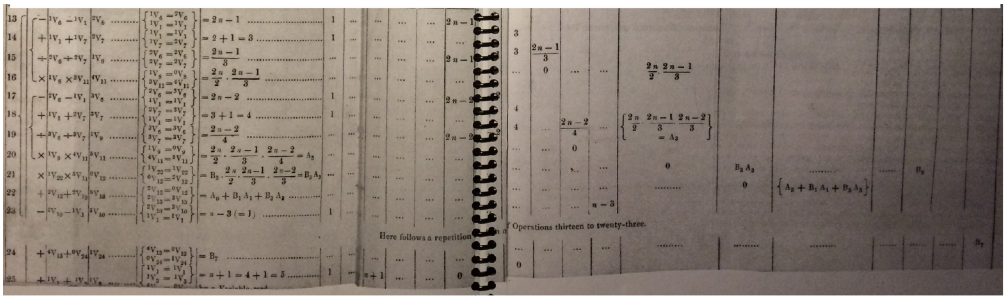

Figura 4.5: Tercer bloque del diagrama.

Una vez que se ha concluido la explicación de este apartado, podemos dar a modo de resumen lo siguiente:

Para $n = 1$ encontramos B_1. Para $n = 2$ encontramos B_3. Para $n = 3$ encontramos B_5 y así sucesivamente tantas veces como números queramos calcular.

4.3 Explicación de la autora del trabajo

Tras ver en los capítulos anteriores la labor que realizó Ada sobre la Máquina Analítica y la definición de los números de Bernoulli, he intentado hacerlo realidad.

Para desarrollar esta tarea, he empleado dos lenguajes de programación: el primero ha sido Java[8], con el que estoy más familiarizada ya que desde el primer curso de la carrera lo hemos empleado; el segundo lenguaje ha sido, C++[9], que me ha llevado más tiempo pero ha salido a delante.

En este apartado se muestran ambos códigos, comenzaremos con el código Java.

-**Java**: en este código tenemos dos clases principales, la clase BigRational, que nos proporciona la funcionalidad que permite trabajar con racionales y la clase BerBigRational, dónde se encuentra el Main.

Clase BigRational:

```
import java.math.BigDecimal;
import java.math.BigInteger;
import java.math.RoundingMode;

public class BigRational implements Comparable<BigRational> {

    public final static BigRational ZERO = new BigRational(0);

    private BigInteger num;   // the numerator
    private BigInteger den;   // the denominator


    // create and initialize a new BigRational object
    public BigRational(int numerator, int denominator) {
        this(new BigInteger("" + numerator), new BigInteger("" + denominator));
    }

    // create and initialize a new BigRational object
    public BigRational(int numerator) {
        this(numerator, 1);
    }

    // create and initialize a new BigRational object from a string, e.g.,
       "-343/1273"
    public BigRational(String s) {
        String[] tokens = s.split("/");
        if (tokens.length == 2)
            init(new BigInteger(tokens[0]), new BigInteger(tokens[1]));
        else if (tokens.length == 1)
            init(new BigInteger(tokens[0]), BigInteger.ONE);
        else
            throw new IllegalArgumentException("For input string: \"" + s + "\"
                ");
    }

    // create and initialize a new BigRational object
    public BigRational(BigInteger numerator, BigInteger denominator) {
        init(numerator, denominator);
    }
```

[8]Se trata de un lenguaje concurrente y orientado a objetos. Fue lanzado en 1995 por James Gosling. Con este lenguaje se puede crear el código en una máquina y ser ejecutado en otra sin necesidad de ser recompilado.

[9]C++ es un lenguaje de programación híbrido, ya que permite la manipulación de objetos y conserva las caracteríticas del lenguaje C. C++ fue Bjarne Stroustrup a mediados de 1980. Es un lenguaje multiparadigma (programación estructurada y programación orientada a objetos)

```
38
39      private void init(BigInteger numerator, BigInteger denominator) {
40
41          // deal with x / 0
42          if (denominator.equals(BigInteger.ZERO)) {
43             throw new ArithmeticException("Denominator is zero");
44          }
45
46          // reduce fraction
47          BigInteger g = numerator.gcd(denominator);
48          num = numerator.divide(g);
49          den = denominator.divide(g);
50
51          // to ensure invariant that denominator is positive
52          if (den.compareTo(BigInteger.ZERO) < 0) {
53              den = den.negate();
54              num = num.negate();
55          }
56      }
57
58      // return string representation of (this)
59      public String toString() {
60          if (den.equals(BigInteger.ONE)) return num + "";
61          else                            return num + "/" + den;
62      }
63
64      // return { -1, 0, + 1 } if a < b, a = b, or a > b
65      public int compareTo(BigRational b) {
66          BigRational a = this;
67          return a.num.multiply(b.den).compareTo(a.den.multiply(b.num));
68      }
69
70      // is this BigRational negative, zero, or positive?
71      public boolean isZero()     { return compareTo(ZERO) == 0; }
72      public boolean isPositive() { return compareTo(ZERO)  > 0; }
73      public boolean isNegative() { return compareTo(ZERO)  < 0; }
74
75      // is this Rational object equal to y?
76      public boolean equals(Object y) {
77          if (y == this) return true;
78          if (y == null) return false;
79          if (y.getClass() != this.getClass()) return false;
80          BigRational b = (BigRational) y;
81          return compareTo(b) == 0;
82      }
83
84      // hashCode consistent with equals() and compareTo()
85      public int hashCode() {
86          return this.toString().hashCode();
87      }
88
89
90      // return a * b
91      public BigRational times(BigRational b) {
92          BigRational a = this;
93          return new BigRational(a.num.multiply(b.num), a.den.multiply(b.den));
94      }
95
96      // return a + b
97      public BigRational plus(BigRational b) {
98          BigRational a = this;
99          BigInteger numerator   = a.num.multiply(b.den).add(b.num.multiply(a.den
                ));
100         BigInteger denominator = a.den.multiply(b.den);
```

```
101         return new BigRational(numerator, denominator);
102     }
103
104     // return -a
105     public BigRational negate() {
106         return new BigRational(num.negate(), den);
107     }
108
109     // return a - b
110     public BigRational minus(BigRational b) {
111         BigRational a = this;
112         return a.plus(b.negate());
113     }
114
115     // return 1 / a
116     public BigRational reciprocal() {
117         return new BigRational(den, num);
118     }
119
120     // return a / b
121     public BigRational divides(BigRational b) {
122         BigRational a = this;
123         return a.times(b.reciprocal());
124     }
125
126     // return double reprentation (within given precision)
127     public double doubleValue() {
128         int SCALE = 32;         // number of digits after the decimal place
129         BigDecimal numerator   = new BigDecimal(num);
130         BigDecimal denominator = new BigDecimal(den);
131         BigDecimal quotient    = numerator.divide(denominator, SCALE,
                RoundingMode.HALF_EVEN);
132         return quotient.doubleValue();
133     }
134
135
136     // test client
137     public static void main(String[] args) {
138         BigRational x, y, z;
139
140         // 1/2 + 1/3 = 5/6
141         x = new BigRational(1, 2);
142         y = new BigRational(1, 3);
143         z = x.plus(y);
144         System.out.println(z);
145
146         // 8/9 + 1/9 = 1
147         x = new BigRational(8, 9);
148         y = new BigRational(1, 9);
149         z = x.plus(y);
150         System.out.println(z);
151
152         // 1/200000000 + 1/300000000 = 1/120000000
153         x = new BigRational(1, 200000000);
154         y = new BigRational(1, 300000000);
155         z = x.plus(y);
156         System.out.println(z);
157
158         // 1073741789/20 + 1073741789/30 = 1073741789/12
159         x = new BigRational(1073741789, 20);
160         y = new BigRational(1073741789, 30);
161         z = x.plus(y);
162         System.out.println(z);
163
```

```
164         //  4/17 * 17/4 = 1
165         x = new BigRational(4, 17);
166         y = new BigRational(17, 4);
167         z = x.times(y);
168         System.out.println(z);
169 
170         // 3037141/3247033 * 3037547/3246599 = 841/961
171         x = new BigRational(3037141, 3247033);
172         y = new BigRational(3037547, 3246599);
173         z = x.times(y);
174         System.out.println(z);
175 
176 
177         // 1/6 - -4/-8 = -1/3
178         x = new BigRational( 1,  6);
179         y = new BigRational(-4, -8);
180         z = x.minus(y);
181         System.out.println(z);
182 
183         // 0
184         x = new BigRational(0,  5);
185         System.out.println(x);
186         System.out.println(x.plus(x).compareTo(x) == 0);
187         /// System.out.println(x.reciprocal());   // divide-by-zero
188 
189         // -1/200000000 + 1/300000000 = 1/120000000
190         x = new BigRational(-1, 200000000);
191         y = new BigRational(1, 300000000);
192         z = x.plus(y);
193         System.out.println(z);
194 
195     }
196 
197 }
```

Clase BerBigRational:

```
1  /**
2   * Write a description of class BerBigRat here.
3   *
4   * @author Sandra Adobes
5   * @version 19/04/2016
6   */
7  import java.math.BigInteger;
8  public class BerBigRat
9  {
10     public static void main(String args[]) {
11         int m = Integer.parseInt(args[0]);
12         BigRational f1,f2,fd,aux;
13         int i,j;
14         //Usaremos una lista en la que guardaremos los valores obtenidos hasta
               el momento
15         //Reduce el tiempo de computo respecto a una solucion recursiva, pues
               almacena el valor
16         //de Bernoulli ya calculado, y en sucesivas recursiones accede a este
               dato
17         BigRational[] bernLista = new BigRational[m+1];
18 
19         /**
20          *Este ejemplo es para m=2
21          *
22          *  f(m)*ber(i)          f(m)*ber(i+1)
23          * ------------- + ------------------- + ...=
24          *  f(i)*f(m-(i-1))     f(i+1)*f(m-(i+1-1))
```

```
25          */
26         //Almacenamos los primeros numeros
27         bernLista[0]=new BigRational(1,1);
28         bernLista[1]=new BigRational(-1,2);
29         //Esta bucle va calculando el numero de Bernoulli donde nos encontramos
30         for(i=2;i<=m-1;i++){
31             //Inicializamos el contenido de la lista
32             bernLista[i] = new BigRational(0, 1);
33             for(j=0;j<i;j++){
34 
35                 f1= new BigRational(factorial(i),BigInteger.ONE );//f(i)
36                 f2= new BigRational(factorial(j).multiply(factorial((i-(j-1))))
                       , BigInteger.ONE);
37                 //f(j)*f((m-(j-1)))
38 
39                 aux=f1.times(bernLista[j]);//f(i)*bernLista[j]
40                                            //  f(i)*bernLista[j]
41                 fd=aux.divides(f2);// ----------------------
42                                            //  f(j)*f((m-(j-1)))
43 
44                 bernLista[i]=bernLista[i].plus(fd);//Sum(BernLista[0..j-1])+fd(
                       j))
45             }
46 
47             bernLista[i]=bernLista[i].negate();//BernLista[i]=-BernLista[i]
48         }
49 
50         //Imprime por pantalla
51         for(i=0;i<m;i++){
52             System.out.print("Bernoulli ("+i+") := "+bernLista[i]+"\n");
53         }
54 
55     }
56     //Calcula el factorial, funciona para enteros grandes
57     public static BigInteger factorial(int num) {
58         BigInteger factorial = BigInteger.ONE;
59         for (int i = num; i > 0; i--) {
60             factorial = factorial.multiply(BigInteger.valueOf(i));
61         } return factorial;
62     }
63 
64 }
```

- **C++**: La versión de C++ es un poco más compleja. En este lenguaje tenemos tres tipos de archivos:

1.- Los archivos .cc: dónde se enceuntra el código perteneciente a las librerías de descarga.

2.- Los archivos .ccp: dónde encontramos el código fuente, es decir, el código que nosotros hemos creado; encontrándo así tres clases: Bernoulli.ccp, BigInteger.ccp y BigRational.ccp que mostraremos acontinuación.

3.- Los archivos .h: dónde se encuentra el código perteneciente al prototipo, es decir, la cabecera de la función con sus correspondientes parámetros de entrada.

En este apartado mostramos el código perteneciente a los archivos .ccp, el resto de códigos podemos encontrarlos en el Apendice A.

Clase Bernoulli: es el programa principal. Contiene el cálculo del factorial.

```
1 // Standard libraries
2 #include <string>
3 #include <iostream>
4 
```

```
// BigRational Library
#include "BigRational.h"
// 'BigIntegerLibrary.hh' includes all of the library headers.
#include "BigIntegerLibrary.hh"
//ttmath library
#include "ttmath.h"

typedef ttmath::Big<TTMATH_BITS(32), TTMATH_BITS(64)> MyBig;


//Calcula el factorial, funciona para enteros grandes
BigInteger factorial(int num) {
     BigInteger factorial = BigInteger(1);

   for (int i = num; i > 0; i--)
   {
           factorial.multiply(factorial, BigInteger(i));
     }

   return factorial;
}

int main(int argc, char* argv[]) {

  if (argc > 1)
  {
    int m = atoi(argv[1]); //convierte a entero el parametro de entrada 1

     //Usaremos una lista en la que guardaremos los valores obtenidos hasta el momento
        //Reduce el tiempo de computo respecto a una solucion recursiva, pues almacena el valor
        //de Bernoulli ya calculado, y en sucesivas recursiones accede a este dato
    std::vector<BigRational> bernLista;



    //Almacenamos los primeros numeros, insertar en el vector de Bernoullis los dos BigRational que inicializan el vector con el constructor
    bernLista.push_back(BigRational(1,1));
    bernLista.push_back(BigRational(-1,2));

    try
    {
       //Esta bucle va calculando el numero de Bernoulli donde nos encontramos
      for (int i=bernLista.size(); i <= m; i++)
      {
         //Inicializamos el contenido de la lista
        bernLista.push_back(BigRational(0,1));
        //empezamos a calcular los factoriales
        for (int j=0; j < i; j++)
        {
          BigRational f1 = BigRational(factorial(i), BigInteger(1));     //f(i)
          BigInteger f3 = factorial(j); //metemos el auxiliar para hacer la mult y no machacar el valor anterior
          f3.multiply(f3, factorial(i-(j-1)));
          BigRational f2 = BigRational(f3, BigInteger(1));  //f(j)*f((m-(j-1)))

          BigRational aux = f1.times(bernLista[j]); //f(i)*bernLista[j]


          BigRational fd = aux.divides(f2);

```

```
64
65            bernLista[i] = bernLista[i].plus(fd);    //Sum(BernLista[0..j-1])+fd(j
                  ))
66          }
67
68          bernLista[i]=bernLista[i].negate();  //BernLista[i]=-BernLista[i]
69        }
70      }
71      catch (exception e)
72      {
73        cout << e.what();
74      }
75      //se hace i es unsigned para que sea del mismo tipo que size() y evitar un
            warning
76      for (unsigned i = 0; i< bernLista.size() -1 ; i++)
77      {
78        cout << "Bernoulli (" << i << ") := " << bernLista[i].toString() << endl;
              //mostrar por pantalla todo el vector
79      }
80    }
81    else
82    {
83      cout << Parametro requerido no introducido. Introduzca en numero de
            Bernoulli a calcular. << endl;
84      return -1;
85    }
86  }
```

Clase BigInteger: constituye la compilación cc original

```
1  #include "BigInteger.hh"
2
3  void BigInteger::operator =(const BigInteger &x) {
4    // Calls like a = a have no effect
5    if (this == &x)
6      return;
7    // Copy sign
8    sign = x.sign;
9    // Copy the rest
10   mag = x.mag;
11 }
12
13 BigInteger::BigInteger(const Blk *b, Index blen, Sign s) : mag(b, blen) {
14   switch (s) {
15   case zero:
16     if (!mag.isZero())
17       throw "BigInteger::BigInteger(const Blk *, Index, Sign): Cannot use a
             sign of zero with a nonzero magnitude";
18     sign = zero;
19     break;
20   case positive:
21   case negative:
22     // If the magnitude is zero, force the sign to zero.
23     sign = mag.isZero() ? zero : s;
24     break;
25   default:
26     /* g++ seems to be optimizing out this case on the assumption
27      * that the sign is a valid member of the enumeration.  Oh well. */
28     throw "BigInteger::BigInteger(const Blk *, Index, Sign): Invalid sign";
29   }
30 }
31
32 BigInteger::BigInteger(const BigUnsigned &x, Sign s) : mag(x) {
33   switch (s) {
```

```
34    case zero:
35      if (!mag.isZero())
36        throw "BigInteger::BigInteger(const BigUnsigned &, Sign): Cannot use a
              sign of zero with a nonzero magnitude";
37      sign = zero;
38      break;
39    case positive:
40    case negative:
41      // If the magnitude is zero, force the sign to zero.
42      sign = mag.isZero() ? zero : s;
43      break;
44    default:
45      /* g++ seems to be optimizing out this case on the assumption
46       * that the sign is a valid member of the enumeration.  Oh well. */
47      throw "BigInteger::BigInteger(const BigUnsigned &, Sign): Invalid sign";
48    }
49  }
50
51  /* CONSTRUCTION FROM PRIMITIVE INTEGERS
52   * Same idea as in BigUnsigned.cc, except that negative input results in a
53   * negative BigInteger instead of an exception. */
54
55  // Done longhand to let us use initialization.
56  BigInteger::BigInteger(unsigned long  x) : mag(x) { sign = mag.isZero() ? zero
        : positive; }
57  BigInteger::BigInteger(unsigned int   x) : mag(x) { sign = mag.isZero() ? zero
        : positive; }
58  BigInteger::BigInteger(unsigned short x) : mag(x) { sign = mag.isZero() ? zero
        : positive; }
59
60  // For signed input, determine the desired magnitude and sign separately.
61
62  namespace {
63    template <class X, class UX>
64    BigInteger::Blk magOf(X x) {
65      /* UX(...) cast needed to stop short(-2^15), which negates to
66       * itself, from sign-extending in the conversion to Blk. */
67      return BigInteger::Blk(x < 0 ? UX(-x) : x);
68    }
69    template <class X>
70    BigInteger::Sign signOf(X x) {
71      return (x == 0) ? BigInteger::zero
72        : (x > 0) ? BigInteger::positive
73        : BigInteger::negative;
74    }
75  }
76
77  BigInteger::BigInteger(long  x) : sign(signOf(x)), mag(magOf<long , unsigned
        long >(x)) {}
78  BigInteger::BigInteger(int   x) : sign(signOf(x)), mag(magOf<int  , unsigned
        int  >(x)) {}
79  BigInteger::BigInteger(short x) : sign(signOf(x)), mag(magOf<short, unsigned
        short>(x)) {}
80
81  // CONVERSION TO PRIMITIVE INTEGERS
82
83  /* Reuse BigUnsigned's conversion to an unsigned primitive integer.
84   * The friend is a separate function rather than
85   * BigInteger::convertToUnsignedPrimitive to avoid requiring BigUnsigned to
86   * declare BigInteger. */
87  template <class X>
88  inline X convertBigUnsignedToPrimitiveAccess(const BigUnsigned &a) {
89    return a.convertToPrimitive<X>();
90  }
```

```
91
92  template <class X>
93  X BigInteger::convertToUnsignedPrimitive() const {
94    if (sign == negative)
95      throw "BigInteger::to<Primitive>: "
96        "Cannot convert a negative integer to an unsigned type";
97    else
98      return convertBigUnsignedToPrimitiveAccess<X>(mag);
99  }
100
101 /* Similar to BigUnsigned::convertToPrimitive, but split into two cases for
102  * nonnegative and negative numbers. */
103 template <class X, class UX>
104 X BigInteger::convertToSignedPrimitive() const {
105   if (sign == zero)
106     return 0;
107   else if (mag.getLength() == 1) {
108     // The single block might fit in an X.  Try the conversion.
109     Blk b = mag.getBlock(0);
110     if (sign == positive) {
111       X x = X(b);
112       if (x >= 0 && Blk(x) == b)
113         return x;
114     } else {
115       X x = -X(b);
116       /* UX(...) needed to avoid rejecting conversion of
117        * -2^15 to a short. */
118       if (x < 0 && Blk(UX(-x)) == b)
119         return x;
120     }
121     // Otherwise fall through.
122   }
123   throw "BigInteger::to<Primitive>: "
124     "Value is too big to fit in the requested type";
125 }
126
127 unsigned long  BigInteger::toUnsignedLong () const { return
      convertToUnsignedPrimitive<unsigned long >       (); }
128 unsigned int   BigInteger::toUnsignedInt  () const { return
      convertToUnsignedPrimitive<unsigned int  >       (); }
129 unsigned short BigInteger::toUnsignedShort() const { return
      convertToUnsignedPrimitive<unsigned short>       (); }
130 long           BigInteger::toLong         () const { return
      convertToSignedPrimitive  <long , unsigned long> (); }
131 int            BigInteger::toInt          () const { return
      convertToSignedPrimitive  <int  , unsigned int>  (); }
132 short          BigInteger::toShort        () const { return
      convertToSignedPrimitive  <short, unsigned short>(); }
133
134 // COMPARISON
135 BigInteger::CmpRes BigInteger::compareTo(const BigInteger &x) const {
136   // A greater sign implies a greater number
137   if (sign < x.sign)
138     return less;
139   else if (sign > x.sign)
140     return greater;
141   else switch (sign) {
142     // If the signs are the same...
143   case zero:
144     return equal; // Two zeros are equal
145   case positive:
146     // Compare the magnitudes
147     return mag.compareTo(x.mag);
148   case negative:
```

```
149     // Compare the magnitudes, but return the opposite result
150     return CmpRes(-mag.compareTo(x.mag));
151   default:
152     throw "BigInteger internal error";
153   }
154 }
155
156 /* COPY-LESS OPERATIONS
157  * These do some messing around to determine the sign of the result,
158  * then call one of BigUnsigned's copy-less operations. */
159
160 // See remarks about aliased calls in BigUnsigned.cc .
161 #define DTRT_ALIASED(cond, op) \
162   if (cond) { \
163     BigInteger tmpThis; \
164     tmpThis.op; \
165     *this = tmpThis; \
166     return; \
167   }
168
169 void BigInteger::add(const BigInteger &a, const BigInteger &b) {
170   DTRT_ALIASED(this == &a || this == &b, add(a, b));
171   // If one argument is zero, copy the other.
172   if (a.sign == zero)
173     operator =(b);
174   else if (b.sign == zero)
175     operator =(a);
176   // If the arguments have the same sign, take the
177   // common sign and add their magnitudes.
178   else if (a.sign == b.sign) {
179     sign = a.sign;
180     mag.add(a.mag, b.mag);
181   } else {
182     // Otherwise, their magnitudes must be compared.
183     switch (a.mag.compareTo(b.mag)) {
184     case equal:
185       // If their magnitudes are the same, copy zero.
186       mag = 0;
187       sign = zero;
188       break;
189       // Otherwise, take the sign of the greater, and subtract
190       // the lesser magnitude from the greater magnitude.
191     case greater:
192       sign = a.sign;
193       mag.subtract(a.mag, b.mag);
194       break;
195     case less:
196       sign = b.sign;
197       mag.subtract(b.mag, a.mag);
198       break;
199     }
200   }
201 }
202
203 void BigInteger::subtract(const BigInteger &a, const BigInteger &b) {
204   // Notice that this routine is identical to BigInteger::add,
205   // if one replaces b.sign by its opposite.
206   DTRT_ALIASED(this == &a || this == &b, subtract(a, b));
207   // If a is zero, copy b and flip its sign.  If b is zero, copy a.
208   if (a.sign == zero) {
209     mag = b.mag;
210     // Take the negative of _b_'s, sign, not ours.
211     // Bug pointed out by Sam Larkin on 2005.03.30.
212     sign = Sign(-b.sign);
```

```
213   } else if (b.sign == zero)
214     operator =(a);
215   // If their signs differ, take a.sign and add the magnitudes.
216   else if (a.sign != b.sign) {
217     sign = a.sign;
218     mag.add(a.mag, b.mag);
219   } else {
220     // Otherwise, their magnitudes must be compared.
221     switch (a.mag.compareTo(b.mag)) {
222       // If their magnitudes are the same, copy zero.
223     case equal:
224       mag = 0;
225       sign = zero;
226       break;
227       // If a's magnitude is greater, take a.sign and
228       // subtract a from b.
229     case greater:
230       sign = a.sign;
231       mag.subtract(a.mag, b.mag);
232       break;
233       // If b's magnitude is greater, take the opposite
234       // of b.sign and subtract b from a.
235     case less:
236       sign = Sign(-b.sign);
237       mag.subtract(b.mag, a.mag);
238       break;
239     }
240   }
241 }
242
243 void BigInteger::multiply(const BigInteger &a, const BigInteger &b) {
244   DTRT_ALIASED(this == &a || this == &b, multiply(a, b));
245   // If one object is zero, copy zero and return.
246   if (a.sign == zero || b.sign == zero) {
247     sign = zero;
248     mag = 0;
249     return;
250   }
251   // If the signs of the arguments are the same, the result
252   // is positive, otherwise it is negative.
253   sign = (a.sign == b.sign) ? positive : negative;
254   // Multiply the magnitudes.
255   mag.multiply(a.mag, b.mag);
256 }
257
258 /*
259  * DIVISION WITH REMAINDER
260  * Please read the comments before the definition of
261  * 'BigUnsigned::divideWithRemainder' in 'BigUnsigned.cc' for lots of
262  * information you should know before reading this function.
263  *
264  * Following Knuth, I decree that x / y is to be
265  * 0 if y==0 and floor(real-number x / y) if y!=0.
266  * Then x %y shall be x - y*(integer x / y).
267  *
268  * Note that x = y * (x / y) + (x %y) always holds.
269  * In addition, (x %y) is from 0 to y - 1 if y > 0,
270  * and from -(|y| - 1) to 0 if y < 0.  (x %y) = x if y = 0.
271  *
272  * Examples: (q = a / b, r = a %b)
273  *  a b q r
274  *  === === === ===
275  *  4 3 1 1
276  *  -4  3 -2  2
```

```
277  *  4 -3  -2  -2
278  *  -4  -3  1 -1
279  */
280 void BigInteger::divideWithRemainder(const BigInteger &b, BigInteger &q) {
281   // Defend against aliased calls;
282   // same idea as in BigUnsigned::divideWithRemainder .
283   if (this == &q)
284     throw "BigInteger::divideWithRemainder: Cannot write quotient and remainder
          into the same variable";
285   if (this == &b || &q == &b) {
286     BigInteger tmpB(b);
287     divideWithRemainder(tmpB, q);
288     return;
289   }
290
291   // Division by zero gives quotient 0 and remainder *this
292   if (b.sign == zero) {
293     q.mag = 0;
294     q.sign = zero;
295     return;
296   }
297   // 0 / b gives quotient 0 and remainder 0
298   if (sign == zero) {
299     q.mag = 0;
300     q.sign = zero;
301     return;
302   }
303
304   // Here *this != 0, b != 0.
305
306   // Do the operands have the same sign?
307   if (sign == b.sign) {
308     // Yes: easy case.  Quotient is zero or positive.
309     q.sign = positive;
310   } else {
311     // No: harder case.  Quotient is negative.
312     q.sign = negative;
313     // Decrease the magnitude of the dividend by one.
314     mag--;
315     /*
316      * We tinker with the dividend before and with the
317      * quotient and remainder after so that the result
318      * comes out right.  To see why it works, consider the following
319      * list of examples, where A is the magnitude-decreased
320      * a, Q and R are the results of BigUnsigned division
321      * with remainder on A and |b|, and q and r are the
322      * final results we want:
323      *
324      *  a A b Q R q r
325      *  -3  -2  3 0 2 -1  0
326      *  -4  -3  3 1 0 -2  2
327      *  -5  -4  3 1 1 -2  1
328      *  -6  -5  3 1 2 -2  0
329      *
330      * It appears that we need a total of 3 corrections:
331      * Decrease the magnitude of a to get A.  Increase the
332      * magnitude of Q to get q (and make it negative).
333      * Find r = (b - 1) - R and give it the desired sign.
334      */
335   }
336
337   // Divide the magnitudes.
338   mag.divideWithRemainder(b.mag, q.mag);
339
```

```
340   if (sign != b.sign) {
341     // More for the harder case (as described):
342     // Increase the magnitude of the quotient by one.
343     q.mag++;
344     // Modify the remainder.
345     mag.subtract(b.mag, mag);
346     mag--;
347   }
348
349   // Sign of the remainder is always the sign of the divisor b.
350   sign = b.sign;
351
352   // Set signs to zero as necessary.  (Thanks David Allen!)
353   if (mag.isZero())
354     sign = zero;
355   if (q.mag.isZero())
356     q.sign = zero;
357
358   // WHEW!!!
359 }
360
361 // Negation
362 void BigInteger::negate(const BigInteger &a) {
363   DTRT_ALIASED(this == &a, negate(a));
364   // Copy a's magnitude
365   mag = a.mag;
366   // Copy the opposite of a.sign
367   sign = Sign(-a.sign);
368 }
369
370 // INCREMENT/DECREMENT OPERATORS
371
372 // Prefix increment
373 void BigInteger::operator ++() {
374   if (sign == negative) {
375     mag--;
376     if (mag == 0)
377       sign = zero;
378   } else {
379     mag++;
380     sign = positive; // if not already
381   }
382 }
383
384 // Postfix increment: same as prefix
385 void BigInteger::operator ++(int) {
386   operator ++();
387 }
388
389 // Prefix decrement
390 void BigInteger::operator --() {
391   if (sign == positive) {
392     mag--;
393     if (mag == 0)
394       sign = zero;
395   } else {
396     mag++;
397     sign = negative;
398   }
399 }
400
401 // Postfix decrement: same as prefix
402 void BigInteger::operator --(int) {
403   operator --();
```

```
}
```

Clase BigRational: esta clase contiene el código perteneciente a BigRational.

```
/**
  BigRational.cpp

  Implementacion de los metodos de la clase BigRational cuyos prototipos esta
      definidos en 'BigRational.h'
**/
//Standar Includes
#include <string>
#include <sstream>
#include <iostream>

//Internal includes
#include "BigRational.h" //Definicion de la clase
#include "ExtraFunctions.h" //Funciones extras
#include "ttmath.h" //Libreria de definicion de tipos matematicos de gran
    precision. Se utiliza para definir dobles grandes

// for convenience we're defining MyBig type
// this type has 2 words for its mantissa and 1 word for its exponent
// (on a 32bit platform one word means a word of 32 bits,
// and on a 64bit platform one word means a word of 64 bits)

// Big<exponent, mantissa>
typedef ttmath::Big<TTMATH_BITS(32), TTMATH_BITS(64)> MyBig;

/*
  Devuelve un BigRational a partir de su numerador y denominador de tipo entero
      .
*/
BigRational::BigRational(int numerator, int denominator) {
  num = BigInteger(numerator);
  den = BigInteger(denominator);
}

 /*
  Devuelve un Bigrational a partir de una cadena de la forma "num/den". Tanto
      num como den deben ser convertibles a entero.
  Si la cadena no contiene el separador "/" se presupone que la cadena
      representa el numerador y el denominador es 1.
  Si la cadena no cumple con el formato esperado se devuelve una excepcion.
*/
BigRational::BigRational(string s) {
  try {

    std::vector<std::string> palabras = split_getline(s, '/'); //esta en
        funcion de utilidades extrafuntions.h

    if (palabras.size() == 2)
    {
      //DEBUG:
      //Tenemos numerador y denominador separados por "/"
      //cout << "Vector de string: " << palabras[0] <<  "/" << palabras[1] <<
          endl;

      num = stringToBigInteger(palabras[0]);
      den = stringToBigInteger(palabras[1]);
    }
    else if (palabras.size() == 1)
    {
      //DEBUG:
```

```
54        //Si la cadena no contiene el caracter separador "/" he de suponer que se
              trata del numerador y el denominador es uno
55        //cout << "Vector de string: " << palabras[0] << "/1"  << endl;
56
57        num = stringToBigInteger(palabras[0]);
58        den = BigInteger(1);
59      }
60    }
61    catch (exception  e)
62    {
63      cout << e.what();
64    }
65  }
66
67  /*
68    Devuelve un BigRational a partir de su numerador y con denominador igual a 1
69  */
70  BigRational::BigRational(int numerator) {
71    num = BigInteger(numerator);
72    den = BigInteger(1);
73  }
74
75  /*
76    Devuelve una representacion en formato string de un BigRational.
77    Si el denominador es 1 devuelve tan solo en numerador
78    Sino devuelve una cadena en formato num/den (siendo num y den enteros)
79  */
80  string BigRational::toString() {
81    //string  r = "";  //string que representa el BigRational
82
83    std::stringstream sstm;
84
85    //char result[100000]; //buffer para convertir a cadena el resultado
86    BigInteger one = BigInteger(1); //BigInteger con valor uno
87
88    if (num == 0)
89    {
90      sstm << "0";
91      return sstm.str(); //devuelve directamente el valor cero
92    }
93
94    //SE SUSTITUYEN LAS CONVERSIONES A ENTEROS POR UN VOLCADO STRINGSTREAM, MAS
          SEGURO
95    switch (den.compareTo(one)) {
96    case 0:
97      //sprintf_s(result, 100000, "%d", num.toInt());
98      sstm << num;
99      break;
100   default:
101     //sprintf_s(result, 100000, "%d/%d", num.toInt(), den.toInt());
102     sstm << num << "/" << den;
103     //r = result;
104     break;
105   }
106
107   //devuelvo el string del stream
108   return sstm.str();
109 }
110
111 /*
112   Instanciador de la clase BigRational para el constructor con dos parametros
          BigInteger
113 */
114 void BigRational::init(BigInteger numerator, BigInteger denominator) {
```

```
115   //deal with x / 0
116
117   if (denominator.isZero()) {
118     throw std::invalid_argument("Denominator is zero");
119   }
120
121   //reduce fraction!
122   BigInteger McD = gcd(numerator, denominator);
123   //DEBUG: cout << "GCD(" << numerator << ", " << denominator << ") : " << McD
          << endl;
124
125   num = (BigInteger) (numerator/McD);
126   den = (BigInteger) (denominator/McD);
127
128   BigInteger zero = BigInteger(0);
129
130   //to ensure invariant that denominator is positive
131   if (den.compareTo(zero) < 0)
132   {
133     den.negate(den);
134     num.negate(num);
135   }
136 }
137
138 /*
139   Constructor a partir de un numerador y denominador de tipo BigInteger
140 */
141 BigRational::BigRational(BigInteger numerator, BigInteger denominator) {
142   init(numerator, denominator);
143 }
144
145 /*
146   Comparador de instancias de la clase BigRational
147   Devuelve si una instancia BigRational a es igual a una segunda instancia b
148   Para ello ambas deben tener identico numerador, denominador y signo
149   Se comprueba multiplicando en numerador de a por el denominador de b y el
          denominador de a por el numerador de b
150   Si ambos resultados coinciden se determina que a y b son iguales
151 */
152 int BigRational::compareTo(const BigRational &b){
153   BigInteger a = num;
154   a.multiply(a,  b.den);
155   BigInteger c = den;
156   c.multiply(c, b.num);
157
158   return a.compareTo(c);
159 }
160
161 /*
162   Determina si la instancia de la clase BigRational es igual a cero
163 */
164 bool BigRational::isZero() {
165   BigRational zero = BigRational(0);
166   if (compareTo(zero) == 0) return true;
167   return false;
168 }
169
170 /*
171   Determina si el signo de la instancia BigRational es positivo
172 */
173 bool BigRational::isPositive() {
174   BigRational zero = BigRational(0);
175   if (compareTo(zero) > 0) return true;
176   return false;
```

```
177 }
178
179 /*
180   Determina si el signo de la instancia BigRational es negativo
181 */
182 bool BigRational::isNegative() {
183   BigRational zero = BigRational(0);
184   if (compareTo(zero) < 0) return true;
185   return false;
186 }
187
188 /*
189   Determina si una instancia de la clase BigRational es equivalente a otro
          objeto de una clase cualquiera
190   Para ello se realiza un cast dinamico del objeto a un puntero de clase
          BigRational y se comparan ambos objetos con el metodo compareTo de la
          clase BigRational
191   Este metodo SOLO funciona correctamente si object es de la clase BigRational.
           De otro modo el cast falla y obtenemos una excepcion
192 */
193 bool BigRational::equals(void* object) {
194   try
195   {
196     if (object == NULL) return false;
197     BigRational *b = static_cast<BigRational *> (object);
198     return compareTo(*b) == 0;
199   }
200   catch (exception  &e)
201   {
202     cout << e.what();
203   }
204   return false;
205 }
206
207 //return a * b
208 BigRational BigRational::times(BigRational b) {
209   BigInteger numerator = num;
210   BigInteger denominator = den;
211   numerator.multiply(numerator, b.num);
212   denominator.multiply(denominator, b.den);
213   BigRational result = BigRational(numerator, denominator);
214   return result;
215 }
216
217 //return a + b
218 BigRational BigRational::plus(BigRational b) {
219   BigInteger numerator, aux;
220   BigInteger denominator;
221
222   //numerator
223   numerator.multiply(num ,b.den); //num de a queda multiplicado por den de b
224   aux.multiply(b.num, den);
225   numerator.add(numerator, aux);
226   //denominator
227   denominator.multiply(den, b.den);
228   BigRational result = BigRational(numerator , denominator);
229   return result;
230 }
231
232 //return -a | Invierte el signo de un BigRational!!
233 //necesitamos dos variables auxiliares para copiar el BigRational y cambiarlo
      de signo sin alterar el objeto original
234 //*a es un puntero al puntero this y b es una copia de *a, cambiando el signo
      de b no cambiamos a this
```

```
235 BigRational BigRational::negate() {
236   BigRational *a = this;
237   BigRational b = *a;
238   b.num.negate(b.num);
239   return b;
240 }
241
242 //return a - b
243 // La llamada a esta funcion produce una excepcion y la interrupcion del
      programa
244 // La excepcion solo se produce si b es mas grande que this y por lo tanto
      result < 0 (negativo)
245 BigRational BigRational::minus(BigRational b) {
246   BigRational b_neg = b.negate();
247   BigRational result = plus(b_neg);
248   return result;
249 }
250
251 //return 1 / a
252 BigRational BigRational::reciprocal() {
253   return BigRational(den, num);
254 }
255
256 //return a / b
257 BigRational BigRational::divides(BigRational b) {
258   if (b.isZero())
259   {
260     cout << "Divides by ZERO" << endl;
261     return 0;
262   }
263   else
264     return times(b.reciprocal());
265 }
266
267 // return double reprentation (within given precision)
268 double BigRational::doubleValue() {
269   MyBig numerator =  this->num.toInt(),
270        denominator = this->den.toInt(),
271        quotient = numerator / denominator;
272
273   return quotient.ToDouble();
274 }
```

CAPÍTULO 5
Conclusiones

En este capítulo incluimos un apartado en el que describimos los diferentes problemas que hemos tenido que afronar en el desarrollo del trabajo, así como las concluisones finales una vez que el trabajo a finalizado.

5.1 Consideraciones finales

Durante el desarrollo del proyecto, tras haber realizado un amplio trabajo de investigación acerca de Ada Byron, en el que hemos indagado sobre su vida y el entrono que la rodeaba, nos hemos centrado en estudiar e intentar plasmar el trabajo que realizó esta mujer en el siglo XIX. Para ello hemos tenido que estudiar el contexto histórico que la rodeaba así como entender sus avanzadas ideas e intentar llevarlas a cabo.

Si a Charles Babbage no se le hubiera ocurrido la descabellada idea de crear una “máquina pensante"basada en la máquina de Pascal y con la novedosa técnica de Jacquard, quizá a Ada jamás se le hubiera ocurrido la brillante idea de concebir un “plan"para dicha máquina; así pues el anexo que añadió Ada a la traducción de Menabrea, constituye lo sustancial de su trabajo teórico y basta para considerarla razonablemente una científica genial.

La Máquina Analítica sería capaz de calcular cualquier tipo de ecuación, desde una mera incógnita hasta funciones trigonométricas como describe Ada en sus Notas. El programa informático que se ha creado intenta llevar a la realidad lo que Ada y Babbage intentaron construir en el siglo XIX, es evidente que nos hemos centrado en el Software porque la parte Hardware era imposible llevarla acabo. Con la creación de este programa hemos comprendido en profundidad la brillante idea que tuvo Ada en su época. Está claro que eligió los Números de Bernoulli ya que estaba familiarizada con sus propiedades desde bien pequeña. En cierto modo debemos agradecerle a lady Byron el empeño en que depositó en que su hija estudiara matemáticas porque de no haber sido así, tal vez nunca hubiera contribuido tan ampliamente al campo de la informática.

Sin embargo, Ada nunca imaginó el enorme valor que la posteridad atribuiría a su legado intelectual: en la época de los ordenadores personales y los Smartphone, es fácil olvidar que, cuando se sumió en la dolorosa inconsciencia que supuso el principio de su muerte, no sospechaba siquiera que su idea de lo que podía ser y hacer una computadora acabaría convirtiéndose en realidad.

Existen diversas opiniones sobre Ada y su trabajo. Algunos se inclinan porque Ada fue la primera mujer programadora de la historia, mientras que otros sostienen que sólo fue una mera codificadora, ya que ella no creó el algoritmo de los números de Bernoulli, sólo la adecuó a la Máquina Analítica. Me permitiré decir que para mí, Ada es una heroí-

na, tanto si creó como si sólo codificó el algoritmo. Para los tiempos que corrían fue muy audaz y emprendedora al intentar destacar en un mundo de hombres a sabiendas de las pocas posibilidades de las que disponía.

Sus ideas no quedaron estacandas en el siglo XIX, ya que Alan Turing se apoyó en la idea de Ada "Objeción de Lady Lovelace" que dice así: la Máquina Analítica no tiene pretensión alguna de crear algo. Ella puede hacer cualquier cosa que sepamos como ordenarle que lo ejecute. Ella puede «seguir un análisis, pero no tiene capacidad de anticipar cualesquiera relaciones o verdades» para crear su máquina.

Ada Byron tiene también un día propio en el calendario: el 16 de octubre. El día de Ada Lovelace rinde homenaje a todas aquellas mujeres del ámbito internacional que han contribuido con esfuerzo y pocas alabanzas en el campo de la ciencia, la tecnología, la ingeniería y las matemáticas.

En el año 2010 se realizó la filmación de una película de la cual es su protagonista, su título es *Enchantress of Numbers* (La encantadora de números). También se ha creado en premio con su nombre; es un reconocimiento de ámbito estatal, pionero a la hora de visualizar el trabajo de las mujeres que aportan avances en los diferentes ámbitos de las nuevas tecnologías.

Ada, no sólo se dedicó a la ciencia y las matemáticas, realizó otras contribuciones en campos tan variados como la agricultura y la arquitectura. Se dedicó también a la divulgación, escribiendo así una crítica del primer libro que trataba de explicar la evolución, "Vestiges of the Natural History of Creation" publicado en 1844, quince años antes de la aparición de *El origen de las especies* y difundió los trabajos de Faraday.

Tal ha sido la contribución de Ada al mundo de la informática que, el Departamento de Defensa de Estados Unidos bautizó su lenguaje de programación en 1979 con el nombre de *ADA*, rindiendo así homenaje y reconocimiento a quien consideran pionera de la informática.

De esta forma, mediante la exposición de nuestro trabajo por parte del Museo de Informática de la Universitat Politècnica de València, podemos considerar que si se hubiera hecho realidad el trabajo de Ada y Babbage, el origen de los ordenadores hubiera sido el siglo XIX, cien años antes. El programa que hemos diseñado y ejecutado contribuye a difundir el patrimonio cultural informático a lo largo de la historia, siendo este un punto de partida.

5.2 Contratiempos y problemas sufridos

A lo largo del desarrollo del trabajo hemos sufrido contratiempos y problemas tanto a la ahora de realizar implementaciones software como a la hora de entender y estudiar el funcionamiento de los Números de Bernoulli y las Notas de Ada. Veámos cada uno de ellos.

5.2.1. Librería BigInteger

Uno de los problemas que más trabajo nos ha llevado ha sido el tener que trabajar con números racionales, ya que para poder efectuar el cálculo de los Números de Bernoulli, y que las divisiones sean exactas, debemos operar con fracciones, y si no utilizamos la librería adecuada, el retorno de carro hace que los valores no coincidan. Para poder solventarlo, hemos tenido que usar la librería BigInteger, una clase que no habíamos utilizado nunca. Con esta clase podemos mandar como argumentos números grandes ya sean cadenas o datos de tipo Long, o simplemente leer desde teclado.

Para ello, debemos importar la librería correspondiente:

```
import java.math.BigInteger;
```

5.2.2. Clase Comparable en C++

Este tipo de lenguaje no posee la Clase Comparable como tal, como la pueden tener otros lenguaje como Java, por ejemplo. Entonces he tenido que crear desde cero toda la clase y nos ha llevado más tiempo del que teniamos pensado.

5.2.3. Números de Bernoulli

Cuando comenzamos el desarrollo del trabajo, fue un poco tedioso y costo llegar a entender el funcionamiento de dichos números, ya que creíamos que nos faltaba la ecuación para poder calcularlos, hasta que llegamos al apartado que corresponde a la **Nota G** y nos dimos cuenta que la función se encuentra en la primera operación a realizar que posee la tabla que Ada creó.

5.2.4. Clase BigDecimal

A la hora de implementar esta clase estándar nos hemos dado cuenta de que no existe, y que es necesario que nos descarguemos otra librería. Se trata de la librería TTMath. Esta librería nos permite a realizar operaciones aritméticas con enteros grandes sin signo, enteros con signo y números grandes en coma flotante. Proporciona operaciones matemáticas estándar como son las sumas, restas, multiplicaciones o divisiones. Esta librería la necesitamos para la función que devuelve el valor doble del racional:

```
#include <stdio.h>
#include <stdlib.h>
 // return double reprentation (within given precision)
    public double doubleValue() {//
        int SCALE = 32;         // number of digits after the decimal place
        BigDecimal numerator   = new BigDecimal(num);
        BigDecimal denominator = new BigDecimal(den);
        BigDecimal quotient    = numerator.divide(denominator, SCALE,
            RoundingMode.HALF_EVEN);
        return quotient.doubleValue();
    }
```

Mediante el uso de esta librería el código correspondiente queda de la siguiente manera:

```
#include <stdio.h>
#include <stdlib.h>
// return double reprentation (within given precision)
double BigRational::doubleValue() {
    MyBig     numerator,
              denominator;

    numerator = this->num.toInt();
    denominator = this->den.toInt();

    MyBig quotient = numerator / denominator;

    return quotient.ToDouble();
}
```

La precisión se establece en la definición del tipo MyBig al principio del programa:

```
#include <stdio.h>
#include <stdlib.h>
// for convenience we're defining MyBig type
// this type has 2 words for its mantissa and 1 word for its exponent
// (on a 32bit platform one word means a word of 32 bits,
// and on a 64bit platform one word means a word of 64 bits)

// Big<exponent, mantissa>
typedef ttmath::Big<TTMATH_BITS(32), TTMATH_BITS(64)> MyBig;
```

5.2.5. Función que calcula el Máximo Común Divisor

Otro problema que hemos tenido está en la función que calcula el Máximo Común Divisor entre el numerador y el denominador cuando ambos son BigInteger (enteros grandes).

Por ejemplo para 500.000.000/600.000.000.000 el resultado debería ser 500.000 y por tanto convertir el BigRational a 1/120.000.000. Sin embargo el Máximo Común Divisor nos da como resultado 4, error.

La función init donde se produce el error es:

```

void BigRational::init(BigInteger numerator, BigInteger denominator) {
    //deal with x / 0

    if (denominator.isZero()) {
        throw std::invalid_argument("Denominator is zero");
    }

    BigUnsigned n = numerator.toUnsignedLong();
    BigUnsigned d = denominator.toUnsignedLong();
    //reduce fraction
    //Falla al calcular el Maximo Comun Divisor de numerador y denominador
    BigUnsigned g = gcd(numerator.toUnsignedInt, denominator.toUnsignedInt);

    cout << "GCD(" << numerator << ", " << denominator << ") : " << g << endl;

   //A partir de aqui se produce el error
   //num = (numerator/g);
    //den = (denominator/g);

    cout << "numerator: " << numerator << endl;
    cout << "denominator: " << denominator << endl;

    BigUnsigned trash;
    num.divideWithRemainder(g, trash);
    den.divideWithRemainder(g, trash);

    BigInteger zero = BigInteger(0);

    //to ensure invariant that denominator is positive
    if (den.compareTo(zero) < 0)
    {
        den.negate(den);
        num.negate(num);
    }
}
```

5.2.6. BigRational

Esta función es la encargada de realizar la resta entre dos números BigRational, cuya llamada sería:

```
x = BigRational(1,6);
y = BigRational(4,8);
z = x.minus(y);
```

Cuando intentaba ejecutarla me salía un error como vemos en la Figura 5.1. Cuyo error

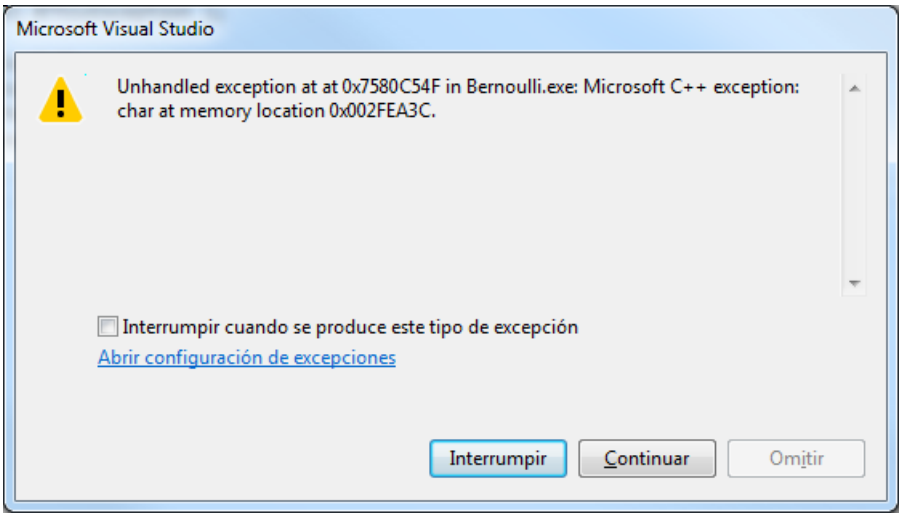

Figura 5.1: Error.

se encuentra en en la funcion init que crea un BigRational a partir de dos BigInteger:

```
void BigRational::init(BigInteger numerator, BigInteger denominator) {
    //deal with x / 0

    if (denominator.isZero()) {
        throw std::invalid_argument("Denominator is zero");
    }

    //reduce fraction
    BigInteger g = gcd(numerator.toUnsignedInt(),denominator.toUnsignedInt());
    num = (numerator/g);
    den = (denominator/g);

    BigInteger zero = BigInteger(0);

    //to ensure invariant that denominator is positive
    if (den.compareTo(zero) < 0)
    {
        den.negate(den);
        num.negate(num);
    }
}

/*
    Constructor a partir de un numerador y denominador de tipo BigInteger
*/
BigRational::BigRational(BigInteger numerator, BigInteger denominator) {
    init(numerator, denominator);
}
```

Bibliografía

[1] Allan Bromley . *The Babbage Papers in the Science Museum*. The Science Museum, Londres, 1991.

[2] Benita Eisler. *Byron*. Hamish Hamilton, Londres, 1999.

[3] Betty Alexandra Toole. *Ada, the Enchantress of Numbers*. Strawberry Press, Mill Valley (California), 1992.

[4] Betty Alexandra Toole. Ada Byron, Lady Lovelace, An Analyst and Metaphysician. *IEEE Annals of the History of Computing*, 1058-618096, octubre, 1996.

[5] Carlos Sánchez Fernández. *Los Bernoulli*. Nivola Libros y Ediciones S.L, Tres Cantos (Madrid), 2001.

[6] Charles Babbage. *Passage from the Life of a Philposopher*. Ediciones disponibles: Rutgers University Press, Nuevo Brunwick (Nueva Jersey) y IEEE Press, Piscataway (Nueva Jersey), 1994.

[7] Charles Babbage. *Science and Reform: Selected Works of Charles Babbage*. Cambridge University Press, Cambridge, 1989.

[8] Denise Gürer. Pioneering Women in Computer Science. *Communications of the ACM*, junio, 2002.

[9] Donald Brown. *Charles Babbage: The Man and His Machine*. The Totnes Museum Study Centre, Totnes, 1992.

[10] Doris Langley Moore. *Ada, Countess of Lovelace: Byron's Legitimate Daughter*. John Murray, Londres, 1977.

[11] Doron Swade. *Charles Babbage and His Calculating Engines*. The Science Museum, Londres, 1991.

[12] Dorothy Stein. *Ada: A Life and Legacy*. MIT Press, Cambridge (Massachusetts), 1985.

[13] Eugene Eric Kim y Betty Toole. Ada and the First Computer. *Scientific American*, mayo, 1999.

[14] James Essinger. *El Algoritmo de Ada*. Alba Trayectos, Sant Llorenç d'Hortons (Barcelona), 2015.

[15] Joasia Krysa. *Ada Lovelace: Introduction*. Documenta 13, Alemania, 2011.

[16] John Fuegi and Jo Francis. Lovelace, Babbage and the Creation of the 1843 Notes. *IEEE Annals of the History of Computing*, 1058-618003, octubre, 2003.

[17] Lord Byron. *The Works of Lord Byron*. Wordsworth Editions, Ware, 1994.

[18] Lucy Lethbridge. *Ada Lovelace: Computer Wizard of Victorian England*. Short Books, Londres, 2004.

[19] Luigi Federico Menabrea. Sketch of The Analitical Engine Invented by Charles Babbage. Consultado en https://www.fourmilab.ch/babbage/sketch.html.

[20] Michael R. Williams. *A History of Computing Technology*. Viley-IEEE Computer Society(California), 1992.

[21] Thomas Haigh y Mark Priestley. Historical Reflections Innovators Assemble: Ada Lovelace, Walter Isaccson, and the Superheroines of Computing. *Communications of the ACM*, 10.1145-2804228, septiembre, 2015.

[22] Xavier Molero Prieto. Las mujeres en la profesión informática: historia, actualidad y retos para el futuro. *Novática*, enero-marzo, 2015.

[23] Walter Isaacson. *Los Innovadores: los genios que inventaron el futuro*. Debate, Madrid, 2014.

APÉNDICE A
Configuración del sistema de programación

Cómo ya mencionamos anteriormente, crear un código en lenguaje C++ es más complejo que en Java ya que necesitamos tener tres tipos de archivos. En este capítulo visualizamos los códigos .cc y .h, la clase Extrafuntion y la clase ttmath. Así como la instalación paso a paso de Visual Studio para poder crear, compilar y ejecutar el código en C++.

A.1 Fase de inicialización: Visual Studio

Para poder desarrollar nuestro trabajo hemos tenido que instalar el Visual Studio en nuestro ordenador y crear un nuevo proyecto. Veámoslo paso a paso.

El primer paso que debemos realizar una vez dentro del Visual Studio para poder crear nuestro proyecto es establecer una aplicación desde consola como podemos ver en la Imagen A.1.

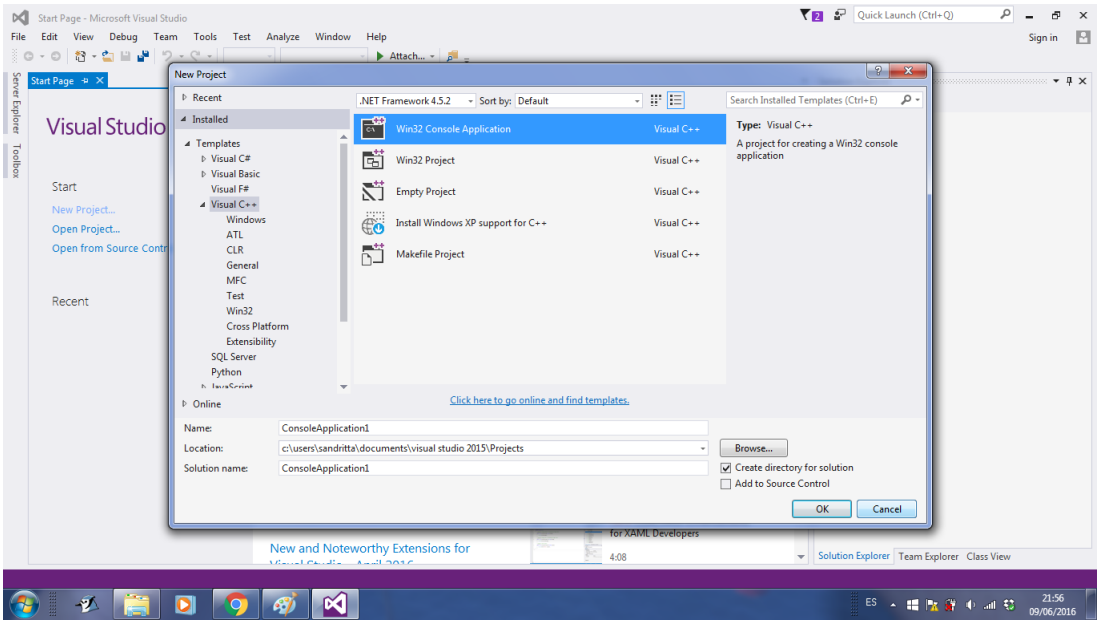

Figura A.1: Aplicación de consola.

A continuación, seguimos los pasos como se muestra en la Imagen A.2.

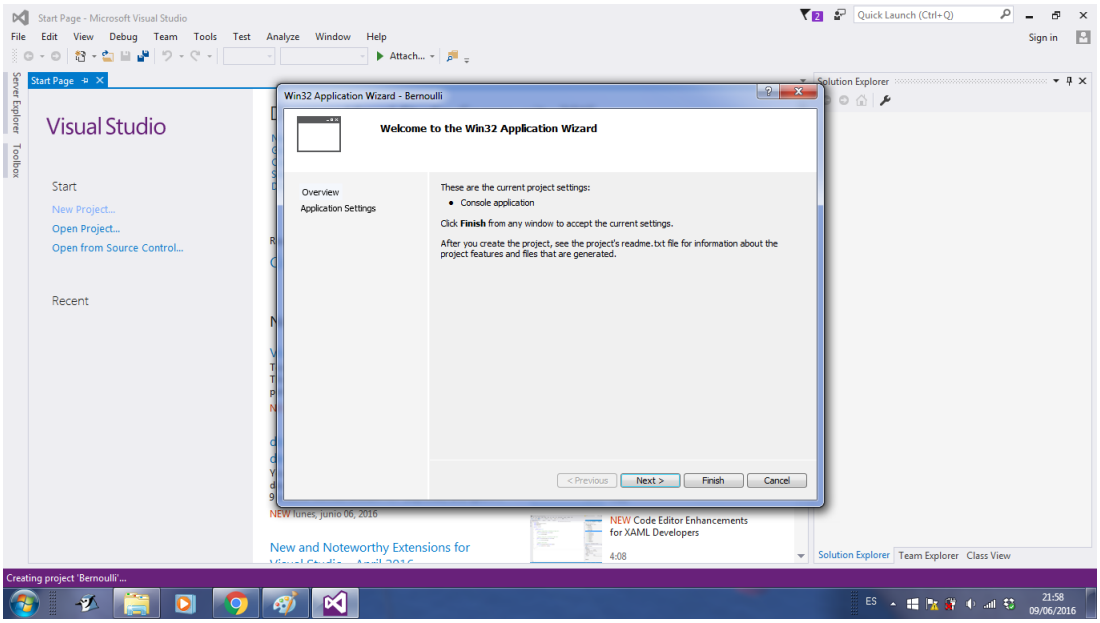

Figura A.2: Inicio.

Elegimos los parámetros de entrada, Imágen A.3.

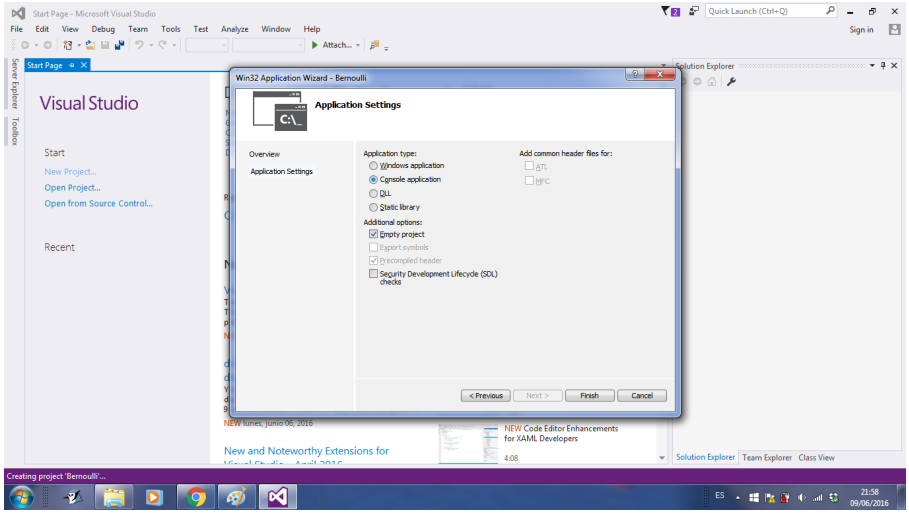

Figura A.3: Parámetros de entrada.

Una vez hecho esto, ya tenemos el entorno configurado. Es hora de crear los ficheros como vemos en la Imágen A.4.

Para poder añadirlos, debemos poner el ratón encima y hacer click derecho en .Add z "New Item", como vemos en la Imágen A.5.

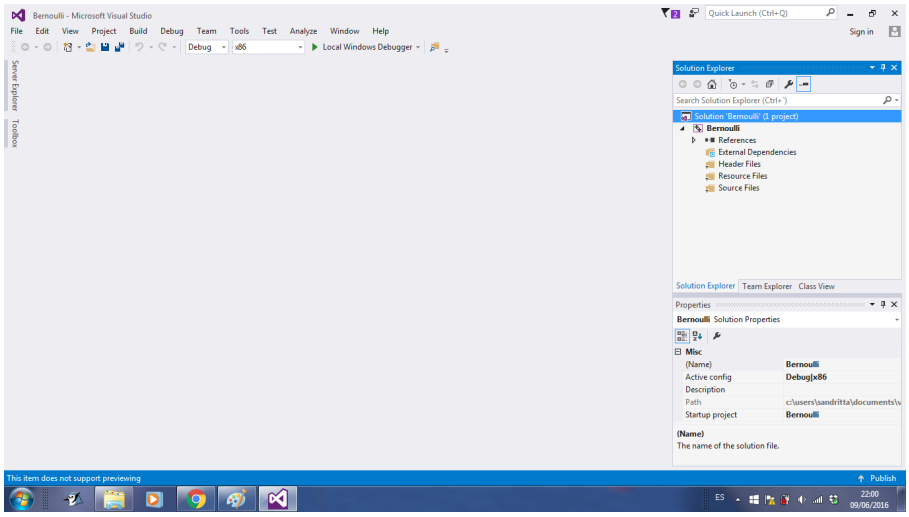

Figura A.4: Creación de ficheros

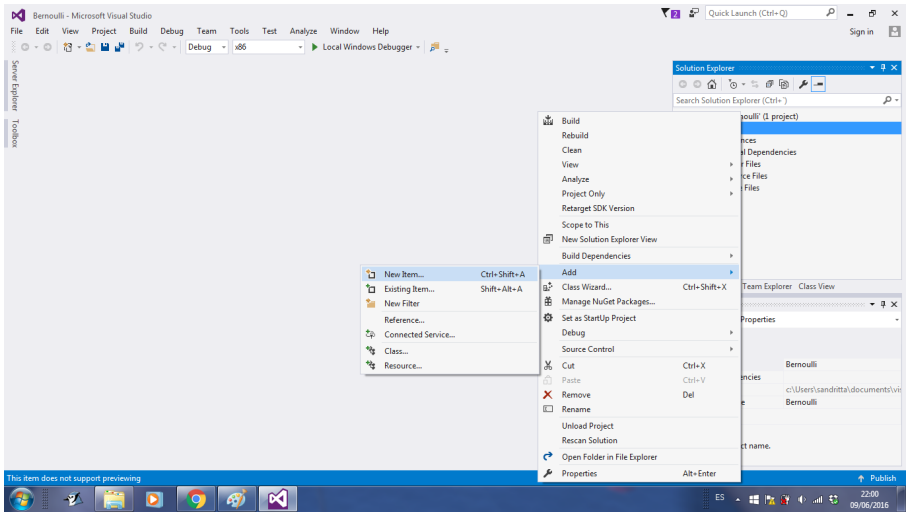

Figura A.5: Añadir nuevo proyecto

A continuación, creamos el fichero C++ como podemos ver en la Imágen A.6

Es el turno ahora de añadir las librerías. Pinchamos sobre «Header Files» botón derecho «Add» y «New Item», como vemos en la imágen A.7.

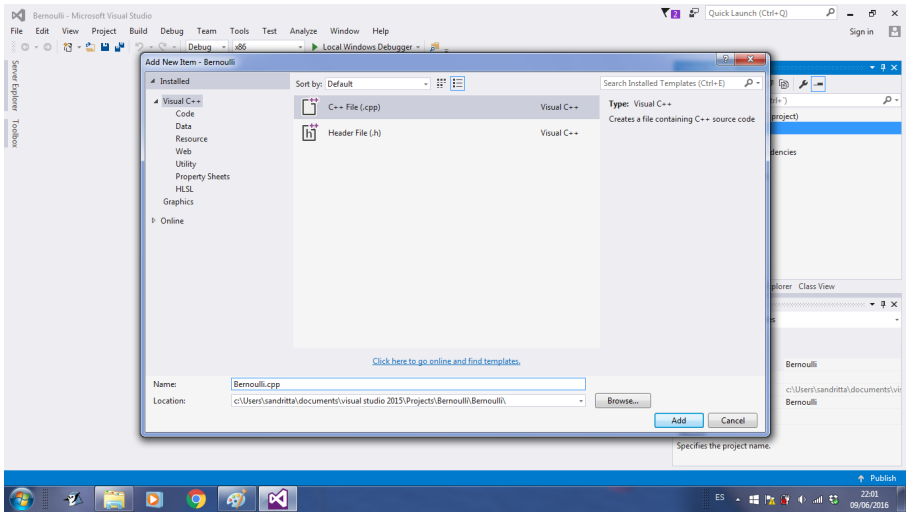

Figura A.6: Nuevo proyecto.

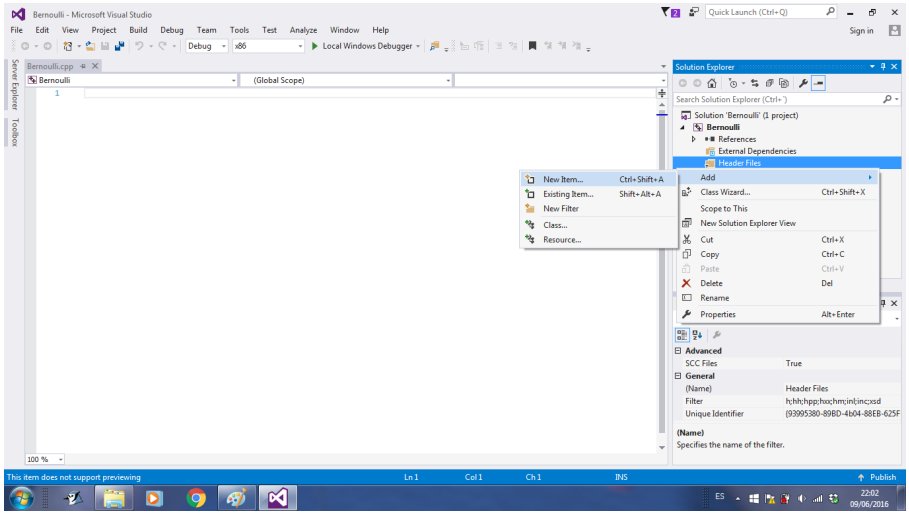

Figura A.7: Añadir libreria.

Lamamos al nuevo archivo como BigRational.h y ya lo tenemos listo para poder trabajar, como se observa en la Imágen A.8

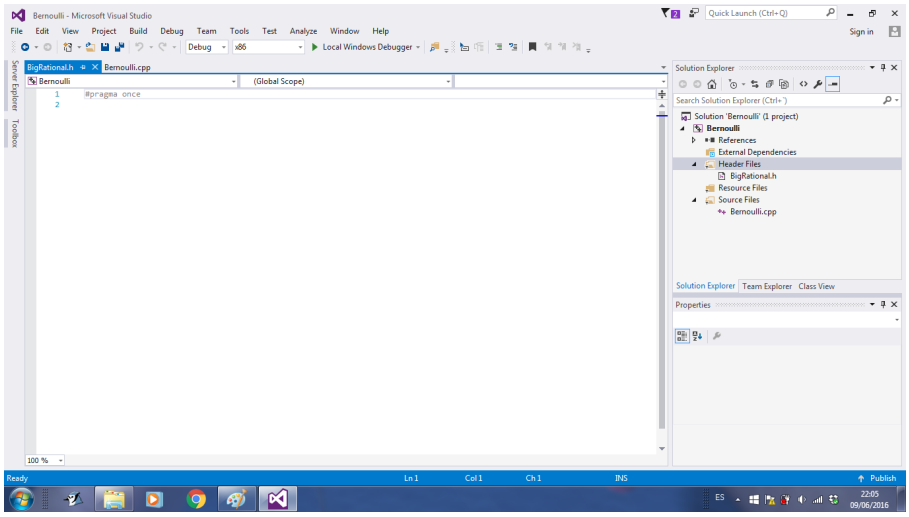

Figura A.8: Nuevo proyecto creado.

Seguidamente, añadimos las librerías adicionales que necesitamos para poder crear nuestro código: BigInteguer y ttmath, como se muestra en las Imágenes A.9 y A.10.

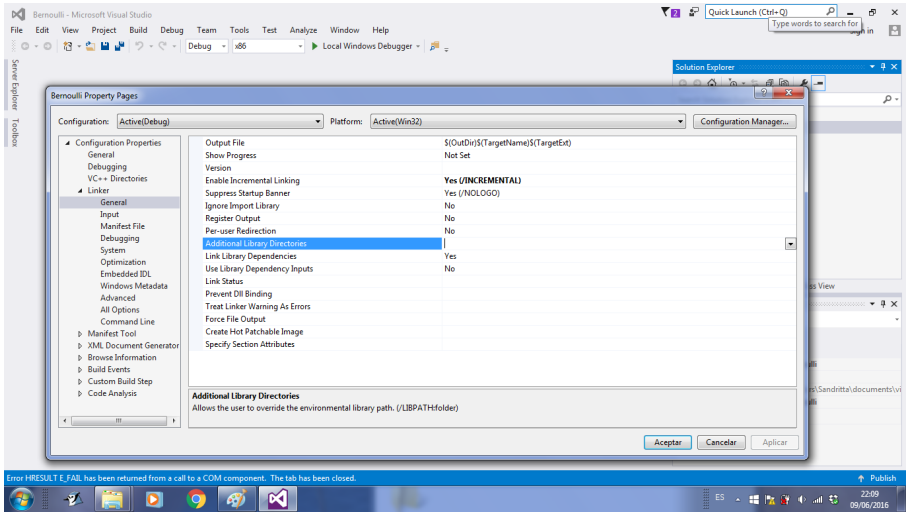

Figura A.9: Añadir librerías adicionales.

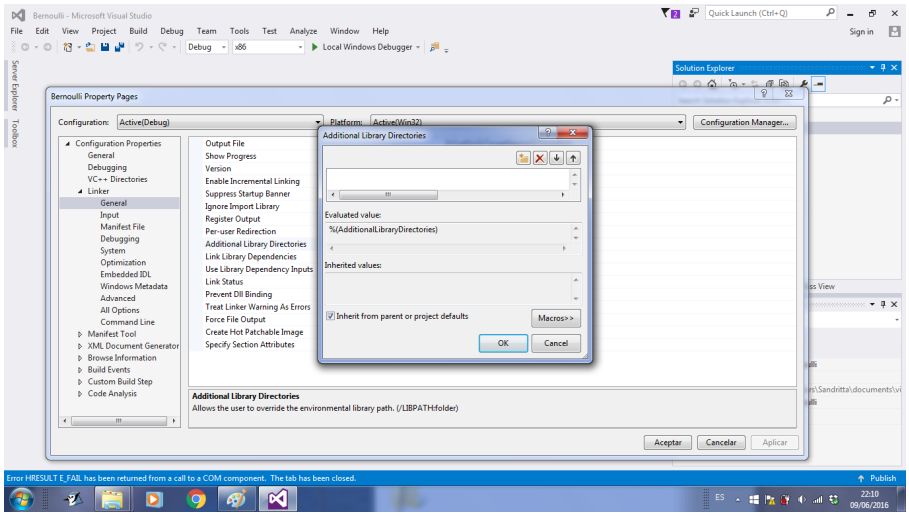

Figura A.10: Librerías adicionales.

Una vez hecho todo lo anterior, creamos nuestro código y nos disponemos a compilarlo para ver si es correcto o hemos cometido algún fallo, como podemos encontrar en la Imágen A.11.

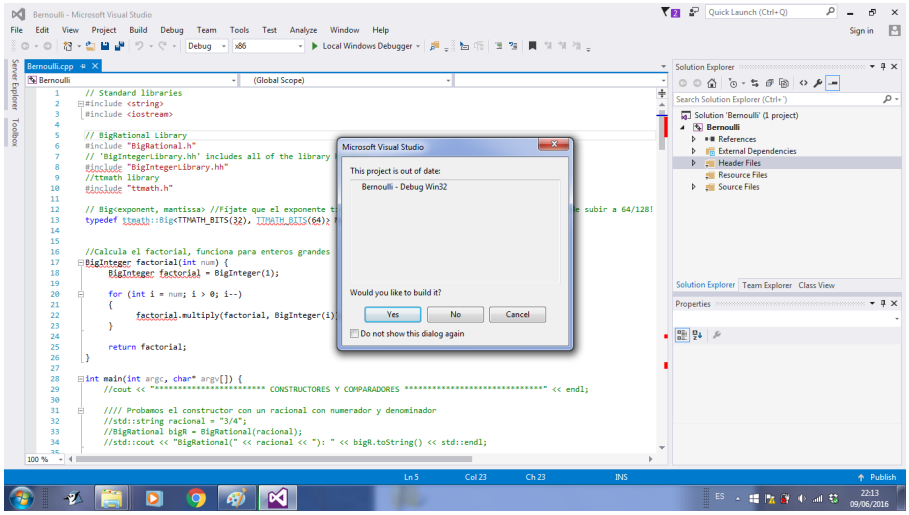

Figura A.11: Compilación.

Al introducir las librerías por separado se producían unos errores que no supe solventar, asíque decidí introducirlas tal cual, Imágen A.12.

Probamos a compilar, cuyo resultado es correcto, como se muestra en la Imágen A.13.

A continuación, creamos un ejecutable de prueba (Carpeta Debug) para comprobar el correcto funcionamiento. Imágen A.14.

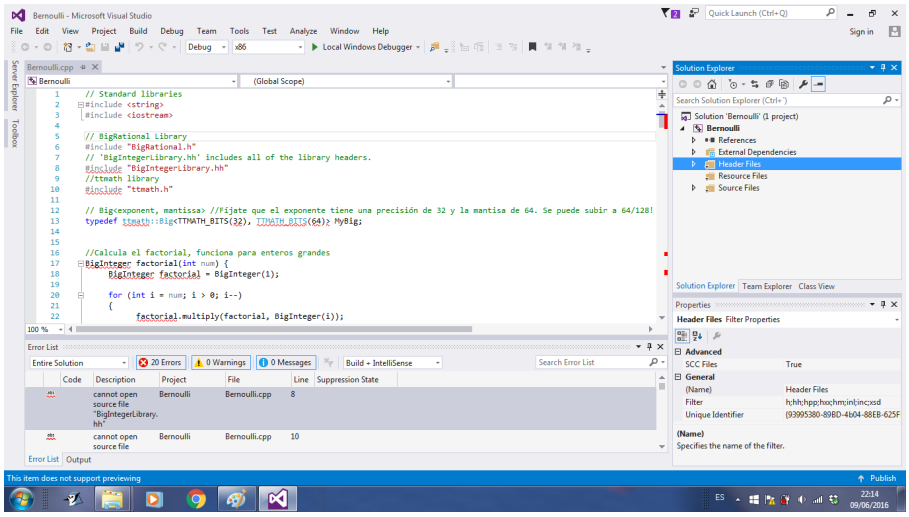

Figura A.12: Librerías.

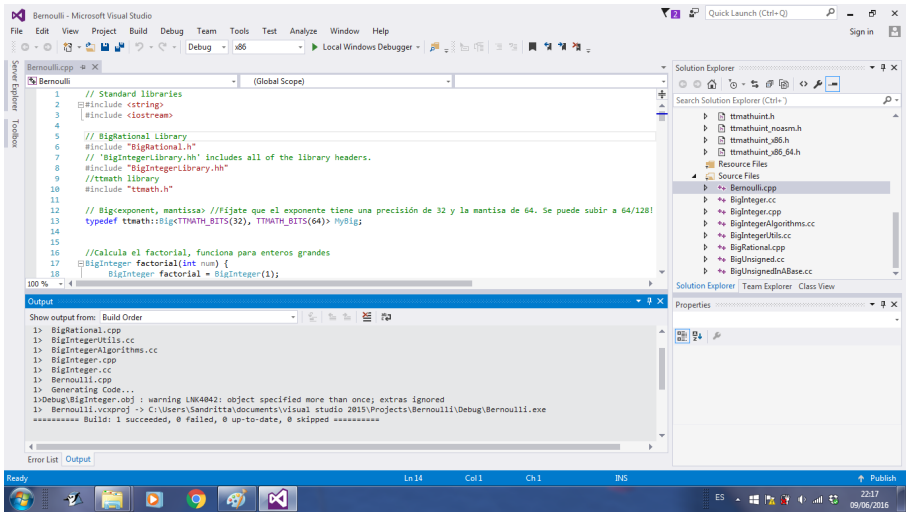

Figura A.13: Compilación correcta.

Compilamos nuevamente para obtener la versión definitiva (Carpeta Release) como podemos observar en la Imágen A.15.

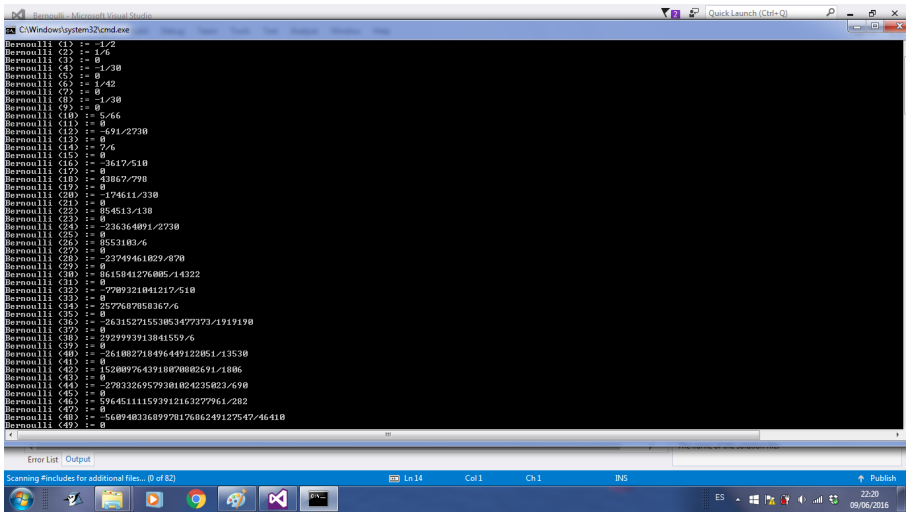

Figura A.14: Ejecución de programa.

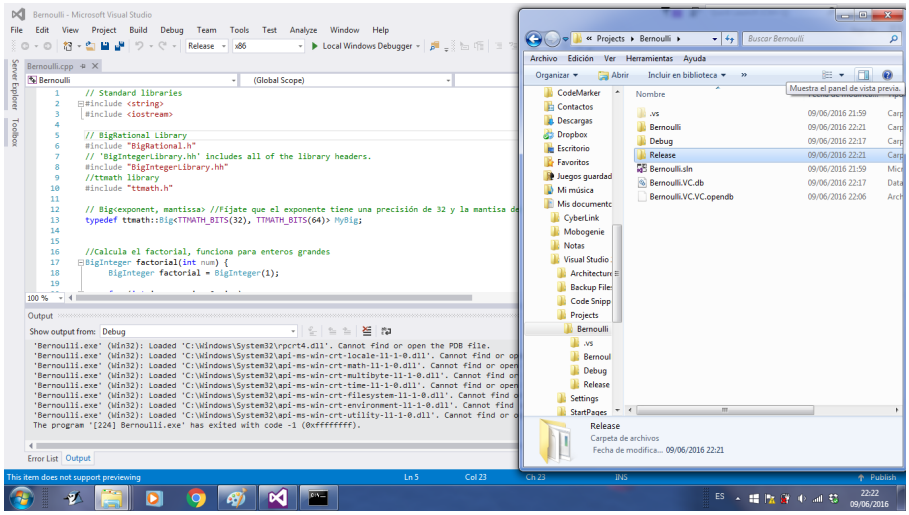

Figura A.15: Versión definitiva.

A.2 Códigos

En esta sección podemos encontrar los arvhivos .cc y .h pertenecientes a nuestro código en C++, así como las funciones ttmath y Extrafuntion necesarias para que todo el conjunto funcione correctamente.

Empezaremos por los archivos .cc:

- **BigInteger.cc**:

```
#include "BigInteger.hh"

void BigInteger::operator =(const BigInteger &x) {
  // Calls like a = a have no effect
  if (this == &x)
    return;
  // Copy sign
  sign = x.sign;
  // Copy the rest
  mag = x.mag;
}

```

```
13 BigInteger::BigInteger(const Blk *b, Index blen, Sign s) : mag(b, blen) {
14   switch (s) {
15   case zero:
16     if (!mag.isZero())
17       throw "BigInteger::BigInteger(const Blk *, Index, Sign): Cannot use a
             sign of zero with a nonzero magnitude";
18     sign = zero;
19     break;
20   case positive:
21   case negative:
22     // If the magnitude is zero, force the sign to zero.
23     sign = mag.isZero() ? zero : s;
24     break;
25   default:
26     /* g++ seems to be optimizing out this case on the assumption
27      * that the sign is a valid member of the enumeration.  Oh well. */
28     throw "BigInteger::BigInteger(const Blk *, Index, Sign): Invalid sign";
29   }
30 }
31
32 BigInteger::BigInteger(const BigUnsigned &x, Sign s) : mag(x) {
33   switch (s) {
34   case zero:
35     if (!mag.isZero())
36       throw "BigInteger::BigInteger(const BigUnsigned &, Sign): Cannot use a
             sign of zero with a nonzero magnitude";
37     sign = zero;
38     break;
39   case positive:
40   case negative:
41     // If the magnitude is zero, force the sign to zero.
42     sign = mag.isZero() ? zero : s;
43     break;
44   default:
45     /* g++ seems to be optimizing out this case on the assumption
46      * that the sign is a valid member of the enumeration.  Oh well. */
47     throw "BigInteger::BigInteger(const BigUnsigned &, Sign): Invalid sign";
48   }
49 }
50
51 /* CONSTRUCTION FROM PRIMITIVE INTEGERS
52  * Same idea as in BigUnsigned.cc, except that negative input results in a
53  * negative BigInteger instead of an exception. */
54
55 // Done longhand to let us use initialization.
56 BigInteger::BigInteger(unsigned long  x) : mag(x) { sign = mag.isZero() ? zero
       : positive; }
57 BigInteger::BigInteger(unsigned int   x) : mag(x) { sign = mag.isZero() ? zero
       : positive; }
58 BigInteger::BigInteger(unsigned short x) : mag(x) { sign = mag.isZero() ? zero
       : positive; }
59
60 // For signed input, determine the desired magnitude and sign separately.
61
62 namespace {
63   template <class X, class UX>
64   BigInteger::Blk magOf(X x) {
65     /* UX(...) cast needed to stop short(-2^15), which negates to
66      * itself, from sign-extending in the conversion to Blk. */
67     return BigInteger::Blk(x < 0 ? UX(-x) : x);
68   }
69   template <class X>
70   BigInteger::Sign signOf(X x) {
71     return (x == 0) ? BigInteger::zero
```

```
72        : (x > 0) ? BigInteger::positive
73        : BigInteger::negative;
74    }
75  }
76
77  BigInteger::BigInteger(long  x) : sign(signOf(x)), mag(magOf<long , unsigned
      long >(x)) {}
78  BigInteger::BigInteger(int   x) : sign(signOf(x)), mag(magOf<int  , unsigned
      int  >(x)) {}
79  BigInteger::BigInteger(short x) : sign(signOf(x)), mag(magOf<short, unsigned
      short>(x)) {}
80
81  // CONVERSION TO PRIMITIVE INTEGERS
82
83  /* Reuse BigUnsigned's conversion to an unsigned primitive integer.
84   * The friend is a separate function rather than
85   * BigInteger::convertToUnsignedPrimitive to avoid requiring BigUnsigned to
86   * declare BigInteger. */
87  template <class X>
88  inline X convertBigUnsignedToPrimitiveAccess(const BigUnsigned &a) {
89    return a.convertToPrimitive<X>();
90  }
91
92  template <class X>
93  X BigInteger::convertToUnsignedPrimitive() const {
94    if (sign == negative)
95      throw "BigInteger::to<Primitive>: "
96        "Cannot convert a negative integer to an unsigned type";
97    else
98      return convertBigUnsignedToPrimitiveAccess<X>(mag);
99  }
100
101 /* Similar to BigUnsigned::convertToPrimitive, but split into two cases for
102  * nonnegative and negative numbers. */
103 template <class X, class UX>
104 X BigInteger::convertToSignedPrimitive() const {
105   if (sign == zero)
106     return 0;
107   else if (mag.getLength() == 1) {
108     // The single block might fit in an X.  Try the conversion.
109     Blk b = mag.getBlock(0);
110     if (sign == positive) {
111       X x = X(b);
112       if (x >= 0 && Blk(x) == b)
113         return x;
114     } else {
115       X x = -X(b);
116       /* UX(...) needed to avoid rejecting conversion of
117        * -2^15 to a short. */
118       if (x < 0 && Blk(UX(-x)) == b)
119         return x;
120     }
121     // Otherwise fall through.
122   }
123   throw "BigInteger::to<Primitive>: "
124     "Value is too big to fit in the requested type";
125 }
126
127 unsigned long  BigInteger::toUnsignedLong () const { return
      convertToUnsignedPrimitive<unsigned long >       (); }
128 unsigned int   BigInteger::toUnsignedInt  () const { return
      convertToUnsignedPrimitive<unsigned int  >       (); }
129 unsigned short BigInteger::toUnsignedShort() const { return
      convertToUnsignedPrimitive<unsigned short>       (); }
```

```
130 long           BigInteger::toLong         () const { return
        convertToSignedPrimitive  <long , unsigned long> (); }
131 int            BigInteger::toInt          () const { return
        convertToSignedPrimitive  <int  , unsigned int>  (); }
132 short          BigInteger::toShort        () const { return
        convertToSignedPrimitive  <short, unsigned short>(); }
133
134 // COMPARISON
135 BigInteger::CmpRes BigInteger::compareTo(const BigInteger &x) const {
136   // A greater sign implies a greater number
137   if (sign < x.sign)
138     return less;
139   else if (sign > x.sign)
140     return greater;
141   else switch (sign) {
142     // If the signs are the same...
143   case zero:
144     return equal; // Two zeros are equal
145   case positive:
146     // Compare the magnitudes
147     return mag.compareTo(x.mag);
148   case negative:
149     // Compare the magnitudes, but return the opposite result
150     return CmpRes(-mag.compareTo(x.mag));
151   default:
152     throw "BigInteger internal error";
153   }
154 }
155
156 /* COPY-LESS OPERATIONS
157  * These do some messing around to determine the sign of the result,
158  * then call one of BigUnsigned's copy-less operations. */
159
160 // See remarks about aliased calls in BigUnsigned.cc .
161 #define DTRT_ALIASED(cond, op) \
162   if (cond) { \
163     BigInteger tmpThis; \
164     tmpThis.op; \
165     *this = tmpThis; \
166     return; \
167   }
168
169 void BigInteger::add(const BigInteger &a, const BigInteger &b) {
170   DTRT_ALIASED(this == &a || this == &b, add(a, b));
171   // If one argument is zero, copy the other.
172   if (a.sign == zero)
173     operator =(b);
174   else if (b.sign == zero)
175     operator =(a);
176   // If the arguments have the same sign, take the
177   // common sign and add their magnitudes.
178   else if (a.sign == b.sign) {
179     sign = a.sign;
180     mag.add(a.mag, b.mag);
181   } else {
182     // Otherwise, their magnitudes must be compared.
183     switch (a.mag.compareTo(b.mag)) {
184     case equal:
185       // If their magnitudes are the same, copy zero.
186       mag = 0;
187       sign = zero;
188       break;
189       // Otherwise, take the sign of the greater, and subtract
190       // the lesser magnitude from the greater magnitude.
```

```
191       case greater:
192         sign = a.sign;
193         mag.subtract(a.mag, b.mag);
194         break;
195       case less:
196         sign = b.sign;
197         mag.subtract(b.mag, a.mag);
198         break;
199       }
200     }
201   }
202
203   void BigInteger::subtract(const BigInteger &a, const BigInteger &b) {
204     // Notice that this routine is identical to BigInteger::add,
205     // if one replaces b.sign by its opposite.
206     DTRT_ALIASED(this == &a || this == &b, subtract(a, b));
207     // If a is zero, copy b and flip its sign.  If b is zero, copy a.
208     if (a.sign == zero) {
209       mag = b.mag;
210       // Take the negative of _b_'s, sign, not ours.
211       // Bug pointed out by Sam Larkin on 2005.03.30.
212       sign = Sign(-b.sign);
213     } else if (b.sign == zero)
214       operator =(a);
215     // If their signs differ, take a.sign and add the magnitudes.
216     else if (a.sign != b.sign) {
217       sign = a.sign;
218       mag.add(a.mag, b.mag);
219     } else {
220       // Otherwise, their magnitudes must be compared.
221       switch (a.mag.compareTo(b.mag)) {
222         // If their magnitudes are the same, copy zero.
223       case equal:
224         mag = 0;
225         sign = zero;
226         break;
227         // If a's magnitude is greater, take a.sign and
228         // subtract a from b.
229       case greater:
230         sign = a.sign;
231         mag.subtract(a.mag, b.mag);
232         break;
233         // If b's magnitude is greater, take the opposite
234         // of b.sign and subtract b from a.
235       case less:
236         sign = Sign(-b.sign);
237         mag.subtract(b.mag, a.mag);
238         break;
239       }
240     }
241   }
242
243   void BigInteger::multiply(const BigInteger &a, const BigInteger &b) {
244     DTRT_ALIASED(this == &a || this == &b, multiply(a, b));
245     // If one object is zero, copy zero and return.
246     if (a.sign == zero || b.sign == zero) {
247       sign = zero;
248       mag = 0;
249       return;
250     }
251     // If the signs of the arguments are the same, the result
252     // is positive, otherwise it is negative.
253     sign = (a.sign == b.sign) ? positive : negative;
254     // Multiply the magnitudes.
```

```
255   mag.multiply(a.mag, b.mag);
256 }
257
258 /*
259  * DIVISION WITH REMAINDER
260  * Please read the comments before the definition of
261  * 'BigUnsigned::divideWithRemainder' in 'BigUnsigned.cc' for lots of
262  * information you should know before reading this function.
263  *
264  * Following Knuth, I decree that x / y is to be
265  * 0 if y==0 and floor(real-number x / y) if y!=0.
266  * Then x %y shall be x - y*(integer x / y).
267  *
268  * Note that x = y * (x / y) + (x %y) always holds.
269  * In addition, (x %y) is from 0 to y - 1 if y > 0,
270  * and from -(|y| - 1) to 0 if y < 0.  (x %y) = x if y = 0.
271  *
272  * Examples: (q = a / b, r = a %b)
273  *  a b q r
274  *  === === === ===
275  *  4 3 1 1
276  *  -4  3 -2  2
277  *  4 -3  -2  -2
278  *  -4  -3  1 -1
279  */
280 void BigInteger::divideWithRemainder(const BigInteger &b, BigInteger &q) {
281   // Defend against aliased calls;
282   // same idea as in BigUnsigned::divideWithRemainder .
283   if (this == &q)
284     throw "BigInteger::divideWithRemainder: Cannot write quotient and remainder
          into the same variable";
285   if (this == &b || &q == &b) {
286     BigInteger tmpB(b);
287     divideWithRemainder(tmpB, q);
288     return;
289   }
290
291   // Division by zero gives quotient 0 and remainder *this
292   if (b.sign == zero) {
293     q.mag = 0;
294     q.sign = zero;
295     return;
296   }
297   // 0 / b gives quotient 0 and remainder 0
298   if (sign == zero) {
299     q.mag = 0;
300     q.sign = zero;
301     return;
302   }
303
304   // Here *this != 0, b != 0.
305
306   // Do the operands have the same sign?
307   if (sign == b.sign) {
308     // Yes: easy case.  Quotient is zero or positive.
309     q.sign = positive;
310   } else {
311     // No: harder case.  Quotient is negative.
312     q.sign = negative;
313     // Decrease the magnitude of the dividend by one.
314     mag--;
315     /*
316      * We tinker with the dividend before and with the
317      * quotient and remainder after so that the result
```

```
318       * comes out right.  To see why it works, consider the following
319       * list of examples, where A is the magnitude-decreased
320       * a, Q and R are the results of BigUnsigned division
321       * with remainder on A and |b|, and q and r are the
322       * final results we want:
323       *
324       *  a A b Q R q r
325       *  -3  -2  3 0 2 -1  0
326       *  -4  -3  3 1 0 -2  2
327       *  -5  -4  3 1 1 -2  1
328       *  -6  -5  3 1 2 -2  0
329       *
330       * It appears that we need a total of 3 corrections:
331       * Decrease the magnitude of a to get A.  Increase the
332       * magnitude of Q to get q (and make it negative).
333       * Find r = (b - 1) - R and give it the desired sign.
334       */
335     }
336
337     // Divide the magnitudes.
338     mag.divideWithRemainder(b.mag, q.mag);
339
340     if (sign != b.sign) {
341       // More for the harder case (as described):
342       // Increase the magnitude of the quotient by one.
343       q.mag++;
344       // Modify the remainder.
345       mag.subtract(b.mag, mag);
346       mag--;
347     }
348
349     // Sign of the remainder is always the sign of the divisor b.
350     sign = b.sign;
351
352     // Set signs to zero as necessary.  (Thanks David Allen!)
353     if (mag.isZero())
354       sign = zero;
355     if (q.mag.isZero())
356       q.sign = zero;
357
358     // WHEW!!!
359   }
360
361   // Negation
362   void BigInteger::negate(const BigInteger &a) {
363     DTRT_ALIASED(this == &a, negate(a));
364     // Copy a's magnitude
365     mag = a.mag;
366     // Copy the opposite of a.sign
367     sign = Sign(-a.sign);
368   }
369
370   // INCREMENT/DECREMENT OPERATORS
371
372   // Prefix increment
373   void BigInteger::operator ++() {
374     if (sign == negative) {
375       mag--;
376       if (mag == 0)
377         sign = zero;
378     } else {
379       mag++;
380       sign = positive; // if not already
381     }
```

```
382 }
383
384 // Postfix increment: same as prefix
385 void BigInteger::operator ++(int) {
386   operator ++();
387 }
388
389 // Prefix decrement
390 void BigInteger::operator --() {
391   if (sign == positive) {
392     mag--;
393     if (mag == 0)
394       sign = zero;
395   } else {
396     mag++;
397     sign = negative;
398   }
399 }
400
401 // Postfix decrement: same as prefix
402 void BigInteger::operator --(int) {
403   operator --();
404 }
```

- BigIntegerAlgorithms.cc:

```
1  #include "BigIntegerAlgorithms.hh"
2
3  BigUnsigned gcd(BigUnsigned a, BigUnsigned b) {
4    BigUnsigned trash;
5    // Neat in-place alternating technique.
6    for (;;) {
7      if (b.isZero())
8        return a;
9      a.divideWithRemainder(b, trash);
10     if (a.isZero())
11       return b;
12     b.divideWithRemainder(a, trash);
13   }
14 }
15
16 /* Esta la he creado yo para tener una funcion que me calcule el MCD de dos
       biginteger sin conversiones */
17 BigInteger gcd(BigInteger a, BigInteger b) {
18   BigInteger trash;
19   // Neat in-place alternating technique.
20   for (;;) {
21     if (b.isZero())
22       return a;
23     a.divideWithRemainder(b, trash);
24     if (a.isZero())
25       return b;
26     b.divideWithRemainder(a, trash);
27   }
28 }
29
30 void extendedEuclidean(BigInteger m, BigInteger n,
31     BigInteger &g, BigInteger &r, BigInteger &s) {
32   if (&g == &r || &g == &s || &r == &s)
33     throw "BigInteger extendedEuclidean: Outputs are aliased";
34   BigInteger r1(1), s1(0), r2(0), s2(1), q;
35   /* Invariants:
36    * r1*m(orig) + s1*n(orig) == m(current)
37    * r2*m(orig) + s2*n(orig) == n(current) */
```

```
38    for (;;) {
39      if (n.isZero()) {
40        r = r1; s = s1; g = m;
41        return;
42      }
43      // Subtract q times the second invariant from the first invariant.
44      m.divideWithRemainder(n, q);
45      r1 -= q*r2; s1 -= q*s2;
46
47      if (m.isZero()) {
48        r = r2; s = s2; g = n;
49        return;
50      }
51      // Subtract q times the first invariant from the second invariant.
52      n.divideWithRemainder(m, q);
53      r2 -= q*r1; s2 -= q*s1;
54    }
55  }
56
57  BigUnsigned modinv(const BigInteger &x, const BigUnsigned &n) {
58    BigInteger g, r, s;
59    extendedEuclidean(x, n, g, r, s);
60    if (g == 1)
61      // r*x + s*n == 1, so r*x === 1 (mod n), so r is the answer.
62      return (r %n).getMagnitude(); // (r %n) will be nonnegative
63    else
64      throw "BigInteger modinv: x and n have a common factor";
65  }
66
67  BigUnsigned modexp(const BigInteger &base, const BigUnsigned &exponent,
68      const BigUnsigned &modulus) {
69    BigUnsigned ans = 1, base2 = (base %modulus).getMagnitude();
70    BigUnsigned::Index i = exponent.bitLength();
71    // For each bit of the exponent, most to least significant...
72    while (i > 0) {
73      i--;
74      // Square.
75      ans *= ans;
76      ans %= modulus;
77      // And multiply if the bit is a 1.
78      if (exponent.getBit(i)) {
79        ans *= base2;
80        ans %= modulus;
81      }
82    }
83    return ans;
84  }
```

- BigIntegerUtils.cc:

```
1  #include "BigIntegerUtils.hh"
2  #include "BigUnsignedInABase.hh"
3
4  std::string bigUnsignedToString(const BigUnsigned &x) {
5    return std::string(BigUnsignedInABase(x, 10));
6  }
7
8  std::string bigIntegerToString(const BigInteger &x) {
9    return (x.getSign() == BigInteger::negative)
10     ? (std::string("-") + bigUnsignedToString(x.getMagnitude()))
11     : (bigUnsignedToString(x.getMagnitude()));
12 }
13
14 BigUnsigned stringToBigUnsigned(const std::string &s) {
```

```
15   return BigUnsigned(BigUnsignedInABase(s, 10));
16 }
17
18 BigInteger stringToBigInteger(const std::string &s) {
19   // Recognize a sign followed by a BigUnsigned.
20   return (s[0] == '-') ? BigInteger(stringToBigUnsigned(s.substr(1, s.length()
         - 1)), BigInteger::negative)
21     : (s[0] == '+') ? BigInteger(stringToBigUnsigned(s.substr(1, s.length() -
           1)))
22     : BigInteger(stringToBigUnsigned(s));
23 }
24
25 std::ostream &operator <<(std::ostream &os, const BigUnsigned &x) {
26   BigUnsignedInABase::Base base;
27   long osFlags = os.flags();
28   if (osFlags & os.dec)
29     base = 10;
30   else if (osFlags & os.hex) {
31     base = 16;
32     if (osFlags & os.showbase)
33       os << "0x";
34   } else if (osFlags & os.oct) {
35     base = 8;
36     if (osFlags & os.showbase)
37       os << '0';
38   } else
39     throw "std::ostream << BigUnsigned: Could not determine the desired base
           from output-stream flags";
40   std::string s = std::string(BigUnsignedInABase(x, base));
41   os << s;
42   return os;
43 }
44
45 std::ostream &operator <<(std::ostream &os, const BigInteger &x) {
46   if (x.getSign() == BigInteger::negative)
47     os << '-';
48   os << x.getMagnitude();
49   return os;
50 }
```

- BigUnsigned.cc:

```
1  #include "BigUnsigned.hh"
2
3  // Memory management definitions have moved to the bottom of NumberlikeArray.hh
       .
4
5  // The templates used by these constructors and converters are at the bottom of
6  // BigUnsigned.hh.
7
8  BigUnsigned::BigUnsigned(unsigned long  x) { initFromPrimitive      (x); }
9  BigUnsigned::BigUnsigned(unsigned int   x) { initFromPrimitive      (x); }
10 BigUnsigned::BigUnsigned(unsigned short x) { initFromPrimitive      (x); }
11 BigUnsigned::BigUnsigned(         long  x) { initFromSignedPrimitive(x); }
12 BigUnsigned::BigUnsigned(         int   x) { initFromSignedPrimitive(x); }
13 BigUnsigned::BigUnsigned(         short x) { initFromSignedPrimitive(x); }
14
15 unsigned long  BigUnsigned::toUnsignedLong () const { return convertToPrimitive
           <unsigned long >(); }
16 unsigned int   BigUnsigned::toUnsignedInt  () const { return convertToPrimitive
           <unsigned int  >(); }
17 unsigned short BigUnsigned::toUnsignedShort() const { return convertToPrimitive
           <unsigned short>(); }
```

```
18 long           BigUnsigned::toLong          () const { return
       convertToSignedPrimitive<          long >(); }
19 int            BigUnsigned::toInt           () const { return
       convertToSignedPrimitive<          int  >(); }
20 short          BigUnsigned::toShort         () const { return
       convertToSignedPrimitive<          short>(); }
21
22 // BIT/BLOCK ACCESSORS
23
24 void BigUnsigned::setBlock(Index i, Blk newBlock) {
25   if (newBlock == 0) {
26     if (i < len) {
27       blk[i] = 0;
28       zapLeadingZeros();
29     }
30     // If i >= len, no effect.
31   } else {
32     if (i >= len) {
33       // The nonzero block extends the number.
34       allocateAndCopy(i+1);
35       // Zero any added blocks that we aren't setting.
36       for (Index j = len; j < i; j++)
37         blk[j] = 0;
38       len = i+1;
39     }
40     blk[i] = newBlock;
41   }
42 }
43
44 /* Evidently the compiler wants BigUnsigned:: on the return type because, at
45  * that point, it hasn't yet parsed the BigUnsigned:: on the name to get the
46  * proper scope. */
47 BigUnsigned::Index BigUnsigned::bitLength() const {
48   if (isZero())
49     return 0;
50   else {
51     Blk leftmostBlock = getBlock(len - 1);
52     Index leftmostBlockLen = 0;
53     while (leftmostBlock != 0) {
54       leftmostBlock >>= 1;
55       leftmostBlockLen++;
56     }
57     return leftmostBlockLen + (len - 1) * N;
58   }
59 }
60
61 void BigUnsigned::setBit(Index bi, bool newBit) {
62   Index blockI = bi / N;
63   Blk block = getBlock(blockI), mask = Blk(1) << (bi %N);
64   block = newBit ? (block | mask) : (block & ~mask);
65   setBlock(blockI, block);
66 }
67
68 // COMPARISON
69 BigUnsigned::CmpRes BigUnsigned::compareTo(const BigUnsigned &x) const {
70   // A bigger length implies a bigger number.
71   if (len < x.len)
72     return less;
73   else if (len > x.len)
74     return greater;
75   else {
76     // Compare blocks one by one from left to right.
77     Index i = len;
78     while (i > 0) {
```

```
79        i--;
80        if (blk[i] == x.blk[i])
81          continue;
82        else if (blk[i] > x.blk[i])
83          return greater;
84        else
85          return less;
86      }
87      // If no blocks differed, the numbers are equal.
88      return equal;
89    }
90  }
91
92  // COPY-LESS OPERATIONS
93
94  /*
95   * On most calls to copy-less operations, it's safe to read the inputs little
         by
96   * little and write the outputs little by little.  However, if one of the
97   * inputs is coming from the same variable into which the output is to be
98   * stored (an "aliased" call), we risk overwriting the input before we read it.
99   * In this case, we first compute the result into a temporary BigUnsigned
100  * variable and then copy it into the requested output variable *this.
101  * Each put-here operation uses the DTRT_ALIASED macro (Do The Right Thing on
102  * aliased calls) to generate code for this check.
103  *
104  * I adopted this approach on 2007.02.13 (see Assignment Operators in
105  * BigUnsigned.hh).  Before then, put-here operations rejected aliased calls
106  * with an exception.  I think doing the right thing is better.
107  *
108  * Some of the put-here operations can probably handle aliased calls safely
109  * without the extra copy because (for example) they process blocks strictly
110  * right-to-left.  At some point I might determine which ones don't need the
111  * copy, but my reasoning would need to be verified very carefully.  For now
112  * I'll leave in the copy.
113  */
114 #define DTRT_ALIASED(cond, op) \
115   if (cond) { \
116     BigUnsigned tmpThis; \
117     tmpThis.op; \
118     *this = tmpThis; \
119     return; \
120   }
121
122
123
124 void BigUnsigned::add(const BigUnsigned &a, const BigUnsigned &b) {
125   DTRT_ALIASED(this == &a || this == &b, add(a, b));
126   // If one argument is zero, copy the other.
127   if (a.len == 0) {
128     operator =(b);
129     return;
130   } else if (b.len == 0) {
131     operator =(a);
132     return;
133   }
134   // Some variables...
135   // Carries in and out of an addition stage
136   bool carryIn, carryOut;
137   Blk temp;
138   Index i;
139   // a2 points to the longer input, b2 points to the shorter
140   const BigUnsigned *a2, *b2;
141   if (a.len >= b.len) {
```

```
142     a2 = &a;
143     b2 = &b;
144   } else {
145     a2 = &b;
146     b2 = &a;
147   }
148   // Set prelimiary length and make room in this BigUnsigned
149   len = a2->len + 1;
150   allocate(len);
151   // For each block index that is present in both inputs...
152   for (i = 0, carryIn = false; i < b2->len; i++) {
153     // Add input blocks
154     temp = a2->blk[i] + b2->blk[i];
155     // If a rollover occurred, the result is less than either input.
156     // This test is used many times in the BigUnsigned code.
157     carryOut = (temp < a2->blk[i]);
158     // If a carry was input, handle it
159     if (carryIn) {
160       temp++;
161       carryOut |= (temp == 0);
162     }
163     blk[i] = temp; // Save the addition result
164     carryIn = carryOut; // Pass the carry along
165   }
166   // If there is a carry left over, increase blocks until
167   // one does not roll over.
168   for (; i < a2->len && carryIn; i++) {
169     temp = a2->blk[i] + 1;
170     carryIn = (temp == 0);
171     blk[i] = temp;
172   }
173   // If the carry was resolved but the larger number
174   // still has blocks, copy them over.
175   for (; i < a2->len; i++)
176     blk[i] = a2->blk[i];
177   // Set the extra block if there's still a carry, decrease length otherwise
178   if (carryIn)
179     blk[i] = 1;
180   else
181     len--;
182 }
183
184 void BigUnsigned::subtract(const BigUnsigned &a, const BigUnsigned &b) {
185   DTRT_ALIASED(this == &a || this == &b, subtract(a, b));
186   if (b.len == 0) {
187     // If b is zero, copy a.
188     operator =(a);
189     return;
190   } else if (a.len < b.len)
191     // If a is shorter than b, the result is negative.
192     throw "BigUnsigned::subtract: "
193       "Negative result in unsigned calculation";
194   // Some variables...
195   bool borrowIn, borrowOut;
196   Blk temp;
197   Index i;
198   // Set preliminary length and make room
199   len = a.len;
200   allocate(len);
201   // For each block index that is present in both inputs...
202   for (i = 0, borrowIn = false; i < b.len; i++) {
203     temp = a.blk[i] - b.blk[i];
204     // If a reverse rollover occurred,
205     // the result is greater than the block from a.
```

```
206     borrowOut = (temp > a.blk[i]);
207     // Handle an incoming borrow
208     if (borrowIn) {
209       borrowOut |= (temp == 0);
210       temp--;
211     }
212     blk[i] = temp; // Save the subtraction result
213     borrowIn = borrowOut; // Pass the borrow along
214   }
215   // If there is a borrow left over, decrease blocks until
216   // one does not reverse rollover.
217   for (; i < a.len && borrowIn; i++) {
218     borrowIn = (a.blk[i] == 0);
219     blk[i] = a.blk[i] - 1;
220   }
221   /* If there's still a borrow, the result is negative.
222    * Throw an exception, but zero out this object so as to leave it in a
223    * predictable state. */
224   if (borrowIn) {
225     len = 0;
226     throw "BigUnsigned::subtract: Negative result in unsigned calculation";
227   } else
228     // Copy over the rest of the blocks
229     for (; i < a.len; i++)
230       blk[i] = a.blk[i];
231   // Zap leading zeros
232   zapLeadingZeros();
233 }
234
235 /*
236  * About the multiplication and division algorithms:
237  *
238  * I searched unsucessfully for fast C++ built-in operations like the `b_0'
239  * and `c_0' Knuth describes in Section 4.3.1 of ``The Art of Computer
240  * Programming'' (replace `place' by `Blk'):
241  *
242  *    ``b_0[:] multiplication of a one-place integer by another one-place
243  *      integer, giving a two-place answer;
244  *
245  *    ``c_0[:] division of a two-place integer by a one-place integer,
246  *      provided that the quotient is a one-place integer, and yielding
247  *      also a one-place remainder.''
248  *
249  * I also missed his note that ``[b]y adjusting the word size, if
250  * necessary, nearly all computers will have these three operations
251  * available'', so I gave up on trying to use algorithms similar to his.
252  * A future version of the library might include such algorithms; I
253  * would welcome contributions from others for this.
254  *
255  * I eventually decided to use bit-shifting algorithms.  To multiply `a'
256  * and `b', we zero out the result.  Then, for each `1' bit in `a', we
257  * shift `b' left the appropriate amount and add it to the result.
258  * Similarly, to divide `a' by `b', we shift `b' left varying amounts,
259  * repeatedly trying to subtract it from `a'.  When we succeed, we note
260  * the fact by setting a bit in the quotient.  While these algorithms
261  * have the same O(n^2) time complexity as Knuth's, the ``constant factor''
262  * is likely to be larger.
263  *
264  * Because I used these algorithms, which require single-block addition
265  * and subtraction rather than single-block multiplication and division,
266  * the innermost loops of all four routines are very similar.  Study one
267  * of them and all will become clear.
268  */
269
```

```
/*
 * This is a little inline function used by both the multiplication
 * routine and the division routine.
 *
 * 'getShiftedBlock' returns the 'x'th block of 'num << y'.
 * 'y' may be anything from 0 to N - 1, and 'x' may be anything from
 * 0 to 'num.len'.
 *
 * Two things contribute to this block:
 *
 * (1) The 'N - y' low bits of 'num.blk[x]', shifted 'y' bits left.
 *
 * (2) The 'y' high bits of 'num.blk[x-1]', shifted 'N - y' bits right.
 *
 * But we must be careful if 'x == 0' or 'x == num.len', in
 * which case we should use 0 instead of (2) or (1), respectively.
 *
 * If 'y == 0', then (2) contributes 0, as it should.  However,
 * in some computer environments, for a reason I cannot understand,
 * 'a >> b' means 'a >> (b %N)'.  This means 'num.blk[x-1] >> (N - y)'
 * will return 'num.blk[x-1]' instead of the desired 0 when 'y == 0';
 * the test 'y == 0' handles this case specially.
 */
inline BigUnsigned::Blk getShiftedBlock(const BigUnsigned &num,
  BigUnsigned::Index x, unsigned int y) {
  BigUnsigned::Blk part1 = (x == 0 || y == 0) ? 0 : (num.blk[x - 1] >> (
      BigUnsigned::N - y));
  BigUnsigned::Blk part2 = (x == num.len) ? 0 : (num.blk[x] << y);
  return part1 | part2;
}

void BigUnsigned::multiply(const BigUnsigned &a, const BigUnsigned &b) {
  DTRT_ALIASED(this == &a || this == &b, multiply(a, b));
  // If either a or b is zero, set to zero.
  if (a.len == 0 || b.len == 0) {
    len = 0;
    return;
  }
  /*
   * Overall method:
   *
   * Set this = 0.
   * For each 1-bit of 'a' (say the 'i2'th bit of block 'i'):
   *    Add 'b << (i blocks and i2 bits)' to *this.
   */
  // Variables for the calculation
  Index i, j, k;
  unsigned int i2;
  Blk temp;
  bool carryIn, carryOut;
  // Set preliminary length and make room
  len = a.len + b.len;
  allocate(len);
  // Zero out this object
  for (i = 0; i < len; i++)
    blk[i] = 0;
  // For each block of the first number...
  for (i = 0; i < a.len; i++) {
    // For each 1-bit of that block...
    for (i2 = 0; i2 < N; i2++) {
      if ((a.blk[i] & (Blk(1) << i2)) == 0)
        continue;
      /*
       * Add b to this, shifted left i blocks and i2 bits.
```

```
333         * j is the index in b, and k = i + j is the index in this.
334         *
335         * 'getShiftedBlock', a short inline function defined above,
336         * is now used for the bit handling.  It replaces the more
337         * complex 'bHigh' code, in which each run of the loop dealt
338         * immediately with the low bits and saved the high bits to
339         * be picked up next time.  The last run of the loop used to
340         * leave leftover high bits, which were handled separately.
341         * Instead, this loop runs an additional time with j == b.len.
342         * These changes were made on 2005.01.11.
343         */
344        for (j = 0, k = i, carryIn = false; j <= b.len; j++, k++) {
345          /*
346           * The body of this loop is very similar to the body of the first loop
347           * in 'add', except that this loop does a '+=' instead of a '+'.
348           */
349          temp = blk[k] + getShiftedBlock(b, j, i2);
350          carryOut = (temp < blk[k]);
351          if (carryIn) {
352            temp++;
353            carryOut |= (temp == 0);
354          }
355          blk[k] = temp;
356          carryIn = carryOut;
357        }
358        // No more extra iteration to deal with 'bHigh'.
359        // Roll-over a carry as necessary.
360        for (; carryIn; k++) {
361          blk[k]++;
362          carryIn = (blk[k] == 0);
363        }
364      }
365    }
366    // Zap possible leading zero
367    if (blk[len - 1] == 0)
368      len--;
369  }
370
371  /*
372   * DIVISION WITH REMAINDER
373   * This monstrous function mods *this by the given divisor b while storing the
374   * quotient in the given object q; at the end, *this contains the remainder.
375   * The seemingly bizarre pattern of inputs and outputs was chosen so that the
376   * function copies as little as possible (since it is implemented by repeated
377   * subtraction of multiples of b from *this).
378   *
379   * "modWithQuotient" might be a better name for this function, but I would
380   * rather not change the name now.
381   */
382  void BigUnsigned::divideWithRemainder(const BigUnsigned &b, BigUnsigned &q) {
383    /* Defending against aliased calls is more complex than usual because we
384     * are writing to both *this and q.
385     *
386     * It would be silly to try to write quotient and remainder to the
387     * same variable.  Rule that out right away. */
388    if (this == &q)
389      throw "BigUnsigned::divideWithRemainder: Cannot write quotient and
           remainder into the same variable";
390    /* Now *this and q are separate, so the only concern is that b might be
391     * aliased to one of them.  If so, use a temporary copy of b. */
392    if (this == &b || &q == &b) {
393      BigUnsigned tmpB(b);
394      divideWithRemainder(tmpB, q);
395      return;
```

```
396   }
397
398   /*
399    * Knuth's definition of mod (which this function uses) is somewhat
400    * different from the C++ definition of % in case of division by 0.
401    *
402    * We let a / 0 == 0 (it doesn't matter much) and a % 0 == a, no
403    * exceptions thrown.  This allows us to preserve both Knuth's demand
404    * that a mod 0 == a and the useful property that
405    * (a / b) * b + (a % b) == a.
406    */
407   if (b.len == 0) {
408     q.len = 0;
409     return;
410   }
411
412   /*
413    * If *this.len < b.len, then *this < b, and we can be sure that b doesn't go
         into
414    * *this at all.  The quotient is 0 and *this is already the remainder (so
         leave it alone).
415    */
416   if (len < b.len) {
417     q.len = 0;
418     return;
419   }
420
421   // At this point we know (*this).len >= b.len > 0.  (Whew!)
422
423   /*
424    * Overall method:
425    *
426    * For each appropriate i and i2, decreasing:
427    *    Subtract (b << (i blocks and i2 bits)) from *this, storing the
428    *      result in subtractBuf.
429    *    If the subtraction succeeds with a nonnegative result:
430    *        Turn on bit i2 of block i of the quotient q.
431    *        Copy subtractBuf back into *this.
432    *    Otherwise bit i2 of block i remains off, and *this is unchanged.
433    *
434    * Eventually q will contain the entire quotient, and *this will
435    * be left with the remainder.
436    *
437    * subtractBuf[x] corresponds to blk[x], not blk[x+i], since 2005.01.11.
438    * But on a single iteration, we don't touch the i lowest blocks of blk
439    * (and don't use those of subtractBuf) because these blocks are
440    * unaffected by the subtraction: we are subtracting
441    * (b << (i blocks and i2 bits)), which ends in at least 'i' zero
442    * blocks. */
443   // Variables for the calculation
444   Index i, j, k;
445   unsigned int i2;
446   Blk temp;
447   bool borrowIn, borrowOut;
448
449   /*
450    * Make sure we have an extra zero block just past the value.
451    *
452    * When we attempt a subtraction, we might shift 'b' so
453    * its first block begins a few bits left of the dividend,
454    * and then we'll try to compare these extra bits with
455    * a nonexistent block to the left of the dividend.  The
456    * extra zero block ensures sensible behavior; we need
457    * an extra block in 'subtractBuf' for exactly the same reason.
```

```
458     */
459   Index origLen = len; // Save real length.
460   /* To avoid an out-of-bounds access in case of reallocation, allocate
461    * first and then increment the logical length. */
462   allocateAndCopy(len + 1);
463   len++;
464   blk[origLen] = 0; // Zero the added block.
465
466   // subtractBuf holds part of the result of a subtraction; see above.
467   Blk *subtractBuf = new Blk[len];
468
469   // Set preliminary length for quotient and make room
470   q.len = origLen - b.len + 1;
471   q.allocate(q.len);
472   // Zero out the quotient
473   for (i = 0; i < q.len; i++)
474     q.blk[i] = 0;
475
476   // For each possible left-shift of b in blocks...
477   i = q.len;
478   while (i > 0) {
479     i--;
480     // For each possible left-shift of b in bits...
481     // (Remember, N is the number of bits in a Blk.)
482     q.blk[i] = 0;
483     i2 = N;
484     while (i2 > 0) {
485       i2--;
486       /*
487        * Subtract b, shifted left i blocks and i2 bits, from *this,
488        * and store the answer in subtractBuf.  In the for loop, 'k == i + j'.
489        *
490        * Compare this to the middle section of 'multiply'.  They
491        * are in many ways analogous.  See especially the discussion
492        * of 'getShiftedBlock'.
493        */
494       for (j = 0, k = i, borrowIn = false; j <= b.len; j++, k++) {
495         temp = blk[k] - getShiftedBlock(b, j, i2);
496         borrowOut = (temp > blk[k]);
497         if (borrowIn) {
498           borrowOut |= (temp == 0);
499           temp--;
500         }
501         // Since 2005.01.11, indices of 'subtractBuf' directly match those of '
               blk', so use 'k'.
502         subtractBuf[k] = temp;
503         borrowIn = borrowOut;
504       }
505       // No more extra iteration to deal with 'bHigh'.
506       // Roll-over a borrow as necessary.
507       for (; k < origLen && borrowIn; k++) {
508         borrowIn = (blk[k] == 0);
509         subtractBuf[k] = blk[k] - 1;
510       }
511       /*
512        * If the subtraction was performed successfully (!borrowIn),
513        * set bit i2 in block i of the quotient.
514        *
515        * Then, copy the portion of subtractBuf filled by the subtraction
516        * back to *this.  This portion starts with block i and ends—
517        * where?  Not necessarily at block 'i + b.len'!  Well, we
518        * increased k every time we saved a block into subtractBuf, so
519        * the region of subtractBuf we copy is just [i, k).
520        */
```

```
521       if (!borrowIn) {
522         q.blk[i] |= (Blk(1) << i2);
523         while (k > i) {
524           k--;
525           blk[k] = subtractBuf[k];
526         }
527       }
528     }
529   }
530   // Zap possible leading zero in quotient
531   if (q.blk[q.len - 1] == 0)
532     q.len--;
533   // Zap any/all leading zeros in remainder
534   zapLeadingZeros();
535   // Deallocate subtractBuf.
536   // (Thanks to Brad Spencer for noticing my accidental omission of this!)
537   delete [] subtractBuf;
538 }
539
540 /* BITWISE OPERATORS
541  * These are straightforward blockwise operations except that they differ in
542  * the output length and the necessity of zapLeadingZeros. */
543
544 void BigUnsigned::bitAnd(const BigUnsigned &a, const BigUnsigned &b) {
545   DTRT_ALIASED(this == &a || this == &b, bitAnd(a, b));
546   // The bitwise & can't be longer than either operand.
547   len = (a.len >= b.len) ? b.len : a.len;
548   allocate(len);
549   Index i;
550   for (i = 0; i < len; i++)
551     blk[i] = a.blk[i] & b.blk[i];
552   zapLeadingZeros();
553 }
554
555 void BigUnsigned::bitOr(const BigUnsigned &a, const BigUnsigned &b) {
556   DTRT_ALIASED(this == &a || this == &b, bitOr(a, b));
557   Index i;
558   const BigUnsigned *a2, *b2;
559   if (a.len >= b.len) {
560     a2 = &a;
561     b2 = &b;
562   } else {
563     a2 = &b;
564     b2 = &a;
565   }
566   allocate(a2->len);
567   for (i = 0; i < b2->len; i++)
568     blk[i] = a2->blk[i] | b2->blk[i];
569   for (; i < a2->len; i++)
570     blk[i] = a2->blk[i];
571   len = a2->len;
572   // Doesn't need zapLeadingZeros.
573 }
574
575 void BigUnsigned::bitXor(const BigUnsigned &a, const BigUnsigned &b) {
576   DTRT_ALIASED(this == &a || this == &b, bitXor(a, b));
577   Index i;
578   const BigUnsigned *a2, *b2;
579   if (a.len >= b.len) {
580     a2 = &a;
581     b2 = &b;
582   } else {
583     a2 = &b;
584     b2 = &a;
```

```
585     }
586     allocate(a2->len);
587     for (i = 0; i < b2->len; i++)
588       blk[i] = a2->blk[i] ^ b2->blk[i];
589     for (; i < a2->len; i++)
590       blk[i] = a2->blk[i];
591     len = a2->len;
592     zapLeadingZeros();
593   }
594
595   void BigUnsigned::bitShiftLeft(const BigUnsigned &a, int b) {
596     DTRT_ALIASED(this == &a, bitShiftLeft(a, b));
597     if (b < 0) {
598       if (b << 1 == 0)
599         throw "BigUnsigned::bitShiftLeft: "
600           "Pathological shift amount not implemented";
601       else {
602         bitShiftRight(a, -b);
603         return;
604       }
605     }
606     Index shiftBlocks = b / N;
607     unsigned int shiftBits = b %N;
608     // + 1: room for high bits nudged left into another block
609     len = a.len + shiftBlocks + 1;
610     allocate(len);
611     Index i, j;
612     for (i = 0; i < shiftBlocks; i++)
613       blk[i] = 0;
614     for (j = 0, i = shiftBlocks; j <= a.len; j++, i++)
615       blk[i] = getShiftedBlock(a, j, shiftBits);
616     // Zap possible leading zero
617     if (blk[len - 1] == 0)
618       len--;
619   }
620
621   void BigUnsigned::bitShiftRight(const BigUnsigned &a, int b) {
622     DTRT_ALIASED(this == &a, bitShiftRight(a, b));
623     if (b < 0) {
624       if (b << 1 == 0)
625         throw "BigUnsigned::bitShiftRight: "
626           "Pathological shift amount not implemented";
627       else {
628         bitShiftLeft(a, -b);
629         return;
630       }
631     }
632     // This calculation is wacky, but expressing the shift as a left bit shift
633     // within each block lets us use getShiftedBlock.
634     Index rightShiftBlocks = (b + N - 1) / N;
635     unsigned int leftShiftBits = N * rightShiftBlocks - b;
636     // Now (N * rightShiftBlocks - leftShiftBits) == b
637     // and 0 <= leftShiftBits < N.
638     if (rightShiftBlocks >= a.len + 1) {
639       // All of a is guaranteed to be shifted off, even considering the left
640       // bit shift.
641       len = 0;
642       return;
643     }
644     // Now we're allocating a positive amount.
645     // + 1: room for high bits nudged left into another block
646     len = a.len + 1 - rightShiftBlocks;
647     allocate(len);
648     Index i, j;
```

```
649   for (j = rightShiftBlocks, i = 0; j <= a.len; j++, i++)
650     blk[i] = getShiftedBlock(a, j, leftShiftBits);
651   // Zap possible leading zero
652   if (blk[len - 1] == 0)
653     len--;
654 }
655
656 // INCREMENT/DECREMENT OPERATORS
657
658 // Prefix increment
659 void BigUnsigned::operator ++() {
660   Index i;
661   bool carry = true;
662   for (i = 0; i < len && carry; i++) {
663     blk[i]++;
664     carry = (blk[i] == 0);
665   }
666   if (carry) {
667     // Allocate and then increase length, as in divideWithRemainder
668     allocateAndCopy(len + 1);
669     len++;
670     blk[i] = 1;
671   }
672 }
673
674 // Postfix increment: same as prefix
675 void BigUnsigned::operator ++(int) {
676   operator ++();
677 }
678
679 // Prefix decrement
680 void BigUnsigned::operator --() {
681   if (len == 0)
682     throw "BigUnsigned::operator --(): Cannot decrement an unsigned zero";
683   Index i;
684   bool borrow = true;
685   for (i = 0; borrow; i++) {
686     borrow = (blk[i] == 0);
687     blk[i]--;
688   }
689   // Zap possible leading zero (there can only be one)
690   if (blk[len - 1] == 0)
691     len--;
692 }
693
694 // Postfix decrement: same as prefix
695 void BigUnsigned::operator --(int) {
696   operator --();
697 }
```

Ahora es el turno de los archivos .h:

- BigInteger.h:

```
1  #ifndef BIGINTEGER_H
2  #define BIGINTEGER_H
3
4  #include "BigUnsigned.hh"
5
6  /* A BigInteger object represents a signed integer of size limited only by
7   * available memory.  BigUnsigneds support most mathematical operators and can
8   * be converted to and from most primitive integer types.
9   *
10  * A BigInteger is just an aggregate of a BigUnsigned and a sign.  (It is no
11  * longer derived from BigUnsigned because that led to harmful implicit
```

```
 * conversions.) */
class BigInteger {

public:
  typedef BigUnsigned::Blk Blk;
  typedef BigUnsigned::Index Index;
  typedef BigUnsigned::CmpRes CmpRes;
  static const CmpRes
    less    = BigUnsigned::less    ,
    equal   = BigUnsigned::equal   ,
    greater = BigUnsigned::greater;
  // Enumeration for the sign of a BigInteger.
  enum Sign { negative = -1, zero = 0, positive = 1 };

protected:
  Sign sign;
  BigUnsigned mag;

public:
  // Constructs zero.
  BigInteger() : sign(zero), mag() {}

  // Copy constructor
  BigInteger(const BigInteger &x) : sign(x.sign), mag(x.mag) {};

  // Assignment operator
  void operator=(const BigInteger &x);

  // Constructor that copies from a given array of blocks with a sign.
  BigInteger(const Blk *b, Index blen, Sign s);

  // Nonnegative constructor that copies from a given array of blocks.
  BigInteger(const Blk *b, Index blen) : mag(b, blen) {
    sign = mag.isZero() ? zero : positive;
  }

  // Constructor from a BigUnsigned and a sign
  BigInteger(const BigUnsigned &x, Sign s);

  // Nonnegative constructor from a BigUnsigned
  BigInteger(const BigUnsigned &x) : mag(x) {
    sign = mag.isZero() ? zero : positive;
  }

  // Constructors from primitive integer types
  BigInteger(unsigned long  x);
  BigInteger(         long  x);
  BigInteger(unsigned int   x);
  BigInteger(         int   x);
  BigInteger(unsigned short x);
  BigInteger(         short x);

  /* Converters to primitive integer types
   * The implicit conversion operators caused trouble, so these are now
   * named. */
  unsigned long  toUnsignedLong () const;
  long           toLong         () const;
  unsigned int   toUnsignedInt  () const;
  int            toInt          () const;
  unsigned short toUnsignedShort() const;
  short          toShort        () const;
protected:
  // Helper
  template <class X> X convertToUnsignedPrimitive() const;
```

```
76    template <class X, class UX> X convertToSignedPrimitive() const;
77  public:
78
79    // ACCESSORS
80    Sign getSign() const { return sign; }
81    /* The client can't do any harm by holding a read-only reference to the
82     * magnitude. */
83    const BigUnsigned &getMagnitude() const { return mag; }
84
85    // Some accessors that go through to the magnitude
86    Index getLength() const { return mag.getLength(); }
87    Index getCapacity() const { return mag.getCapacity(); }
88    Blk getBlock(Index i) const { return mag.getBlock(i); }
89    bool isZero() const { return sign == zero; } // A bit special
90
91    // COMPARISONS
92
93    // Compares this to x like Perl's <=>
94    CmpRes compareTo(const BigInteger &x) const;
95
96    // Ordinary comparison operators
97    bool operator ==(const BigInteger &x) const {
98      return sign == x.sign && mag == x.mag;
99    }
100   bool operator !=(const BigInteger &x) const { return !operator ==(x); };
101   bool operator < (const BigInteger &x) const { return compareTo(x) == less    ;
        }
102   bool operator <=(const BigInteger &x) const { return compareTo(x) != greater;
        }
103   bool operator >=(const BigInteger &x) const { return compareTo(x) != less    ;
        }
104   bool operator > (const BigInteger &x) const { return compareTo(x) == greater;
        }
105
106   // OPERATORS -- See the discussion in BigUnsigned.hh.
107   void add     (const BigInteger &a, const BigInteger &b);
108   void subtract(const BigInteger &a, const BigInteger &b);
109   void multiply(const BigInteger &a, const BigInteger &b);
110   /* See the comment on BigUnsigned::divideWithRemainder.  Semantics
111    * differ from those of primitive integers when negatives and/or zeros
112    * are involved. */
113   void divideWithRemainder(const BigInteger &b, BigInteger &q);
114   void negate(const BigInteger &a);
115
116   /* Bitwise operators are not provided for BigIntegers.  Use
117    * getMagnitude to get the magnitude and operate on that instead. */
118
119   BigInteger operator +(const BigInteger &x) const;
120   BigInteger operator -(const BigInteger &x) const;
121   BigInteger operator *(const BigInteger &x) const;
122   BigInteger operator /(const BigInteger &x) const;
123   BigInteger operator %(const BigInteger &x) const;
124   BigInteger operator -() const;
125
126   void operator +=(const BigInteger &x);
127   void operator -=(const BigInteger &x);
128   void operator *=(const BigInteger &x);
129   void operator /=(const BigInteger &x);
130   void operator %=(const BigInteger &x);
131   void flipSign();
132
133   // INCREMENT/DECREMENT OPERATORS
134   void operator ++(   );
135   void operator ++(int);
```

```
136   void operator --(   );
137   void operator --(int);
138 };
139
140 // NORMAL OPERATORS
141 /* These create an object to hold the result and invoke
142  * the appropriate put-here operation on it, passing
143  * this and x.  The new object is then returned. */
144 inline BigInteger BigInteger::operator +(const BigInteger &x) const {
145   BigInteger ans;
146   ans.add(*this, x);
147   return ans;
148 }
149 inline BigInteger BigInteger::operator -(const BigInteger &x) const {
150   BigInteger ans;
151   ans.subtract(*this, x);
152   return ans;
153 }
154 inline BigInteger BigInteger::operator *(const BigInteger &x) const {
155   BigInteger ans;
156   ans.multiply(*this, x);
157   return ans;
158 }
159 inline BigInteger BigInteger::operator /(const BigInteger &x) const {
160   if (x.isZero()) throw "BigInteger::operator /: division by zero";
161   BigInteger q, r;
162   r = *this;
163   r.divideWithRemainder(x, q);
164   return q;
165 }
166 inline BigInteger BigInteger::operator %(const BigInteger &x) const {
167   if (x.isZero()) throw "BigInteger::operator %: division by zero";
168   BigInteger q, r;
169   r = *this;
170   r.divideWithRemainder(x, q);
171   return r;}
172 inline BigInteger BigInteger::operator -() const {
173   BigInteger ans;
174   ans.negate(*this);
175   return ans;
176 }
177
178 /*
179  * ASSIGNMENT OPERATORS
180  *
181  * Now the responsibility for making a temporary copy if necessary
182  * belongs to the put-here operations.  See Assignment Operators in
183  * BigUnsigned.hh.
184  */
185 inline void BigInteger::operator +=(const BigInteger &x) {
186   add(*this, x);
187 }
188 inline void BigInteger::operator -=(const BigInteger &x) {
189   subtract(*this, x);
190 }
191 inline void BigInteger::operator *=(const BigInteger &x) {
192   multiply(*this, x);
193 }
194 inline void BigInteger::operator /=(const BigInteger &x) {
195   if (x.isZero()) throw "BigInteger::operator /=: division by zero";
196   /* The following technique is slightly faster than copying *this first
197    * when x is large. */
198   BigInteger q;
199   divideWithRemainder(x, q);
```

```
  // *this contains the remainder, but we overwrite it with the quotient.
  *this = q;
}
inline void BigInteger::operator %=(const BigInteger &x) {
  if (x.isZero()) throw "BigInteger::operator %=: division by zero";
  BigInteger q;
  // Mods *this by x.  Don't care about quotient left in q.
  divideWithRemainder(x, q);
}
// This one is trivial
inline void BigInteger::flipSign() {
  sign = Sign(-sign);
}

#endif
```

- BigRational.h:

```
/*****************************************************************************
 *  BigRational.h
 *****************************************************************************/
//C++ Standar
#include <string>
#include <sstream>
#include <iostream>
//BigInteger Library. See https://mattmccutchen.net/bigint/ for documentation
#include "BigIntegerLibrary.hh"

using namespace std;

class BigRational
{

private:
  BigInteger num;
  BigInteger den;
  void init(BigInteger, BigInteger);

public:
  BigRational(int, int);
  BigRational(string);
  BigRational(int);
  BigRational(BigInteger, BigInteger);
  string toString();
  int BigRational::compareTo(const BigRational&);
  bool isZero();
  bool isPositive();
  bool isNegative();
  bool BigRational::equals(void*);
  BigRational BigRational::times(BigRational);
  BigRational BigRational::plus(BigRational);
  BigRational BigRational::negate();
  BigRational BigRational::minus(BigRational);
  BigRational BigRational::reciprocal();
  BigRational BigRational::divides(BigRational);
  double BigRational::doubleValue();
};
```

A continuación se muestra la funcion ttmath.

- ttmath.h:

```
/*
 * This file is a part of TTMath Bignum Library
```

```
 * and is distributed under the (new) BSD licence.
 * Author: Tomasz Sowa <t.sowa@ttmath.org>
 */

/*
 * Copyright (c) 2006-2012, Tomasz Sowa
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 *  * Redistributions of source code must retain the above copyright notice,
 *    this list of conditions and the following disclaimer.
 *
 *  * Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 *  * Neither the name Tomasz Sowa nor the names of contributors to this
 *    project may be used to endorse or promote products derived
 *    from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
 * THE POSSIBILITY OF SUCH DAMAGE.
 */



#ifndef headerfilettmathmathtt
#define headerfilettmathmathtt

/*!
  \file ttmath.h
    \brief Mathematics functions.
*/

#ifdef _MSC_VER
//warning C4127: conditional expression is constant
#pragma warning( disable: 4127 )
//warning C4702: unreachable code
#pragma warning( disable: 4702 )
//warning C4800: forcing value to bool 'true' or 'false' (performance warning)
#pragma warning( disable: 4800 )
#endif


#include "ttmathbig.h"
#include "ttmathobjects.h"


namespace ttmath
{
  /*
   *
   *  functions defined here are used only with Big<> types
```

```
67    *
68    *
69    */
70
71
72    /*
73     *
74     *  functions for rounding
75     *
76     *
77     */
78
79
80    /*!
81      this function skips the fraction from x
82      e.g  2.2  = 2
83           2.7  = 2
84          -2.2 = 2
85          -2.7 = 2
86    */
87    template<class ValueType>
88    ValueType SkipFraction(const ValueType & x)
89    {
90      ValueType result( x );
91      result.SkipFraction();
92
93    return result;
94    }
95
96
97    /*!
98      this function rounds to the nearest integer value
99      e.g  2.2  = 2
100          2.7  = 3
101         -2.2 = -2
102         -2.7 = -3
103   */
104   template<class ValueType>
105   ValueType Round(const ValueType & x, ErrorCode * err = 0)
106   {
107     if( x.IsNan() )
108     {
109       if( err )
110         *err = err_improper_argument;
111
112     return x; // NaN
113     }
114
115     ValueType result( x );
116     uint c = result.Round();
117
118     if( err )
119       *err = c ? err_overflow : err_ok;
120
121   return result;
122   }
123
124
125
126   /*!
127     this function returns a value representing the smallest integer
128     that is greater than or equal to x
129
130     Ceil(-3.7) = -3
```

```
131     Ceil(-3.1) = -3
132     Ceil(-3.0) = -3
133     Ceil(4.0)  = 4
134     Ceil(4.2)  = 5
135     Ceil(4.8)  = 5
136   */
137   template<class ValueType>
138   ValueType Ceil(const ValueType & x, ErrorCode * err = 0)
139   {
140     if( x.IsNan() )
141     {
142       if( err )
143         *err = err_improper_argument;
144
145     return x; // NaN
146     }
147
148     ValueType result(x);
149     uint c = 0;
150
151     result.SkipFraction();
152
153     if( result != x )
154     {
155       // x is with fraction
156       // if x is negative we don't have to do anything
157       if( !x.IsSign() )
158       {
159         ValueType one;
160         one.SetOne();
161
162         c += result.Add(one);
163       }
164     }
165
166     if( err )
167       *err = c ? err_overflow : err_ok;
168
169   return result;
170   }
171
172
173   /*!
174     this function returns a value representing the largest integer
175     that is less than or equal to x
176
177     Floor(-3.6) = -4
178     Floor(-3.1) = -4
179     Floor(-3)   = -3
180     Floor(2)    = 2
181     Floor(2.3)  = 2
182     Floor(2.8)  = 2
183   */
184   template<class ValueType>
185   ValueType Floor(const ValueType & x, ErrorCode * err = 0)
186   {
187     if( x.IsNan() )
188     {
189       if( err )
190         *err = err_improper_argument;
191
192     return x; // NaN
193     }
194
```

```
195     ValueType result(x);
196     uint c = 0;
197
198     result.SkipFraction();
199
200     if( result != x )
201     {
202       // x is with fraction
203       // if x is positive we don't have to do anything
204       if( x.IsSign() )
205       {
206         ValueType one;
207         one.SetOne();
208
209         c += result.Sub(one);
210       }
211     }
212
213     if( err )
214       *err = c ? err_overflow : err_ok;
215
216   return result;
217   }
218
219
220
221   /*
222    *
223    *  logarithms and the exponent
224    *
225    *
226    */
227
228
229   /*!
230     this function calculates the natural logarithm (logarithm with the base 'e
          ')
231   */
232   template<class ValueType>
233   ValueType Ln(const ValueType & x, ErrorCode * err = 0)
234   {
235     if( x.IsNan() )
236     {
237       if( err )
238         *err = err_improper_argument;
239
240     return x; // NaN
241     }
242
243     ValueType result;
244     uint state = result.Ln(x);
245
246     if( err )
247     {
248       switch( state )
249       {
250       case 0:
251         *err = err_ok;
252         break;
253       case 1:
254         *err = err_overflow;
255         break;
256       case 2:
257         *err = err_improper_argument;
```

```
258         break;
259       default:
260         *err = err_internal_error;
261         break;
262       }
263     }
264
265
266   return result;
267   }
268
269
270   /*!
271     this function calculates the logarithm
272   */
273   template<class ValueType>
274   ValueType Log(const ValueType & x, const ValueType & base, ErrorCode * err =
        0)
275   {
276     if( x.IsNan() )
277     {
278       if( err ) *err = err_improper_argument;
279       return x;
280     }
281
282     if( base.IsNan() )
283     {
284       if( err ) *err = err_improper_argument;
285       return base;
286     }
287
288     ValueType result;
289     uint state = result.Log(x, base);
290
291     if( err )
292     {
293       switch( state )
294       {
295       case 0:
296         *err = err_ok;
297         break;
298       case 1:
299         *err = err_overflow;
300         break;
301       case 2:
302       case 3:
303         *err = err_improper_argument;
304         break;
305       default:
306         *err = err_internal_error;
307         break;
308       }
309     }
310
311   return result;
312   }
313
314
315   /*!
316     this function calculates the expression e^x
317   */
318   template<class ValueType>
319   ValueType Exp(const ValueType & x, ErrorCode * err = 0)
320   {
```

```
321     if( x.IsNan() )
322     {
323       if( err )
324         *err = err_improper_argument;
325 
326     return x; // NaN
327     }
328 
329     ValueType result;
330     uint c = result.Exp(x);
331 
332     if( err )
333       *err = c ? err_overflow : err_ok;
334 
335   return result;
336   }
337 
338 
339   /*!
340   *
341   * trigonometric functions
342   *
343   */
344 
345 
346   /*
347     this namespace consists of auxiliary functions
348     (something like 'private' in a class)
349   */
350   namespace auxiliaryfunctions
351   {
352 
353   /*!
354     an auxiliary function for calculating the Sine
355     (you don't have to call this function)
356   */
357   template<class ValueType>
358   uint PrepareSin(ValueType & x, bool & change_sign)
359   {
360   ValueType temp;
361 
362     change_sign = false;
363 
364     if( x.IsSign() )
365     {
366       // we're using the formula 'sin(-x) = -sin(x)'
367       change_sign = !change_sign;
368       x.ChangeSign();
369     }
370 
371     // we're reducing the period 2*PI
372     // (for big values there'll always be zero)
373     temp.Set2Pi();
374 
375     if( x.Mod(temp) )
376       return 1;
377 
378 
379     // we're setting 'x' as being in the range of <0, 0.5PI>
380 
381     temp.SetPi();
382 
383     if( x > temp )
384     {
```

```
385       // x is in (pi, 2*pi>
386       x.Sub( temp );
387       change_sign = !change_sign;
388     }
389
390     temp.Set05Pi();
391
392     if( x > temp )
393     {
394       // x is in (0.5pi, pi>
395       x.Sub( temp );
396       x = temp - x;
397     }
398
399   return 0;
400   }
401
402
403   /*!
404     an auxiliary function for calculating the Sine
405     (you don't have to call this function)
406
407     it returns Sin(x) where 'x' is from <0, PI/2>
408     we're calculating the Sin with using Taylor series in zero or PI/2
409     (depending on which point of these two points is nearer to the 'x')
410
411     Taylor series:
412     sin(x) = sin(a) + cos(a)*(x-a)/(1!)
413              - sin(a)*((x-a)^2)/(2!) - cos(a)*((x-a)^3)/(3!)
414          + sin(a)*((x-a)^4)/(4!) + ...
415
416     when a=0 it'll be:
417     sin(x) = (x)/(1!) - (x^3)/(3!) + (x^5)/(5!) - (x^7)/(7!) + (x^9)/(9!) ...
418
419     and when a=PI/2:
420     sin(x) = 1 - ((x-PI/2)^2)/(2!) + ((x-PI/2)^4)/(4!) - ((x-PI/2)^6)/(6!) ...
421   */
422   template<class ValueType>
423   ValueType Sin0pi05(const ValueType & x)
424   {
425   ValueType result;
426   ValueType numerator, denominator;
427   ValueType d_numerator, d_denominator;
428   ValueType one, temp, old_result;
429
430     // temp = pi/4
431     temp.Set05Pi();
432     temp.exponent.SubOne();
433
434     one.SetOne();
435
436     if( x < temp )
437     {
438       // we're using the Taylor series with a=0
439       result    = x;
440       numerator = x;
441       denominator = one;
442
443       // d_numerator = x^2
444       d_numerator = x;
445       d_numerator.Mul(x);
446
447       d_denominator = 2;
448     }
```

```
449     else
450     {
451       // we're using the Taylor series with a=PI/2
452       result = one;
453       numerator = one;
454       denominator = one;
455
456       // d_numerator = (x-pi/2)^2
457       ValueType pi05;
458       pi05.Set05Pi();
459
460       temp = x;
461       temp.Sub( pi05 );
462       d_numerator = temp;
463       d_numerator.Mul( temp );
464
465       d_denominator = one;
466     }
467
468     uint c = 0;
469     bool addition = false;
470
471     old_result = result;
472     for(uint i=1 ; i<=TTMATH_ARITHMETIC_MAX_LOOP ; ++i)
473     {
474       // we're starting from a second part of the formula
475       c += numerator.     Mul( d_numerator );
476       c += denominator.   Mul( d_denominator );
477       c += d_denominator.Add( one );
478       c += denominator.   Mul( d_denominator );
479       c += d_denominator.Add( one );
480       temp = numerator;
481       c += temp.Div(denominator);
482
483       if( c )
484         // Sin is from <-1,1> and cannot make an overflow
485         // but the carry can be from the Taylor series
486         // (then we only break our calculations)
487         break;
488
489       if( addition )
490         result.Add( temp );
491       else
492         result.Sub( temp );
493
494
495       addition = !addition;
496
497       // we're testing whether the result has changed after adding
498       // the next part of the Taylor formula, if not we end the loop
499       // (it means 'x' is zero or 'x' is PI/2 or this part of the formula
500       // is too small)
501       if( result == old_result )
502         break;
503
504       old_result = result;
505     }
506
507   return result;
508   }
509
510   } // namespace auxiliaryfunctions
511
512
```

```
513
514   /*!
515     this function calculates the Sine
516   */
517   template<class ValueType>
518   ValueType Sin(ValueType x, ErrorCode * err = 0)
519   {
520   using namespace auxiliaryfunctions;
521
522   ValueType one, result;
523   bool change_sign;
524
525     if( x.IsNan() )
526     {
527       if( err )
528         *err = err_improper_argument;
529
530     return x;
531     }
532
533     if( err )
534       *err = err_ok;
535
536     if( PrepareSin( x, change_sign ) )
537     {
538       // x is too big, we cannnot reduce the 2*PI period
539       // prior to version 0.8.5 the result was zero
540
541       // result has NaN flag set by default
542
543       if( err )
544         *err = err_overflow; // maybe another error code? err_improper_argument
              ?
545
546     return result; // NaN is set by default
547     }
548
549     result = Sin0pi05( x );
550
551     one.SetOne();
552
553     // after calculations there can be small distortions in the result
554     if( result > one )
555       result = one;
556     else
557     if( result.IsSign() )
558       // we've calculated the sin from <0, pi/2> and the result
559       // should be positive
560       result.SetZero();
561
562     if( change_sign )
563       result.ChangeSign();
564
565   return result;
566   }
567
568
569   /*!
570     this function calulates the Cosine
571     we're using the formula cos(x) = sin(x + PI/2)
572   */
573   template<class ValueType>
574   ValueType Cos(ValueType x, ErrorCode * err = 0)
575   {
```

```
576     if( x.IsNan() )
577     {
578       if( err )
579         *err = err_improper_argument;
580 
581     return x; // NaN
582     }
583 
584     ValueType pi05;
585     pi05.Set05Pi();
586 
587     uint c = x.Add( pi05 );
588 
589     if( c )
590     {
591       if( err )
592         *err = err_overflow;
593 
594     return ValueType(); // result is undefined (NaN is set by default)
595     }
596 
597   return Sin(x, err);
598   }
599 
600 
601   /*!
602     this function calulates the Tangent
603     we're using the formula tan(x) = sin(x) / cos(x)
604 
605     it takes more time than calculating the Tan directly
606     from for example Taylor series but should be a bit preciser
607     because Tan receives its values from -infinity to +infinity
608     and when we calculate it from any series then we can make
609     a greater mistake than calculating 'sin/cos'
610   */
611   template<class ValueType>
612   ValueType Tan(const ValueType & x, ErrorCode * err = 0)
613   {
614     ValueType result = Cos(x, err);
615 
616     if( err && *err != err_ok )
617       return result;
618 
619     if( result.IsZero() )
620     {
621       if( err )
622         *err = err_improper_argument;
623 
624       result.SetNan();
625 
626     return result;
627     }
628 
629   return Sin(x, err) / result;
630   }
631 
632 
633   /*!
634     this function calulates the Tangent
635     look at the description of Tan(...)
636 
637     (the abbreviation of Tangent can be 'tg' as well)
638   */
639   template<class ValueType>
```

```
640 ValueType Tg(const ValueType & x, ErrorCode * err = 0)
641 {
642   return Tan(x, err);
643 }
644
645
646 /*!
647   this function calulates the Cotangent
648   we're using the formula tan(x) = cos(x) / sin(x)
649
650   (why do we make it in this way?
651   look at information in Tan() function)
652 */
653 template<class ValueType>
654 ValueType Cot(const ValueType & x, ErrorCode * err = 0)
655 {
656   ValueType result = Sin(x, err);
657
658   if( err && *err != err_ok )
659     return result;
660
661   if( result.IsZero() )
662   {
663     if( err )
664       *err = err_improper_argument;
665
666     result.SetNan();
667
668   return result;
669   }
670
671 return Cos(x, err) / result;
672 }
673
674
675 /*!
676   this function calulates the Cotangent
677   look at the description of Cot(...)
678
679   (the abbreviation of Cotangent can be 'ctg' as well)
680 */
681 template<class ValueType>
682 ValueType Ctg(const ValueType & x, ErrorCode * err = 0)
683 {
684   return Cot(x, err);
685 }
686
687
688 /*
689  *
690  *  inverse trigonometric functions
691  *
692  *
693  */
694
695 namespace auxiliaryfunctions
696 {
697
698 /*!
699   an auxiliary function for calculating the Arc Sine
700
701   we're calculating asin from the following formula:
702   asin(x) = x + (1*x^3)/(2*3) + (1*3*x^5)/(2*4*5) + (1*3*5*x^7)/(2*4*6*7) +
      ...
```

```
703     where abs(x) <= 1
704
705     we're using this formula when x is from <0, 1/2>
706   */
707   template<class ValueType>
708   ValueType ASin_0(const ValueType & x)
709   {
710   ValueType nominator, denominator, nominator_add, nominator_x, denominator_add
          , denominator_x;
711   ValueType two, result(x), x2(x);
712   ValueType nominator_temp, denominator_temp, old_result = result;
713   uint c = 0;
714
715   x2.Mul(x);
716   two = 2;
717
718   nominator.SetOne();
719   denominator      = two;
720   nominator_add    = nominator;
721   denominator_add = denominator;
722   nominator_x      = x;
723   denominator_x    = 3;
724
725     for(uint i=1 ; i<=TTMATH_ARITHMETIC_MAX_LOOP ; ++i)
726     {
727       c += nominator_x.Mul(x2);
728       nominator_temp = nominator_x;
729       c += nominator_temp.Mul(nominator);
730       denominator_temp = denominator;
731       c += denominator_temp.Mul(denominator_x);
732       c += nominator_temp.Div(denominator_temp);
733
734       // if there is a carry somewhere we only break the calculating
735       // the result should be ok -- it's from <-pi/2, pi/2>
736       if( c )
737         break;
738
739       result.Add(nominator_temp);
740
741       if( result == old_result )
742          // there's no sense to calculate more
743         break;
744
745       old_result = result;
746
747
748       c += nominator_add.Add(two);
749       c += denominator_add.Add(two);
750       c += nominator.Mul(nominator_add);
751       c += denominator.Mul(denominator_add);
752       c += denominator_x.Add(two);
753     }
754
755   return result;
756   }
757
758
759
760   /*!
761     an auxiliary function for calculating the Arc Sine
762
763     we're calculating asin from the following formula:
764     asin(x) = pi/2 - sqrt(2)*sqrt(1-x) * asin_temp
```

```
765     asin_temp = 1 + (1*(1-x))/((2*3)*(2)) + (1*3*(1-x)^2)/((2*4*5)*(4)) +
            (1*3*5*(1-x)^3)/((2*4*6*7)*(8)) + ...
766
767     where abs(x) <= 1
768
769     we're using this formula when x is from (1/2, 1>
770   */
771   template<class ValueType>
772   ValueType ASin_1(const ValueType & x)
773   {
774   ValueType nominator, denominator, nominator_add, nominator_x, nominator_x_add
        , denominator_add, denominator_x;
775   ValueType denominator2;
776   ValueType one, two, result;
777   ValueType nominator_temp, denominator_temp, old_result;
778   uint c = 0;
779
780   two = 2;
781
782   one.SetOne();
783   nominator    = one;
784   result       = one;
785   old_result   = result;
786   denominator      = two;
787   nominator_add    = nominator;
788   denominator_add = denominator;
789   nominator_x      = one;
790   nominator_x.Sub(x);
791   nominator_x_add = nominator_x;
792   denominator_x    = 3;
793   denominator2   = two;
794
795
796     for(uint i=1 ; i<=TTMATH_ARITHMETIC_MAX_LOOP ; ++i)
797     {
798       nominator_temp = nominator_x;
799       c += nominator_temp.Mul(nominator);
800       denominator_temp = denominator;
801       c += denominator_temp.Mul(denominator_x);
802       c += denominator_temp.Mul(denominator2);
803       c += nominator_temp.Div(denominator_temp);
804
805       // if there is a carry somewhere we only break the calculating
806       // the result should be ok -- it's from <-pi/2, pi/2>
807       if( c )
808         break;
809
810       result.Add(nominator_temp);
811
812       if( result == old_result )
813          // there's no sense to calculate more
814         break;
815
816       old_result = result;
817
818       c += nominator_x.Mul(nominator_x_add);
819       c += nominator_add.Add(two);
820       c += denominator_add.Add(two);
821       c += nominator.Mul(nominator_add);
822       c += denominator.Mul(denominator_add);
823       c += denominator_x.Add(two);
824       c += denominator2.Mul(two);
825     }
826
```

```
827
828     nominator_x_add.exponent.AddOne(); // *2
829     one.exponent.SubOne(); // =0.5
830     nominator_x_add.Pow(one); // =sqrt(nominator_x_add)
831     result.Mul(nominator_x_add);
832
833     one.Set05Pi();
834     one.Sub(result);
835
836   return one;
837   }
838
839
840   } // namespace auxiliaryfunctions
841
842
843   /*!
844     this function calculates the Arc Sine
845     x is from <-1,1>
846   */
847   template<class ValueType>
848   ValueType ASin(ValueType x, ErrorCode * err = 0)
849   {
850   using namespace auxiliaryfunctions;
851
852     ValueType result, one;
853     one.SetOne();
854     bool change_sign = false;
855
856     if( x.IsNan() )
857     {
858       if( err )
859         *err = err_improper_argument;
860
861     return x;
862     }
863
864     if( x.GreaterWithoutSignThan(one) )
865     {
866       if( err )
867         *err = err_improper_argument;
868
869       return result; // NaN is set by default
870     }
871
872     if( x.IsSign() )
873     {
874       change_sign = true;
875       x.Abs();
876     }
877
878     one.exponent.SubOne(); // =0.5
879
880     // asin(-x) = -asin(x)
881     if( x.GreaterWithoutSignThan(one) )
882       result = ASin_1(x);
883     else
884       result = ASin_0(x);
885
886     if( change_sign )
887       result.ChangeSign();
888
889     if( err )
890       *err = err_ok;
```

```
891
892   return result;
893   }
894
895
896   /*!
897     this function calculates the Arc Cosine
898
899     we're using the formula:
900     acos(x) = pi/2 - asin(x)
901   */
902   template<class ValueType>
903   ValueType ACos(const ValueType & x, ErrorCode * err = 0)
904   {
905   ValueType temp;
906
907     temp.Set05Pi();
908     temp.Sub(ASin(x, err));
909
910   return temp;
911   }
912
913
914
915   namespace auxiliaryfunctions
916   {
917
918   /*!
919     an auxiliary function for calculating the Arc Tangent
920
921     arc tan (x) where x is in <0; 0.5)
922     (x can be in (-0.5 ; 0.5) too)
923
924     we're using the Taylor series expanded in zero:
925     atan(x) = x - (x^3)/3 + (x^5)/5 - (x^7)/7 + ...
926   */
927   template<class ValueType>
928   ValueType ATan0(const ValueType & x)
929   {
930     ValueType nominator, denominator, nominator_add, denominator_add, temp;
931     ValueType result, old_result;
932     bool adding = false;
933     uint c = 0;
934
935     result        = x;
936     old_result    = result;
937     nominator     = x;
938     nominator_add = x;
939     nominator_add.Mul(x);
940
941     denominator.SetOne();
942     denominator_add = 2;
943
944     for(uint i=1 ; i<=TTMATH_ARITHMETIC_MAX_LOOP ; ++i)
945     {
946       c += nominator.Mul(nominator_add);
947       c += denominator.Add(denominator_add);
948
949       temp = nominator;
950       c += temp.Div(denominator);
951
952       if( c )
953         // the result should be ok
954         break;
```

```
955
956       if( adding )
957         result.Add(temp);
958       else
959         result.Sub(temp);
960
961       if( result == old_result )
962          // there's no sense to calculate more
963         break;
964
965       old_result = result;
966       adding     = !adding;
967     }
968
969   return result;
970   }
971
972
973   /*!
974     an auxiliary function for calculating the Arc Tangent
975
976     where x is in <0 ; 1>
977   */
978   template<class ValueType>
979   ValueType ATan01(const ValueType & x)
980   {
981     ValueType half;
982     half.Set05();
983
984     /*
985       it would be better if we chose about sqrt(2)-1=0.41... instead of 0.5
          here
986
987       because as you can see below:
988       when x = sqrt(2)-1
989       abs(x) = abs( (x-1)/(1+x) )
990       so when we're calculating values around x
991       then they will be better converged to each other
992
993       for example if we have x=0.4999 then during calculating ATan0(0.4999)
994       we have to make about 141 iterations but when we have x=0.5
995       then during calculating ATan0( (x-1)/(1+x) ) we have to make
996       only about 89 iterations (both for Big<3,9>)
997
998       in the future this 0.5 can be changed
999     */
1000    if( x.SmallerWithoutSignThan(half) )
1001      return ATan0(x);
1002
1003
1004    /*
1005      x>=0.5 and x<=1
1006      (x can be even smaller than 0.5)
1007
1008      y = atac(x)
1009      x = tan(y)
1010
1011      tan(y-b) = (tan(y)-tab(b)) / (1+tan(y)*tan(b))
1012      y-b      = atan( (tan(y)-tab(b)) / (1+tan(y)*tan(b)) )
1013      y        = b + atan( (x-tab(b)) / (1+x*tan(b)) )
1014
1015      let b = pi/4
1016      tan(b) = tan(pi/4) = 1
1017      y = pi/4 + atan( (x-1)/(1+x) )
```

```
1018
1019        so
1020        atac(x) = pi/4 + atan( (x-1)/(1+x) )
1021        when x->1 (x converges to 1) the (x-1)/(1+x) -> 0
1022        and we can use ATan0() function here
1023      */
1024
1025      ValueType n(x),d(x),one,result;
1026
1027      one.SetOne();
1028      n.Sub(one);
1029      d.Add(one);
1030      n.Div(d);
1031
1032      result = ATan0(n);
1033
1034      n.Set05Pi();
1035      n.exponent.SubOne(); // =pi/4
1036      result.Add(n);
1037
1038    return result;
1039    }
1040
1041
1042    /*!
1043      an auxiliary function for calculating the Arc Tangent
1044      where x > 1
1045
1046      we're using the formula:
1047      atan(x) = pi/2 - atan(1/x) for x>0
1048    */
1049    template<class ValueType>
1050    ValueType ATanGreaterThanPlusOne(const ValueType & x)
1051    {
1052    ValueType temp, atan;
1053
1054      temp.SetOne();
1055
1056      if( temp.Div(x) )
1057      {
1058        // if there was a carry here that means x is very big
1059        // and atan(1/x) fast converged to 0
1060        atan.SetZero();
1061      }
1062      else
1063        atan = ATan01(temp);
1064
1065      temp.Set05Pi();
1066      temp.Sub(atan);
1067
1068    return temp;
1069    }
1070
1071    } // namespace auxiliaryfunctions
1072
1073
1074    /*!
1075      this function calculates the Arc Tangent
1076    */
1077    template<class ValueType>
1078    ValueType ATan(ValueType x)
1079    {
1080    using namespace auxiliaryfunctions;
1081
```

```
1082      ValueType one, result;
1083      one.SetOne();
1084      bool change_sign = false;
1085
1086      if( x.IsNan() )
1087        return x;
1088
1089      // if x is negative we're using the formula:
1090      // atan(-x) = -atan(x)
1091      if( x.IsSign() )
1092      {
1093        change_sign = true;
1094        x.Abs();
1095      }
1096
1097      if( x.GreaterWithoutSignThan(one) )
1098        result = ATanGreaterThanPlusOne(x);
1099      else
1100        result = ATan01(x);
1101
1102      if( change_sign )
1103        result.ChangeSign();
1104
1105    return result;
1106    }
1107
1108
1109    /*!
1110      this function calculates the Arc Tangent
1111      look at the description of ATan(...)
1112
1113      (the abbreviation of Arc Tangent can be 'atg' as well)
1114    */
1115    template<class ValueType>
1116    ValueType ATg(const ValueType & x)
1117    {
1118      return ATan(x);
1119    }
1120
1121
1122    /*!
1123      this function calculates the Arc Cotangent
1124
1125      we're using the formula:
1126      actan(x) = pi/2 - atan(x)
1127    */
1128    template<class ValueType>
1129    ValueType ACot(const ValueType & x)
1130    {
1131    ValueType result;
1132
1133      result.Set05Pi();
1134      result.Sub(ATan(x));
1135
1136    return result;
1137    }
1138
1139
1140    /*!
1141      this function calculates the Arc Cotangent
1142      look at the description of ACot(...)
1143
1144      (the abbreviation of Arc Cotangent can be 'actg' as well)
1145    */
```

```
1146  template<class ValueType>
1147  ValueType ACtg(const ValueType & x)
1148  {
1149    return ACot(x);
1150  }
1151
1152
1153  /*
1154   *
1155   *  hyperbolic functions
1156   *
1157   *
1158   */
1159
1160
1161  /*!
1162    this function calculates the Hyperbolic Sine
1163
1164    we're using the formula sinh(x)= ( e^x - e^(-x) ) / 2
1165  */
1166  template<class ValueType>
1167  ValueType Sinh(const ValueType & x, ErrorCode * err = 0)
1168  {
1169    if( x.IsNan() )
1170    {
1171      if( err )
1172        *err = err_improper_argument;
1173
1174    return x; // NaN
1175    }
1176
1177    ValueType ex, emx;
1178    uint c = 0;
1179
1180    c += ex.Exp(x);
1181    c += emx.Exp(-x);
1182
1183    c += ex.Sub(emx);
1184    c += ex.exponent.SubOne();
1185
1186    if( err )
1187      *err = c ? err_overflow : err_ok;
1188
1189  return ex;
1190  }
1191
1192
1193  /*!
1194    this function calculates the Hyperbolic Cosine
1195
1196    we're using the formula cosh(x)= ( e^x + e^(-x) ) / 2
1197  */
1198  template<class ValueType>
1199  ValueType Cosh(const ValueType & x, ErrorCode * err = 0)
1200  {
1201    if( x.IsNan() )
1202    {
1203      if( err )
1204        *err = err_improper_argument;
1205
1206    return x; // NaN
1207    }
1208
1209    ValueType ex, emx;
```

```
1210    uint c = 0;
1211
1212    c += ex.Exp(x);
1213    c += emx.Exp(-x);
1214
1215    c += ex.Add(emx);
1216    c += ex.exponent.SubOne();
1217
1218    if( err )
1219      *err = c ? err_overflow : err_ok;
1220
1221  return ex;
1222  }
1223
1224
1225  /*!
1226    this function calculates the Hyperbolic Tangent
1227
1228    we're using the formula tanh(x)= ( e^x - e^(-x) ) / ( e^x + e^(-x) )
1229  */
1230  template<class ValueType>
1231  ValueType Tanh(const ValueType & x, ErrorCode * err = 0)
1232  {
1233    if( x.IsNan() )
1234    {
1235      if( err )
1236        *err = err_improper_argument;
1237
1238    return x; // NaN
1239    }
1240
1241    ValueType ex, emx, nominator, denominator;
1242    uint c = 0;
1243
1244    c += ex.Exp(x);
1245    c += emx.Exp(-x);
1246
1247    nominator = ex;
1248    c += nominator.Sub(emx);
1249    denominator = ex;
1250    c += denominator.Add(emx);
1251
1252    c += nominator.Div(denominator);
1253
1254    if( err )
1255      *err = c ? err_overflow : err_ok;
1256
1257  return nominator;
1258  }
1259
1260
1261  /*!
1262    this function calculates the Hyperbolic Tangent
1263    look at the description of Tanh(...)
1264
1265    (the abbreviation of Hyperbolic Tangent can be 'tgh' as well)
1266  */
1267  template<class ValueType>
1268  ValueType Tgh(const ValueType & x, ErrorCode * err = 0)
1269  {
1270    return Tanh(x, err);
1271  }
1272
1273  /*!
```

```
1274      this function calculates the Hyperbolic Cotangent
1275
1276      we're using the formula coth(x)= ( e^x + e^(-x) ) / ( e^x - e^(-x) )
1277    */
1278    template<class ValueType>
1279    ValueType Coth(const ValueType & x, ErrorCode * err = 0)
1280    {
1281      if( x.IsNan() )
1282      {
1283        if( err )
1284          *err = err_improper_argument;
1285
1286      return x; // NaN
1287      }
1288
1289      if( x.IsZero() )
1290      {
1291        if( err )
1292          *err = err_improper_argument;
1293
1294        return ValueType(); // NaN is set by default
1295      }
1296
1297      ValueType ex, emx, nominator, denominator;
1298      uint c = 0;
1299
1300      c += ex.Exp(x);
1301      c += emx.Exp(-x);
1302
1303      nominator = ex;
1304      c += nominator.Add(emx);
1305      denominator = ex;
1306      c += denominator.Sub(emx);
1307
1308      c += nominator.Div(denominator);
1309
1310      if( err )
1311        *err = c ? err_overflow : err_ok;
1312
1313    return nominator;
1314    }
1315
1316
1317    /*!
1318      this function calculates the Hyperbolic Cotangent
1319      look at the description of Coth(...)
1320
1321      (the abbreviation of Hyperbolic Cotangent can be 'ctgh' as well)
1322    */
1323    template<class ValueType>
1324    ValueType Ctgh(const ValueType & x, ErrorCode * err = 0)
1325    {
1326      return Coth(x, err);
1327    }
1328
1329
1330    /*
1331     *
1332     *  inverse hyperbolic functions
1333     *
1334     *
1335     */
1336
1337
```

```
1338  /*!
1339    inverse hyperbolic sine
1340
1341    asinh(x) = ln( x + sqrt(x^2 + 1) )
1342  */
1343  template<class ValueType>
1344  ValueType ASinh(const ValueType & x, ErrorCode * err = 0)
1345  {
1346    if( x.IsNan() )
1347    {
1348      if( err )
1349        *err = err_improper_argument;
1350
1351    return x; // NaN
1352    }
1353
1354    ValueType xx(x), one, result;
1355    uint c = 0;
1356    one.SetOne();
1357
1358    c += xx.Mul(x);
1359    c += xx.Add(one);
1360    one.exponent.SubOne(); // one=0.5
1361    // xx is >= 1
1362    c += xx.PowFrac(one); // xx=sqrt(xx)
1363    c += xx.Add(x);
1364    c += result.Ln(xx); // xx > 0
1365
1366    // here can only be a carry
1367    if( err )
1368      *err = c ? err_overflow : err_ok;
1369
1370  return result;
1371  }
1372
1373
1374  /*!
1375    inverse hyperbolic cosine
1376
1377    acosh(x) = ln( x + sqrt(x^2 - 1) )  x in <1, infinity)
1378  */
1379  template<class ValueType>
1380  ValueType ACosh(const ValueType & x, ErrorCode * err = 0)
1381  {
1382    if( x.IsNan() )
1383    {
1384      if( err )
1385        *err = err_improper_argument;
1386
1387    return x; // NaN
1388    }
1389
1390    ValueType xx(x), one, result;
1391    uint c = 0;
1392    one.SetOne();
1393
1394    if( x < one )
1395    {
1396      if( err )
1397        *err = err_improper_argument;
1398
1399    return result; // NaN is set by default
1400    }
1401
```

```
1402      c += xx.Mul(x);
1403      c += xx.Sub(one);
1404      // xx is >= 0
1405      // we can't call a PowFrac when the 'x' is zero
1406      // if x is 0 the sqrt(0) is 0
1407      if( !xx.IsZero() )
1408      {
1409        one.exponent.SubOne(); // one=0.5
1410        c += xx.PowFrac(one); // xx=sqrt(xx)
1411      }
1412      c += xx.Add(x);
1413      c += result.Ln(xx); // xx >= 1
1414
1415      // here can only be a carry
1416      if( err )
1417        *err = c ? err_overflow : err_ok;
1418
1419    return result;
1420    }
1421
1422
1423    /*!
1424      inverse hyperbolic tangent
1425
1426      atanh(x) = 0.5 * ln( (1+x) / (1-x) )  x in (-1, 1)
1427    */
1428    template<class ValueType>
1429    ValueType ATanh(const ValueType & x, ErrorCode * err = 0)
1430    {
1431      if( x.IsNan() )
1432      {
1433        if( err )
1434          *err = err_improper_argument;
1435
1436      return x; // NaN
1437      }
1438
1439      ValueType nominator(x), denominator, one, result;
1440      uint c = 0;
1441      one.SetOne();
1442
1443      if( !x.SmallerWithoutSignThan(one) )
1444      {
1445        if( err )
1446          *err = err_improper_argument;
1447
1448      return result; // NaN is set by default
1449      }
1450
1451      c += nominator.Add(one);
1452      denominator = one;
1453      c += denominator.Sub(x);
1454      c += nominator.Div(denominator);
1455      c += result.Ln(nominator);
1456      c += result.exponent.SubOne();
1457
1458      // here can only be a carry
1459      if( err )
1460        *err = c ? err_overflow : err_ok;
1461
1462    return result;
1463    }
1464
1465
```

```
1466  /*!
1467    inverse hyperbolic tantent
1468  */
1469  template<class ValueType>
1470  ValueType ATgh(const ValueType & x, ErrorCode * err = 0)
1471  {
1472    return ATanh(x, err);
1473  }
1474
1475
1476  /*!
1477    inverse hyperbolic cotangent
1478
1479    acoth(x) = 0.5 * ln( (x+1) / (x-1) )  x in (-infinity, -1) or (1, infinity)
1480  */
1481  template<class ValueType>
1482  ValueType ACoth(const ValueType & x, ErrorCode * err = 0)
1483  {
1484    if( x.IsNan() )
1485    {
1486      if( err )
1487        *err = err_improper_argument;
1488
1489    return x; // NaN
1490    }
1491
1492    ValueType nominator(x), denominator(x), one, result;
1493    uint c = 0;
1494    one.SetOne();
1495
1496    if( !x.GreaterWithoutSignThan(one) )
1497    {
1498      if( err )
1499        *err = err_improper_argument;
1500
1501    return result; // NaN is set by default
1502    }
1503
1504    c += nominator.Add(one);
1505    c += denominator.Sub(one);
1506    c += nominator.Div(denominator);
1507    c += result.Ln(nominator);
1508    c += result.exponent.SubOne();
1509
1510    // here can only be a carry
1511    if( err )
1512      *err = c ? err_overflow : err_ok;
1513
1514  return result;
1515  }
1516
1517
1518  /*!
1519    inverse hyperbolic cotantent
1520  */
1521  template<class ValueType>
1522  ValueType ACtgh(const ValueType & x, ErrorCode * err = 0)
1523  {
1524    return ACoth(x, err);
1525  }
1526
1527
1528
1529
```

```
1530
1531  /*
1532   *
1533   *  functions for converting between degrees, radians and gradians
1534   *
1535   *
1536   */
1537
1538
1539  /*!
1540    this function converts degrees to radians
1541
1542    it returns: x * pi / 180
1543  */
1544  template<class ValueType>
1545  ValueType DegToRad(const ValueType & x, ErrorCode * err = 0)
1546  {
1547  ValueType result, temp;
1548  uint c = 0;
1549
1550    if( x.IsNan() )
1551    {
1552      if( err )
1553        *err = err_improper_argument;
1554
1555    return x;
1556    }
1557
1558    result = x;
1559
1560    // it is better to make division first and then multiplication
1561    // the result is more accurate especially when x is: 90,180,270 or 360
1562    temp = 180;
1563    c += result.Div(temp);
1564
1565    temp.SetPi();
1566    c += result.Mul(temp);
1567
1568    if( err )
1569      *err = c ? err_overflow : err_ok;
1570
1571  return result;
1572  }
1573
1574
1575  /*!
1576    this function converts radians to degrees
1577
1578    it returns: x * 180 / pi
1579  */
1580  template<class ValueType>
1581  ValueType RadToDeg(const ValueType & x, ErrorCode * err = 0)
1582  {
1583  ValueType result, delimiter;
1584  uint c = 0;
1585
1586    if( x.IsNan() )
1587    {
1588      if( err )
1589        *err = err_improper_argument;
1590
1591    return x;
1592    }
1593
```

```
1594      result = 180;
1595      c += result.Mul(x);
1596
1597      delimiter.SetPi();
1598      c += result.Div(delimiter);
1599
1600      if( err )
1601        *err = c ? err_overflow : err_ok;
1602
1603    return result;
1604    }
1605
1606
1607    /*!
1608      this function converts degrees in the long format into one value
1609
1610      long format: (degrees, minutes, seconds)
1611      minutes and seconds must be greater than or equal zero
1612
1613      result:
1614      if d>=0 : result= d + ((s/60)+m)/60
1615      if d<0  : result= d - ((s/60)+m)/60
1616
1617      ((s/60)+m)/60 = (s+60*m)/3600 (second version is faster because
1618      there's only one division)
1619
1620      for example:
1621      DegToDeg(10, 30, 0) = 10.5
1622      DegToDeg(10, 24, 35.6)=10.4098(8)
1623    */
1624    template<class ValueType>
1625    ValueType DegToDeg( const ValueType & d, const ValueType & m, const ValueType
         & s,
1626                ErrorCode * err = 0)
1627    {
1628    ValueType delimiter, multipler;
1629    uint c = 0;
1630
1631      if( d.IsNan() || m.IsNan() || s.IsNan() || m.IsSign() || s.IsSign() )
1632      {
1633        if( err )
1634          *err = err_improper_argument;
1635
1636        delimiter.SetZeroNan(); // not needed, only to get rid of GCC warning
             about an uninitialized variable
1637
1638      return delimiter;
1639      }
1640
1641      multipler = 60;
1642      delimiter = 3600;
1643
1644      c += multipler.Mul(m);
1645      c += multipler.Add(s);
1646      c += multipler.Div(delimiter);
1647
1648      if( d.IsSign() )
1649        multipler.ChangeSign();
1650
1651      c += multipler.Add(d);
1652
1653      if( err )
1654        *err = c ? err_overflow : err_ok;
1655
```

```
1656  return multipler;
1657  }
1658
1659
1660  /*!
1661    this function converts degrees in the long format to radians
1662  */
1663  template<class ValueType>
1664  ValueType DegToRad( const ValueType & d, const ValueType & m, const ValueType
          & s,
1665              ErrorCode * err = 0)
1666  {
1667    ValueType temp_deg = DegToDeg(d,m,s,err);
1668
1669    if( err && *err!=err_ok )
1670      return temp_deg;
1671
1672  return DegToRad(temp_deg, err);
1673  }
1674
1675
1676  /*!
1677    this function converts gradians to radians
1678
1679    it returns: x * pi / 200
1680  */
1681  template<class ValueType>
1682  ValueType GradToRad(const ValueType & x, ErrorCode * err = 0)
1683  {
1684  ValueType result, temp;
1685  uint c = 0;
1686
1687    if( x.IsNan() )
1688    {
1689      if( err )
1690        *err = err_improper_argument;
1691
1692    return x;
1693    }
1694
1695    result = x;
1696
1697    // it is better to make division first and then multiplication
1698    // the result is more accurate especially when x is: 100,200,300 or 400
1699    temp = 200;
1700    c += result.Div(temp);
1701
1702    temp.SetPi();
1703    c += result.Mul(temp);
1704
1705    if( err )
1706      *err = c ? err_overflow : err_ok;
1707
1708  return result;
1709  }
1710
1711
1712  /*!
1713    this function converts radians to gradians
1714
1715    it returns: x * 200 / pi
1716  */
1717  template<class ValueType>
1718  ValueType RadToGrad(const ValueType & x, ErrorCode * err = 0)
```

```
1719  {
1720  ValueType result, delimiter;
1721  uint c = 0;
1722
1723    if( x.IsNan() )
1724    {
1725      if( err )
1726        *err = err_improper_argument;
1727
1728    return x;
1729    }
1730
1731    result = 200;
1732    c += result.Mul(x);
1733
1734    delimiter.SetPi();
1735    c += result.Div(delimiter);
1736
1737    if( err )
1738      *err = c ? err_overflow : err_ok;
1739
1740  return result;
1741  }
1742
1743
1744  /*!
1745    this function converts degrees to gradians
1746
1747    it returns: x * 200 / 180
1748  */
1749  template<class ValueType>
1750  ValueType DegToGrad(const ValueType & x, ErrorCode * err = 0)
1751  {
1752  ValueType result, temp;
1753  uint c = 0;
1754
1755    if( x.IsNan() )
1756    {
1757      if( err )
1758        *err = err_improper_argument;
1759
1760    return x;
1761    }
1762
1763    result = x;
1764
1765    temp = 200;
1766    c += result.Mul(temp);
1767
1768    temp = 180;
1769    c += result.Div(temp);
1770
1771    if( err )
1772      *err = c ? err_overflow : err_ok;
1773
1774  return result;
1775  }
1776
1777
1778  /*!
1779    this function converts degrees in the long format to gradians
1780  */
1781  template<class ValueType>
```

```
1782  ValueType DegToGrad( const ValueType & d, const ValueType & m, const
          ValueType & s,
1783                ErrorCode * err = 0)
1784  {
1785    ValueType temp_deg = DegToDeg(d,m,s,err);
1786
1787    if( err && *err!=err_ok )
1788      return temp_deg;
1789
1790  return DegToGrad(temp_deg, err);
1791  }
1792
1793
1794  /*!
1795    this function converts degrees to gradians
1796
1797    it returns: x * 180 / 200
1798  */
1799  template<class ValueType>
1800  ValueType GradToDeg(const ValueType & x, ErrorCode * err = 0)
1801  {
1802  ValueType result, temp;
1803  uint c = 0;
1804
1805    if( x.IsNan() )
1806    {
1807      if( err )
1808        *err = err_improper_argument;
1809
1810    return x;
1811    }
1812
1813    result = x;
1814
1815    temp = 180;
1816    c += result.Mul(temp);
1817
1818    temp = 200;
1819    c += result.Div(temp);
1820
1821    if( err )
1822      *err = c ? err_overflow : err_ok;
1823
1824  return result;
1825  }
1826
1827
1828
1829
1830  /*
1831   *
1832   *  another functions
1833   *
1834   *
1835   */
1836
1837
1838  /*!
1839    this function calculates the square root
1840
1841    Sqrt(9) = 3
1842  */
1843  template<class ValueType>
1844  ValueType Sqrt(ValueType x, ErrorCode * err = 0)
```

```
1845  {
1846    if( x.IsNan() || x.IsSign() )
1847    {
1848      if( err )
1849        *err = err_improper_argument;
1850
1851      x.SetNan();
1852
1853    return x;
1854    }
1855
1856    uint c = x.Sqrt();
1857
1858    if( err )
1859      *err = c ? err_overflow : err_ok;
1860
1861  return x;
1862  }
1863
1864
1865
1866  namespace auxiliaryfunctions
1867  {
1868
1869  template<class ValueType>
1870  bool RootCheckIndexSign(ValueType & x, const ValueType & index, ErrorCode *
         err)
1871  {
1872    if( index.IsSign() )
1873    {
1874      // index cannot be negative
1875      if( err )
1876        *err = err_improper_argument;
1877
1878      x.SetNan();
1879
1880    return true;
1881    }
1882
1883  return false;
1884  }
1885
1886
1887  template<class ValueType>
1888  bool RootCheckIndexZero(ValueType & x, const ValueType & index, ErrorCode *
         err)
1889  {
1890    if( index.IsZero() )
1891    {
1892      if( x.IsZero() )
1893      {
1894        // there isn't root(0;0) - we assume it's not defined
1895        if( err )
1896          *err = err_improper_argument;
1897
1898        x.SetNan();
1899
1900      return true;
1901      }
1902
1903      // root(x;0) is 1 (if x!=0)
1904      x.SetOne();
1905
1906      if( err )
```

```
1907          *err = err_ok;
1908
1909        return true;
1910        }
1911
1912      return false;
1913      }
1914
1915
1916      template<class ValueType>
1917      bool RootCheckIndexOne(const ValueType & index, ErrorCode * err)
1918      {
1919        ValueType one;
1920        one.SetOne();
1921
1922        if( index == one )
1923        {
1924          //root(x;1) is x
1925          // we do it because if we used the PowFrac function
1926          // we would lose the precision
1927          if( err )
1928            *err = err_ok;
1929
1930        return true;
1931        }
1932
1933      return false;
1934      }
1935
1936
1937      template<class ValueType>
1938      bool RootCheckIndexTwo(ValueType & x, const ValueType & index, ErrorCode *
           err)
1939      {
1940        if( index == 2 )
1941        {
1942          x = Sqrt(x, err);
1943
1944        return true;
1945        }
1946
1947      return false;
1948      }
1949
1950
1951      template<class ValueType>
1952      bool RootCheckIndexFrac(ValueType & x, const ValueType & index, ErrorCode *
           err)
1953      {
1954        if( !index.IsInteger() )
1955        {
1956          // index must be integer
1957          if( err )
1958            *err = err_improper_argument;
1959
1960          x.SetNan();
1961
1962        return true;
1963        }
1964
1965      return false;
1966      }
1967
1968
```

```
1969  template<class ValueType>
1970  bool RootCheckXZero(ValueType & x, ErrorCode * err)
1971  {
1972    if( x.IsZero() )
1973    {
1974      // root(0;index) is zero (if index!=0)
1975      // RootCheckIndexZero() must be called beforehand
1976      x.SetZero();
1977
1978      if( err )
1979        *err = err_ok;
1980
1981    return true;
1982    }
1983
1984  return false;
1985  }
1986
1987
1988  template<class ValueType>
1989  bool RootCheckIndex(ValueType & x, const ValueType & index, ErrorCode * err,
       bool * change_sign)
1990  {
1991    *change_sign = false;
1992
1993    if( index.Mod2() )
1994    {
1995      // index is odd (1,3,5...)
1996      if( x.IsSign() )
1997      {
1998        *change_sign = true;
1999        x.Abs();
2000      }
2001    }
2002    else
2003    {
2004      // index is even
2005      // x cannot be negative
2006      if( x.IsSign() )
2007      {
2008        if( err )
2009          *err = err_improper_argument;
2010
2011        x.SetNan();
2012
2013        return true;
2014      }
2015    }
2016
2017  return false;
2018  }
2019
2020
2021  template<class ValueType>
2022  uint RootCorrectInteger(ValueType & old_x, ValueType & x, const ValueType &
       index)
2023  {
2024    if( !old_x.IsInteger() || x.IsInteger() || !index.exponent.IsSign() )
2025      return 0;
2026
2027    // old_x is integer,
2028    // x is not integer,
2029    // index is relatively small (index.exponent<0 or index.exponent<=0)
2030    // (because we're using a special powering algorithm Big::PowUInt())
```

```
2031
2032      uint c = 0;
2033
2034      ValueType temp(x);
2035      c += temp.Round();
2036
2037      ValueType temp_round(temp);
2038      c += temp.PowUInt(index);
2039
2040      if( temp == old_x )
2041        x = temp_round;
2042
2043    return (c==0)? 0 : 1;
2044    }
2045
2046
2047
2048    } // namespace auxiliaryfunctions
2049
2050
2051
2052    /*!
2053      indexth Root of x
2054      index must be integer and not negative  <0;1;2;3....)
2055
2056      if index==0 the result is one
2057      if x==0 the result is zero and we assume root(0;0) is not defined
2058
2059      if index is even (2;4;6...) the result is x^(1/index) and x>0
2060      if index is odd (1;2;3;...) the result is either
2061        -(abs(x)^(1/index)) if x<0    or
2062              x^(1/index)) if x>0
2063
2064      (for index==1 the result is equal x)
2065    */
2066    template<class ValueType>
2067    ValueType Root(ValueType x, const ValueType & index, ErrorCode * err = 0)
2068    {
2069      using namespace auxiliaryfunctions;
2070
2071      if( x.IsNan() || index.IsNan() )
2072      {
2073        if( err )
2074          *err = err_improper_argument;
2075
2076        x.SetNan();
2077
2078      return x;
2079      }
2080
2081      if( RootCheckIndexSign(x, index, err) ) return x;
2082      if( RootCheckIndexZero(x, index, err) ) return x;
2083      if( RootCheckIndexOne (   index, err) ) return x;
2084      if( RootCheckIndexTwo (x, index, err) ) return x;
2085      if( RootCheckIndexFrac(x, index, err) ) return x;
2086      if( RootCheckXZero    (x,        err) ) return x;
2087
2088      // index integer and index!=0
2089      // x!=0
2090
2091      ValueType old_x(x);
2092      bool change_sign;
2093
2094      if( RootCheckIndex(x, index, err, &change_sign ) ) return x;
```

```
2095
2096      ValueType temp;
2097      uint c = 0;
2098
2099      // we're using the formula: root(x ; n) = exp( ln(x) / n )
2100      c += temp.Ln(x);
2101      c += temp.Div(index);
2102      c += x.Exp(temp);
2103
2104      if( change_sign )
2105      {
2106        // x is different from zero
2107        x.SetSign();
2108      }
2109
2110      c += RootCorrectInteger(old_x, x, index);
2111
2112      if( err )
2113        *err = c ? err_overflow : err_ok;
2114
2115    return x;
2116    }
2117
2118
2119
2120    /*!
2121      absolute value of x
2122      e.g.  -2 = 2
2123             2 = 2
2124    */
2125    template<class ValueType>
2126    ValueType Abs(const ValueType & x)
2127    {
2128      ValueType result( x );
2129      result.Abs();
2130
2131    return result;
2132    }
2133
2134
2135    /*!
2136      it returns the sign of the value
2137      e.g.  -2 = -1
2138             0 = 0
2139            10 = 1
2140    */
2141    template<class ValueType>
2142    ValueType Sgn(ValueType x)
2143    {
2144      x.Sgn();
2145
2146    return x;
2147    }
2148
2149
2150    /*!
2151      the remainder from a division
2152
2153      e.g.
2154      mod( 12.6 ;  3) =  0.6   because 12.6  = 3*4 + 0.6
2155      mod(-12.6 ;  3) = -0.6   bacause -12.6 = 3*(-4) + (-0.6)
2156      mod( 12.6 ; -3) =  0.6
2157      mod(-12.6 ; -3) = -0.6
2158    */
```

```
2159  template<class ValueType>
2160  ValueType Mod(ValueType a, const ValueType & b, ErrorCode * err = 0)
2161  {
2162    if( a.IsNan() || b.IsNan() )
2163    {
2164      if( err )
2165        *err = err_improper_argument;
2166
2167      a.SetNan();
2168
2169    return a;
2170    }
2171
2172    uint c = a.Mod(b);
2173
2174    if( err )
2175      *err = c ? err_overflow : err_ok;
2176
2177  return a;
2178  }
2179
2180
2181
2182  namespace auxiliaryfunctions
2183  {
2184
2185  /*!
2186    this function is used to store factorials in a given container
2187    'more' means how many values should be added at the end
2188
2189      e.g.
2190      std::vector<ValueType> fact;
2191      SetFactorialSequence(fact, 3);
2192      // now the container has three values: 1  1  2
2193
2194      SetFactorialSequence(fact, 2);
2195      // now the container has five values:  1  1  2  6  24
2196  */
2197  template<class ValueType>
2198  void SetFactorialSequence(std::vector<ValueType> & fact, uint more = 20)
2199  {
2200    if( more == 0 )
2201      more = 1;
2202
2203    uint start = static_cast<uint>(fact.size());
2204    fact.resize(fact.size() + more);
2205
2206    if( start == 0 )
2207    {
2208      fact[0] = 1;
2209      ++start;
2210    }
2211
2212    for(uint i=start ; i<fact.size() ; ++i)
2213    {
2214      fact[i] = fact[i-1];
2215      fact[i].MulInt(i);
2216    }
2217  }
2218
2219
2220  /*!
2221    an auxiliary function used to calculate Bernoulli numbers
2222
```

```
2223    this function returns a sum:
2224    sum(m) = sum_{k=0}^{m-1} {2^k * (m k) * B(k)}     k in [0, m-1]    (m k)
            means binomial coefficient = (m! / (k! * (m-k)!))
2225
2226    you should have sufficient factorials in cgamma.fact
2227    (cgamma.fact should have at least m items)
2228
2229    n_ should be equal 2
2230  */
2231  template<class ValueType>
2232  ValueType SetBernoulliNumbersSum(CGamma<ValueType> & cgamma, const ValueType
        & n_, uint m,
2233                   const volatile StopCalculating * stop = 0)
2234  {
2235  ValueType k_, temp, temp2, temp3, sum;
2236
2237    sum.SetZero();
2238
2239    for(uint k=0 ; k<m ; ++k)      // k<m means k<=m-1
2240    {
2241      if( stop && (k & 15)==0 )    // means: k % 16 == 0
2242        if( stop->WasStopSignal() )
2243          return ValueType();    // NaN
2244
2245      if( k>1 && (k & 1) == 1 )    // for that k the Bernoulli number is zero
2246        continue;
2247
2248      k_ = k;
2249
2250      temp = n_;          // n_ is equal 2
2251      temp.Pow(k_);
2252      // temp = 2^k
2253
2254      temp2 = cgamma.fact[m];
2255      temp3 = cgamma.fact[k];
2256      temp3.Mul(cgamma.fact[m-k]);
2257      temp2.Div(temp3);
2258      // temp2 = (m k) = m! / ( k! * (m-k)! )
2259
2260      temp.Mul(temp2);
2261      temp.Mul(cgamma.bern[k]);
2262
2263      sum.Add(temp);
2264      // sum += 2^k * (m k) * B(k)
2265
2266      if( sum.IsNan() )
2267        break;
2268    }
2269
2270  return sum;
2271  }
2272
2273
2274  /*!
2275    an auxiliary function used to calculate Bernoulli numbers
2276    start is >= 2
2277
2278    we use the recurrence formula:
2279       B(m) = 1 / (2*(1 - 2^m)) * sum(m)
2280       where sum(m) is calculated by SetBernoulliNumbersSum()
2281  */
2282  template<class ValueType>
2283  bool SetBernoulliNumbersMore(CGamma<ValueType> & cgamma, uint start, const
       volatile StopCalculating * stop = 0)
```

```
2284  {
2285  ValueType denominator, temp, temp2, temp3, m_, sum, sum2, n_, k_;
2286
2287    const uint n = 2;
2288    n_ = n;
2289
2290    // start is >= 2
2291    for(uint m=start ; m<cgamma.bern.size() ; ++m)
2292    {
2293      if( (m & 1) == 1 )
2294      {
2295        cgamma.bern[m].SetZero();
2296      }
2297      else
2298      {
2299        m_ = m;
2300
2301        temp = n_;          // n_ = 2
2302        temp.Pow(m_);
2303        // temp = 2^m
2304
2305        denominator.SetOne();
2306        denominator.Sub(temp);
2307        if( denominator.exponent.AddOne() ) // it means: denominator.MulInt(2)
2308          denominator.SetNan();
2309
2310        // denominator = 2 * (1 - 2^m)
2311
2312        cgamma.bern[m] = SetBernoulliNumbersSum(cgamma, n_, m, stop);
2313
2314        if( stop && stop->WasStopSignal() )
2315        {
2316          cgamma.bern.resize(m);     // valid numbers are in [0, m-1]
2317          return false;
2318        }
2319
2320        cgamma.bern[m].Div(denominator);
2321      }
2322    }
2323
2324  return true;
2325  }
2326
2327
2328  /*!
2329    this function is used to calculate Bernoulli numbers,
2330    returns false if there was a stop signal,
2331    'more' means how many values should be added at the end
2332
2333      e.g.
2334      typedef Big<1,2> MyBig;
2335      CGamma<MyBig> cgamma;
2336      SetBernoulliNumbers(cgamma, 3);
2337      // now we have three first Bernoulli numbers:  1  -0.5  0.16667
2338
2339      SetBernoulliNumbers(cgamma, 4);
2340      // now we have 7 Bernoulli numbers:  1  -0.5  0.16667  0  -0.0333  0
           0.0238
2341  */
2342  template<class ValueType>
2343  bool SetBernoulliNumbers(CGamma<ValueType> & cgamma, uint more = 20, const
       volatile StopCalculating * stop = 0)
2344  {
2345    if( more == 0 )
```

```
2346        more = 1;
2347
2348      uint start = static_cast<uint>(cgamma.bern.size());
2349      cgamma.bern.resize(cgamma.bern.size() + more);
2350
2351      if( start == 0 )
2352      {
2353        cgamma.bern[0].SetOne();
2354        ++start;
2355      }
2356
2357      if( cgamma.bern.size() == 1 )
2358        return true;
2359
2360      if( start == 1 )
2361      {
2362        cgamma.bern[1].Set05();
2363        cgamma.bern[1].ChangeSign();
2364        ++start;
2365      }
2366
2367      // we should have sufficient factorials in cgamma.fact
2368      if( cgamma.fact.size() < cgamma.bern.size() )
2369        SetFactorialSequence(cgamma.fact, static_cast<uint>(cgamma.bern.size() -
             cgamma.fact.size()));
2370
2371
2372    return SetBernoulliNumbersMore(cgamma, start, stop);
2373    }
2374
2375
2376    /*!
2377      an auxiliary function used to calculate the Gamma() function
2378
2379      we calculate a sum:
2380         sum(n) = sum_{m=2} { B(m) / ( (m^2 - m) * n^(m-1) )  } = 1/(12*n) -
              1/(360*n^3) + 1/(1260*n^5) + ...
2381           B(m) means a mth Bernoulli number
2382         the sum starts from m=2, we calculate as long as the value will not
              change after adding a next part
2383    */
2384    template<class ValueType>
2385    ValueType GammaFactorialHighSum(const ValueType & n, CGamma<ValueType> &
         cgamma, ErrorCode & err,
2386                    const volatile StopCalculating * stop)
2387    {
2388    ValueType temp, temp2, denominator, sum, oldsum;
2389
2390      sum.SetZero();
2391
2392      for(uint m=2 ; m<TTMATH_ARITHMETIC_MAX_LOOP ; m+=2)
2393      {
2394        if( stop && (m & 3)==0 ) // (m & 3)==0 means: (m % 4)==0
2395          if( stop->WasStopSignal() )
2396          {
2397            err = err_interrupt;
2398            return ValueType(); // NaN
2399          }
2400
2401        temp = (m-1);
2402        denominator = n;
2403        denominator.Pow(temp);
2404        // denominator = n ^ (m-1)
2405
```

```
2406        temp = m;
2407        temp2 = temp;
2408        temp.Mul(temp2);
2409        temp.Sub(temp2);
2410        // temp = m^2 - m
2411
2412        denominator.Mul(temp);
2413        // denominator = (m^2 - m) * n ^ (m-1)
2414
2415        if( m >= cgamma.bern.size() )
2416        {
2417          if( !SetBernoulliNumbers(cgamma, m - cgamma.bern.size() + 1 + 3, stop)
                ) // 3 more than needed
2418          {
2419            // there was the stop signal
2420            err = err_interrupt;
2421            return ValueType(); // NaN
2422          }
2423        }
2424
2425        temp = cgamma.bern[m];
2426        temp.Div(denominator);
2427
2428        oldsum = sum;
2429        sum.Add(temp);
2430
2431        if( sum.IsNan() || oldsum==sum )
2432          break;
2433      }
2434
2435    return sum;
2436    }
2437
2438
2439    /*!
2440      an auxiliary function used to calculate the Gamma() function
2441
2442      we calculate a helper function GammaFactorialHigh() by using Stirling's
           series:
2443        n! = (n/e)^n * sqrt(2*pi*n) * exp( sum(n) )
2444        where n is a real number (not only an integer) and is sufficient large (
               greater than TTMATH_GAMMA_BOUNDARY)
2445        and sum(n) is calculated by GammaFactorialHighSum()
2446    */
2447    template<class ValueType>
2448    ValueType GammaFactorialHigh(const ValueType & n, CGamma<ValueType> & cgamma,
         ErrorCode & err,
2449                  const volatile StopCalculating * stop)
2450    {
2451    ValueType temp, temp2, temp3, denominator, sum;
2452
2453      temp.Set2Pi();
2454      temp.Mul(n);
2455      temp2 = Sqrt(temp);
2456      // temp2 = sqrt(2*pi*n)
2457
2458      temp = n;
2459      temp3.SetE();
2460      temp.Div(temp3);
2461      temp.Pow(n);
2462      // temp = (n/e)^n
2463
2464      sum = GammaFactorialHighSum(n, cgamma, err, stop);
2465      temp3.Exp(sum);
```

```
2466      // temp3 = exp(sum)
2467
2468      temp.Mul(temp2);
2469      temp.Mul(temp3);
2470
2471    return temp;
2472    }
2473
2474
2475    /*!
2476      an auxiliary function used to calculate the Gamma() function
2477
2478      Gamma(x) = GammaFactorialHigh(x-1)
2479    */
2480    template<class ValueType>
2481    ValueType GammaPlusHigh(ValueType n, CGamma<ValueType> & cgamma, ErrorCode &
         err, const volatile StopCalculating * stop)
2482    {
2483    ValueType one;
2484
2485      one.SetOne();
2486      n.Sub(one);
2487
2488    return GammaFactorialHigh(n, cgamma, err, stop);
2489    }
2490
2491
2492    /*!
2493      an auxiliary function used to calculate the Gamma() function
2494
2495      we use this function when n is integer and a small value (from 0 to
           TTMATH_GAMMA_BOUNDARY)
2496      we use the formula:
2497         gamma(n) = (n-1)! = 1 * 2 * 3 * ... * (n-1)
2498    */
2499    template<class ValueType>
2500    ValueType GammaPlusLowIntegerInt(uint n, CGamma<ValueType> & cgamma)
2501    {
2502      TTMATH_ASSERT( n > 0 )
2503
2504      if( n - 1 < static_cast<uint>(cgamma.fact.size()) )
2505        return cgamma.fact[n - 1];
2506
2507      ValueType res;
2508      uint start = 2;
2509
2510      if( cgamma.fact.size() < 2 )
2511      {
2512        res.SetOne();
2513      }
2514      else
2515      {
2516        start = static_cast<uint>(cgamma.fact.size());
2517        res   = cgamma.fact[start-1];
2518      }
2519
2520      for(uint i=start ; i<n ; ++i)
2521        res.MulInt(i);
2522
2523    return res;
2524    }
2525
2526
2527    /*!
```

```
2528    an auxiliary function used to calculate the Gamma() function
2529
2530    we use this function when n is integer and a small value (from 0 to
            TTMATH_GAMMA_BOUNDARY]
2531  */
2532  template<class ValueType>
2533  ValueType GammaPlusLowInteger(const ValueType & n, CGamma<ValueType> & cgamma
        )
2534  {
2535  sint n_;
2536
2537    n.ToInt(n_);
2538
2539  return GammaPlusLowIntegerInt(n_, cgamma);
2540  }
2541
2542
2543  /*!
2544    an auxiliary function used to calculate the Gamma() function
2545
2546    we use this function when n is a small value (from 0 to
            TTMATH_GAMMA_BOUNDARY]
2547    we use a recurrence formula:
2548       gamma(z+1) = z * gamma(z)
2549       then: gamma(z) = gamma(z+1) / z
2550
2551       e.g.
2552       gamma(3.89) = gamma(2001.89) / ( 3.89 * 4.89 * 5.89 * ... * 1999.89 *
               2000.89 )
2553  */
2554  template<class ValueType>
2555  ValueType GammaPlusLow(ValueType n, CGamma<ValueType> & cgamma, ErrorCode &
        err, const volatile StopCalculating * stop)
2556  {
2557  ValueType one, denominator, temp, boundary;
2558
2559    if( n.IsInteger() )
2560      return GammaPlusLowInteger(n, cgamma);
2561
2562    one.SetOne();
2563    denominator = n;
2564    boundary    = TTMATH_GAMMA_BOUNDARY;
2565
2566    while( n < boundary )
2567    {
2568      n.Add(one);
2569      denominator.Mul(n);
2570    }
2571
2572    n.Add(one);
2573
2574    // now n is sufficient big
2575    temp = GammaPlusHigh(n, cgamma, err, stop);
2576    temp.Div(denominator);
2577
2578  return temp;
2579  }
2580
2581
2582  /*!
2583    an auxiliary function used to calculate the Gamma() function
2584  */
2585  template<class ValueType>
```

```
2586  ValueType GammaPlus(const ValueType & n, CGamma<ValueType> & cgamma,
          ErrorCode & err, const volatile StopCalculating * stop)
2587  {
2588    if( n > TTMATH_GAMMA_BOUNDARY )
2589      return GammaPlusHigh(n, cgamma, err, stop);
2590
2591  return GammaPlusLow(n, cgamma, err, stop);
2592  }
2593
2594
2595  /*!
2596    an auxiliary function used to calculate the Gamma() function
2597
2598    this function is used when n is negative
2599    we use the reflection formula:
2600       gamma(1-z) * gamma(z) = pi / sin(pi*z)
2601       then: gamma(z) = pi / (sin(pi*z) * gamma(1-z))
2602
2603  */
2604  template<class ValueType>
2605  ValueType GammaMinus(const ValueType & n, CGamma<ValueType> & cgamma,
          ErrorCode & err, const volatile StopCalculating * stop)
2606  {
2607  ValueType pi, denominator, temp, temp2;
2608
2609    if( n.IsInteger() )
2610    {
2611      // gamma function is not defined when n is negative and integer
2612      err = err_improper_argument;
2613      return temp; // NaN
2614    }
2615
2616    pi.SetPi();
2617
2618    temp = pi;
2619    temp.Mul(n);
2620    temp2 = Sin(temp);
2621    // temp2 = sin(pi * n)
2622
2623    temp.SetOne();
2624    temp.Sub(n);
2625    temp = GammaPlus(temp, cgamma, err, stop);
2626    // temp = gamma(1 - n)
2627
2628    temp.Mul(temp2);
2629    pi.Div(temp);
2630
2631  return pi;
2632  }
2633
2634  } // namespace auxiliaryfunctions
2635
2636
2637
2638  /*!
2639    this function calculates the Gamma function
2640
2641    it's multithread safe, you should create a CGamma<> object and use it
          whenever you call the Gamma()
2642    e.g.
2643      typedef Big<1,2> MyBig;
2644      MyBig x=234, y=345.53;
2645      CGamma<MyBig> cgamma;
2646      std::cout << Gamma(x, cgamma) << std::endl;
```

```
2647      std::cout << Gamma(y, cgamma) << std::endl;
2648    in the CGamma<> object the function stores some coefficients (factorials,
          Bernoulli numbers),
2649    and they will be reused in next calls to the function
2650
2651    each thread should have its own CGamma<> object, and you can use these
          objects with Factorial() function too
2652  */
2653  template<class ValueType>
2654  ValueType Gamma(const ValueType & n, CGamma<ValueType> & cgamma, ErrorCode *
      err = 0,
2655          const volatile StopCalculating * stop = 0)
2656  {
2657  using namespace auxiliaryfunctions;
2658
2659  ValueType result;
2660  ErrorCode err_tmp;
2661
2662    if( n.IsNan() )
2663    {
2664      if( err )
2665        *err = err_improper_argument;
2666
2667    return n;
2668    }
2669
2670    if( cgamma.history.Get(n, result, err_tmp) )
2671    {
2672      if( err )
2673        *err = err_tmp;
2674
2675      return result;
2676    }
2677
2678    err_tmp = err_ok;
2679
2680    if( n.IsSign() )
2681    {
2682      result = GammaMinus(n, cgamma, err_tmp, stop);
2683    }
2684    else
2685    if( n.IsZero() )
2686    {
2687      err_tmp = err_improper_argument;
2688      result.SetNan();
2689    }
2690    else
2691    {
2692      result = GammaPlus(n, cgamma, err_tmp, stop);
2693    }
2694
2695    if( result.IsNan() && err_tmp==err_ok )
2696      err_tmp = err_overflow;
2697
2698    if( err )
2699      *err = err_tmp;
2700
2701    if( stop && !stop->WasStopSignal() )
2702      cgamma.history.Add(n, result, err_tmp);
2703
2704  return result;
2705  }
2706
2707
```

```
2708  /*!
2709    this function calculates the Gamma function
2710
2711    note: this function should be used only in a single-thread environment
2712  */
2713  template<class ValueType>
2714  ValueType Gamma(const ValueType & n, ErrorCode * err = 0)
2715  {
2716  // warning: this static object is not thread safe
2717  static CGamma<ValueType> cgamma;
2718
2719  return Gamma(n, cgamma, err);
2720  }
2721
2722
2723
2724  namespace auxiliaryfunctions
2725  {
2726
2727  /*!
2728    an auxiliary function for calculating the factorial function
2729
2730    we use the formula:
2731       x! = gamma(x+1)
2732  */
2733  template<class ValueType>
2734  ValueType Factorial2(ValueType x,
2735             CGamma<ValueType> * cgamma = 0,
2736             ErrorCode * err = 0,
2737             const volatile StopCalculating * stop = 0)
2738  {
2739  ValueType result, one;
2740
2741    if( x.IsNan() || x.IsSign() || !x.IsInteger() )
2742    {
2743      if( err )
2744        *err = err_improper_argument;
2745
2746      x.SetNan();
2747
2748    return x;
2749    }
2750
2751    one.SetOne();
2752    x.Add(one);
2753
2754    if( cgamma )
2755      return Gamma(x, *cgamma, err, stop);
2756
2757  return Gamma(x, err);
2758  }
2759
2760  } // namespace auxiliaryfunctions
2761
2762
2763
2764  /*!
2765    the factorial from given 'x'
2766    e.g.
2767    Factorial(4) = 4! = 1*2*3*4
2768
2769    it's multithread safe, you should create a CGamma<> object and use it
         whenever you call the Factorial()
2770    e.g.
```

```
2771        typedef Big<1,2> MyBig;
2772        MyBig x=234, y=54345;
2773        CGamma<MyBig> cgamma;
2774        std::cout << Factorial(x, cgamma) << std::endl;
2775        std::cout << Factorial(y, cgamma) << std::endl;
2776      in the CGamma<> object the function stores some coefficients (factorials, Bernoulli numbers),
2777      and they will be reused in next calls to the function
2778
2779      each thread should have its own CGamma<> object, and you can use these objects with Gamma() function too
2780    */
2781    template<class ValueType>
2782    ValueType Factorial(const ValueType & x, CGamma<ValueType> & cgamma, ErrorCode * err = 0,
2783                const volatile StopCalculating * stop = 0)
2784    {
2785      return auxiliaryfunctions::Factorial2(x, &cgamma, err, stop);
2786    }
2787
2788
2789    /*!
2790      the factorial from given 'x'
2791      e.g.
2792      Factorial(4) = 4! = 1*2*3*4
2793
2794      note: this function should be used only in a single-thread environment
2795    */
2796    template<class ValueType>
2797    ValueType Factorial(const ValueType & x, ErrorCode * err = 0)
2798    {
2799      return auxiliaryfunctions::Factorial2(x, (CGamma<ValueType>*)0, err, 0);
2800    }
2801
2802
2803    /*!
2804      this method prepares some coefficients: factorials and Bernoulli numbers
2805      stored in 'fact' and 'bern' objects
2806
2807      we're defining the method here because we're using Gamma() function which
2808      is not available in ttmathobjects.h
2809
2810      read the doc info in ttmathobjects.h file where CGamma<> struct is declared
2811    */
2812    template<class ValueType>
2813    void CGamma<ValueType>::InitAll()
2814    {
2815      ValueType x = TTMATH_GAMMA_BOUNDARY + 1;
2816
2817      // history.Remove(x) removes only one object
2818      // we must be sure that there are not others objects with the key 'x'
2819      while( history.Remove(x) )
2820      {
2821      }
2822
2823      // the simplest way to initialize is to call the Gamma function with (TTMATH_GAMMA_BOUNDARY + 1)
2824      // when x is larger then fewer coefficients we need
2825      Gamma(x, *this);
2826    }
2827
2828
2829
2830  } // namespace
```

```
2831
2832
2833 /*!
2834   this is for convenience for the user
2835   he can only use '#include <ttmath/ttmath.h>'
2836 */
2837 #include "ttmathparser.h"
2838
2839 // Dec is not finished yet
2840 //#include "ttmathdec.h"
2841
2842
2843
2844 #ifdef _MSC_VER
2845 //warning C4127: conditional expression is constant
2846 #pragma warning( default: 4127 )
2847 //warning C4702: unreachable code
2848 #pragma warning( default: 4702 )
2849 //warning C4800: forcing value to bool 'true' or 'false' (performance warning)
2850 #pragma warning( default: 4800 )
2851 #endif
2852
2853 #endif
```

- ExtraFunctions.h:

```
1  #include <string>
2  #include <iostream>
3  #include <sstream>
4  #include <vector>
5
6  using namespace std;
7
8   std::vector<std::string> split_getline(std::string str, char delim) {
9      std::vector<std::string> resultado;
10     std::istringstream isstream(str);
11     std::string palabra;
12
13     while(std::getline(isstream, palabra, delim)){
14         resultado.push_back(palabra);
15     }
16
17     return resultado;
18 }
```